Master Thesis

Automatic contract extraction
developing a CIL parser

Author(s):
Marti, Christof

Publication Date:
2003

Permanent Link:
https://doi.org/10.3929/ethz-a-005115115

Rights / License:
In Copyright - Non-Commercial Use Permitted
Automatic Contract Extraction: 
Developing a CIL Parser

Diploma Thesis Report 

Christof Marti 

Supervision by Prof. Dr. Bertrand Meyer and Karine Arnout 
Chair of Software Engineering - ETH Zürich 

25 September 2003
Abstract

The analysis of .NET libraries in [1] suggests the development of a tool for automatic contract extraction from .NET classes. The article observes that preconditions tend to be hidden under explicit exception cases. A tool leveraging this observation has been developed as part of this diploma thesis and is documented in this report. Although the chosen approach is limited to elementary cases, the application of the tool to classes ArrayList, Stack and Queue of the .NET framework [16] reveals that, in these classes, half or more of all explicit exception cases can be addressed and the corresponding preconditions are extracted by the current implementation. The report includes the documentation of the tool’s implementation, a presentation of the results and a discussion of limitations and extensions of the present approach and implementation.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Overview</td>
<td>8</td>
</tr>
<tr>
<td>2 Scanner</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Declarations</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Rules</td>
<td>10</td>
</tr>
<tr>
<td>2.3 User Code</td>
<td>12</td>
</tr>
<tr>
<td>3 Parser</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Declarations</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Rules</td>
<td>14</td>
</tr>
<tr>
<td>3.3 User Code</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Shift/Reduce Conflicts</td>
<td>15</td>
</tr>
<tr>
<td>4 Abstract Syntax Tree</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Module</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Entities</td>
<td>19</td>
</tr>
<tr>
<td>4.3 Type Members</td>
<td>20</td>
</tr>
<tr>
<td>4.4 Entities and Type Members</td>
<td>23</td>
</tr>
<tr>
<td>4.5 Method Body Items</td>
<td>25</td>
</tr>
<tr>
<td>4.6 Entities, Type Members and Method Body Items</td>
<td>37</td>
</tr>
<tr>
<td>4.7 Method Signatures</td>
<td>40</td>
</tr>
<tr>
<td>4.8 Miscellaneous</td>
<td>41</td>
</tr>
<tr>
<td>4.9 Referable Nodes</td>
<td>42</td>
</tr>
<tr>
<td>4.10 Referencing Nodes</td>
<td>43</td>
</tr>
<tr>
<td>4.11 Names, Identifiers and Strings</td>
<td>45</td>
</tr>
<tr>
<td>4.12 Factory</td>
<td>46</td>
</tr>
<tr>
<td>4.13 Visitors</td>
<td>47</td>
</tr>
<tr>
<td>4.14 Annotation</td>
<td>48</td>
</tr>
</tbody>
</table>
5 Contract Extraction
5.1 High-Level Description .......................... 49
5.2 Directed Graphs ................................. 71
5.3 Expressions ................................... 74
5.4 Basic Block Analysis ............................. 84
5.5 Basic Block Classification ......................... 88
5.6 Code Path Extraction ............................. 91
5.7 Symbolic Execution ............................... 91
5.8 Precondition Extraction .......................... 94
5.9 Results ...................................... 96

6 Assessment .................................. 100
6.1 Limitations .................................. 100
6.2 Future Work .................................. 102

7 Conclusion .................................. 104

A Glossary .................................. 105

Bibliography .................................. 106
List of Figures

5.1 Vertex Class Diagram ........................................ 72
5.2 Edge Class Diagram ........................................ 73
5.3 Path Traversal Algorithm Class Diagram ................. 74
5.4 Path Traversal Strategy Class Diagram .................. 75
5.5 Expression Class Diagram ...................................... 77
5.6 Identity Expression Class Diagram .......................... 80
5.7 Void Expression Class Diagram ............................... 81
5.8 Expression Visitor Class Diagram ........................... 82
5.9 Basic Block Analysis Visitor Class Diagram .............. 86
5.10 Symbolic Execution Visitor Class Diagram ............... 93
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Examples of extracted precondition clauses from <code>ArrayList</code></td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Deferred descendants of <code>CIL_DATA_INSTRUCTION</code></td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Effective descendants of <code>CIL_DATA_INSTRUCTION</code></td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>Descendants of <code>CIL_BRANCH_INSTRUCTION</code></td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Descendants of <code>CIL_LOCAL_VARIABLE_INSTRUCTION</code></td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Descendants of <code>CIL_PARAMETER_VARIABLE_INSTRUCTION</code></td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>Descendants of <code>CIL_NONE_INSTRUCTION</code></td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>Descendants of <code>CIL_SIGNATURE_INSTRUCTION</code></td>
<td>33</td>
</tr>
<tr>
<td>4.8</td>
<td>Descendants of <code>CIL_FIELD_INSTRUCTION</code></td>
<td>34</td>
</tr>
<tr>
<td>4.9</td>
<td>Descendants of <code>CIL_METHOD_INSTRUCTION</code></td>
<td>34</td>
</tr>
<tr>
<td>4.10</td>
<td>Descendants of <code>CIL_TYPE_INSTRUCTION</code></td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Deferred routines of <code>CIL_EXPRESSION</code></td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Effective factory queries of <code>CIL_EXPRESSION</code></td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>Logic expression classes and related CIL instructions</td>
<td>78</td>
</tr>
<tr>
<td>5.4</td>
<td>Deferred commands of the basic block analysis Visitor and associated CIL instructions</td>
<td>85</td>
</tr>
<tr>
<td>5.5</td>
<td>Effected commands of the first basic block analysis Visitor and introduced basic blocks</td>
<td>87</td>
</tr>
<tr>
<td>5.6</td>
<td>Effected commands of the second basic block analysis Visitor and introduced basic block link targets</td>
<td>87</td>
</tr>
<tr>
<td>5.7</td>
<td>Classifications defined as constants and the corresponding method body items</td>
<td>89</td>
</tr>
<tr>
<td>5.8</td>
<td>Implementation of the code path traversal strategy by the classification statistics extractor</td>
<td>90</td>
</tr>
<tr>
<td>5.9</td>
<td>Implementation of the code path traversal strategy by the code path extractor</td>
<td>91</td>
</tr>
<tr>
<td>5.10</td>
<td>Summary of the implementation of the evaluation stack in the backward symbolic execution Visitor</td>
<td>94</td>
</tr>
<tr>
<td>5.11</td>
<td>Number of code paths finishing at an exception basic block and their classification</td>
<td>96</td>
</tr>
<tr>
<td>5.12</td>
<td>Number of basic blocks on normal and exception code paths finishing at an exception</td>
<td>96</td>
</tr>
<tr>
<td>5.13</td>
<td>Extracted precondition clauses and their number of occurrence in <code>ArrayList</code></td>
<td>97</td>
</tr>
<tr>
<td>5.14</td>
<td>Selection of extracted precondition clauses from <code>ArrayList</code></td>
<td>98</td>
</tr>
</tbody>
</table>
5.15 Extracted precondition clauses and their number of occurrence in Stack .......................................................... 99
5.16 Extracted precondition clauses and their number of occurrence in Queue ......................................................... 99
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Keyword CIL_DOT_CLASS scanner rule</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Simple identifier scanner rule</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Declarations grammar rules</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Declarations grammar rule with actions</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Fragments of the rules for Native_type and Variant_type symbols</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Fragments of the rules for Method_declaration, Identifier and External_sourceSpecification</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Portions of the grammar relevant to the .namespace directive</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic Block</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>Basic Block Content</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>Basic Block Link</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>Basic Block Link Content</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>Basic Blocks Analysis - First Pass (Part 1 of 2)</td>
<td>53</td>
</tr>
<tr>
<td>5.6</td>
<td>Basic Blocks Analysis - First Pass (Part 2 of 2)</td>
<td>54</td>
</tr>
<tr>
<td>5.7</td>
<td>Basic Blocks Analysis - Second Pass (Part 1 of 2)</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>Basic Blocks Analysis - Second Pass (Part 2 of 2)</td>
<td>57</td>
</tr>
<tr>
<td>5.9</td>
<td>Basic Blocks Classification</td>
<td>58</td>
</tr>
<tr>
<td>5.10</td>
<td>Code Path Traversal</td>
<td>60</td>
</tr>
<tr>
<td>5.11</td>
<td>Code Path Extraction</td>
<td>62</td>
</tr>
<tr>
<td>5.12</td>
<td>Symbolic Execution (Part 1 of 4)</td>
<td>64</td>
</tr>
<tr>
<td>5.13</td>
<td>Symbolic Execution (Part 2 of 4)</td>
<td>65</td>
</tr>
<tr>
<td>5.14</td>
<td>Symbolic Execution (Part 3 of 4)</td>
<td>67</td>
</tr>
<tr>
<td>5.15</td>
<td>Symbolic Execution (Part 4 of 4)</td>
<td>68</td>
</tr>
<tr>
<td>5.16</td>
<td>Precondition Extraction (Part 1 of 2)</td>
<td>70</td>
</tr>
<tr>
<td>5.17</td>
<td>Precondition Extraction (Part 2 of 2)</td>
<td>71</td>
</tr>
<tr>
<td>5.18</td>
<td>visit.bge_instruction of CIL_SYMBOLIC_EXECUTION_VISITOR</td>
<td>94</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Although Design by Contract, as supported by Eiffel (see [15] and [14]), has been shown to provide several benefits, it is not widely used yet. The question arises whether there are implicit, so called “closet contracts” hidden in the documentation or code, even if the underlying language and method do not support them explicitly. This has led to the “Closet Contract Conjecture”, described by Karine Arnout and Bertrand Meyer in [1]. The paper also details the results of an examination, inspired by the conjecture, of the .NET libraries, which has shown some patterns about closet contracts and supports their existence.

Design by Contract has been applied as an integral part of software design, mostly in Eiffel. The fact that recently published libraries like the .NET libraries have not adopted this method, despite its virtue, lets one think of its application a posteriori. The source for extracting contracts from given libraries could be source code, comments in the source code, documentation, specification, exceptions, generated code (e.g., Java Bytecode, .NET CIL code) or other available information.

The Closet Contract Conjecture states that contracts are either but an artefact of their support or an inherent part in the design of libraries. Wherever the latter holds, users may enhance their understanding of the concerned libraries and even improve them by eliciting and making explicit the underlying contracts.

The .NET libraries are an interesting target for studying closet contracts due to the flexibility of the .NET component model, which has enabled the development of the “Contract Wizard” [2]. The wizard is a tool that enables its user to examine a compiled module (“assembly” in .NET), interactively add contracts to its classes and routines and produce a contracted proxy assembly that forwards calls to the original implementation.

A tool like the Contract Wizard would not be of interest if no closet contracts existed and continuing work on it would be meaningless. The paper [1] further explores on the extraction of contracts based on documentation and specification of classes and interfaces in the .NET framework and concludes that the observations made support the existence of closet contracts. The authors also observe that routine preconditions tend to be buried under exception
conditions and suggest that the extraction of such preconditions could—at least partially—be done automatically by a corresponding tool.

The goal of the present project is the development of a tool, that automatically infers routine preconditions based on the exceptions a concrete routine of a .NET framework [16] class can throw.

1.2 Overview

The developed tool takes the CIL source code of a .NET library module, as output by ildasm [16] from a binary .NET assembly, as input and extracts routine preconditions based on the routine implementations. The tool proceeds in several stages.

The CIL source code is read by a scanner (see chapter 2 on page 10) which outputs tokens according to rules specified in a scanner description file for the scanner generator gelex [3].

The tokens from the scanner are read by a parser (see chapter 3 on page 13) which creates an abstract syntax tree (see chapter 4 on page 18) as specified in a parser description file for the parser generator geyacc [6].

The contract extraction (see chapter 5 on page 49) analyzes the instruction sequence of each method and infers, based on that analysis, routine preconditions. The contract extraction is arranged into the following steps.

The basic block analysis (see section 5.4 on page 84) partitions the instruction sequence into basic blocks, where the flow of execution within a basic block is sequential, for example branch instructions point only to the start and occur only at the end of a basic block. The result of the basic block analysis is a directed graph (see section 5.2 on page 71) with the basic blocks as vertices and the branch targets, including the implicit sequential targets, as edges.

Each basic block is classified (see section 5.5 on page 88) according to the instructions it contains. The classification scheme allows to detect code paths in a method that are currently not tractable by the implementation of the later steps in the contract extraction. For example a basic block, which calls another method or stores to a field, local variable or parameter is currently classified as intractable—unless it finishes at an explicit exception.

The code path extraction (see section 5.6 on page 91) computes a subgraph of the complete basic block graph where the subgraph does not contain any intractable basic blocks.

The symbolic execution (see section 5.7 on page 91) extracts symbolic expressions (see section 5.3 on page 74) of the branch conditions in the basic block subgraph.

The precondition extraction (see section 5.8 on page 94) associates the precondition false to basic blocks finishing at an exception and from this and the extracted branch conditions incrementally builds preconditions for the preceding basic blocks. The result is a precondition for the first basic block in the method, which is a precondition for the method itself.

An algorithm for computing string representations of expressions (see section 5.3 on page 74) is used by the tool to output the preconditions in an Eiffel-like syntax.

The results of an algorithm for collecting statistics on the classification scheme (see section 5.5 on page 88) show, in the case of ArrayList and its
nested classes (see [16]), that 150 out of 246 code paths finishing at an explicit exception are tractable by the current implementation for contract extraction. The symbolic execution algorithm is able to extract 98 of the 99 branch conditions on these tractable code paths under consideration. The number of the extracted precondition clauses, in this case, is 115, which is less than the number of tractable paths finishing at an exception, because the precondition extraction algorithm employs an optimization to avoid redundant precondition clauses.

Table 1.1: Examples of extracted precondition clauses from ArrayList

<table>
<thead>
<tr>
<th>Precondition Clause</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>not (index &lt; 0)</td>
<td>22</td>
</tr>
<tr>
<td>count &gt;= 0</td>
<td>14</td>
</tr>
<tr>
<td>(this._size - index) &gt;= count</td>
<td>7</td>
</tr>
<tr>
<td>index &lt; this._size</td>
<td>3</td>
</tr>
<tr>
<td>index &lt;= this._size</td>
<td>2</td>
</tr>
<tr>
<td>value &gt;= 0</td>
<td>1</td>
</tr>
<tr>
<td>arrayIndex &gt;= 0</td>
<td>1</td>
</tr>
<tr>
<td>(startIndex + count) &lt;= this._size</td>
<td>1</td>
</tr>
<tr>
<td>not (startIndex &lt; 0)</td>
<td>1</td>
</tr>
<tr>
<td>startIndex &lt; this._size</td>
<td>1</td>
</tr>
<tr>
<td>startIndex &lt;= this._size</td>
<td>1</td>
</tr>
<tr>
<td>this._remaining &gt;= 0</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 2

Scanner

The scanner reads a stream of characters (e.g., from a CIL source file) and outputs a stream of tokens. Each token corresponds to a sequence of characters in the input stream. The stream of tokens is input to the parser. The scanner is implemented in a scanner description file for the scanner generator gelex (see Gobo Eiffel Lex [3]).

The gelex tool generates an Eiffel class CIL_SCANNER from the scanner description file. The scanner description file has three sections: declarations, rules and user code. Each section is described below, for a comprehensive documentation on gelex and the scanner description file in general see [3].

2.1 Declarations

The declarations in the first section of the scanner description file contain the Eiffel class header of the generated class CIL_SCANNER. The options of gelex suppress the generation of a default rule for unmatched characters, enable the tracking of line and column numbers and supply the class file name.

2.2 Rules

Each rule specifies a pattern of input characters and an action. The pattern is given as a regular expression that is matched against the current characters in the input stream. The action is denoted as an Eiffel code snippet. At runtime the scanner matches the patterns against the current input characters and selects the rule matching the most characters. If there is more than one rule with a longest match, the first of these rules in the description file is selected. The matched characters are then removed from the input stream and the action of the matching rule is executed. A commonly used action is to attach last_token to a token corresponding to the matched characters. The following documents the different logical parts the rules section contains.

Ignored Input

Whitespace (i.e., spaces, tabs and new lines) and comments (i.e., starting with “//” and finishing at the end of the current line) are ignored.
Keywords

Most rules match against a CIL keyword and attach `last_token` to a token corresponding to the keyword. For a list of keywords see also [11, section C.1.].

```
".class"
{
    last_token := CIL_DOT_CLASS
}
```

Listing 2.1: Keyword CIL_DOT_CLASS scanner rule

When the scanner attempts to match the rules against the input characters, an input of “.class” will cause the associated action to be executed (this rule provides the only longest match in this case). The action assigns `last_token` to the CIL_DOT_CLASS token.

Values

The rules for values are split into the following logical groups and rules:

- Hexadecimal 8-bit integer numbers
  - a letter followed by a letter or a digit
  - two digits
  - a digit followed by a letter
- 64-bit integer numbers
  - signed decimal integers
  - unsigned hexadecimal integers
- Floating point numbers
  - only one rule
- Identifiers and strings
  - simple identifiers
  - singly quoted strings
  - doubly quoted strings

The reason for distinguishing between the three cases of hexadecimal integers is the ambiguities of a letter followed by a letter or a digit, which represent either a hexadecimal integer or a simple identifier, and two digits, which denote either a hexadecimal integer or a decimal integer. The information on the input CIL code available to the scanner (basically only the next characters in the input stream) is not sufficient to resolve the ambiguities. The parser has to determine which kind of value is permitted at this point in the token stream (see section 3 on page 13).

An identifier can be a simple identifier or a singly quoted string, see also section 3 on page 13 and [8, section 5.2].
Listing 2.2: Simple identifier scanner rule

A simple identifier starts with an alphabetic character, "_", "$", "@" or "?" and continues with any number of alphanumeric characters, "_", "$", "@" or "?" (see [8, section 5.2]). If this rule matches the current input characters, the associated action assigns last_token to the CIL_IDENTIFIER token and attaches last_string_value to the matched input string. The semantic value of the CIL_IDENTIFIER token is declared to be of type STRING in the parser description file (see section 3.1 on the next page).

Special

There are two special rules at the end of the scanner description. The first specifies that an end-of-file terminates the scanning process. The second matches any single character and attaches last_token to the corresponding character code. The parser generator geyacc (see [6] and section 3 on the following page) supports this scanner rule by allowing the use of single character tokens where the token in the parser description is the corresponding character enclosed in single quotes.

2.3 User Code

The user code section is copied to the end of the generated class file. The only content is the end-of-class keyword end.
Chapter 3

Parser

The parser reads a stream of tokens (i.e., from the scanner, see section 2 on page 10) and creates an abstract syntax tree (section 4 on page 18) with the factory (section 4.12 on page 46) attached to one of its attributes. The parser is implemented as a parser description file for the parser generator *geyacc* (see *Gobo Eiffel Yacc* [6]).

The *geyacc* tool generates a parser CIL_PARSER and a token CIL_TOKENS Eiffel class from the parser description file. The parser description file is split into three sections: declarations, rules and user code, described below. For a comprehensive documentation on *geyacc* and the parser description file in general see [6].

3.1 Declarations

The declarations section is split into four parts, each of them described in the following sections.

Class Header

This part of the declarations section is the class header of the generated parser class CIL_PARSER. The parser class inherits its basic features from CIL_PARSER_SKELETON and the complete scanner from CIL_SCANNER (see section 2 on page 10).

Tokens

The second part of the declarations section declares the tokens (also called terminal symbols of the grammar) that do not correspond to a single character and therefore have to be explicitly declared. It is from these token declarations that *geyacc* generates the token Eiffel class CIL_TOKENS. The first eight token declarations (i.e., the tokens CIL_IDENTIFIER, CIL_DOUBLY_QUOTED_STRING, CIL_SINGLY_QUOTED_STRING, CIL_INT32, CIL_INT64, CIL_INT64_HEX, CIL_FLOAT64, CIL_HEX_BYTE1, CIL_HEX_BYTE2, CIL_HEX_BYTE3) stand out as they also declare a type, STRING in this case, for the semantic values of the tokens. All other tokens correspond to keywords and do not have a semantic value.
Nonterminal Symbols

While nonterminal symbols do not have to be declared, it is possible to declare a type for each. This is done in the third part of the declaration section. Having types of nonterminal symbols declared explicitly does not only improve type-safety, but also simplifies the task of writing Eiffel source code that relies on features of a particular class in the actions described in section 3.2.

Options

The options part declares the start symbol Module and the number of expected shift/reduce conflicts (see section 3.4 on the following page).

3.2 Rules

The rules section describes the syntax of the language (i.e., CIL) with grammar rules. Each rule specifies a sequence of terminal and nonterminal symbols (the components) for a particular nonterminal symbol (the result). The parser can reduce such a sequence of symbols to the nonterminal symbol. Several rules for the same resulting nonterminal symbol can be combined with each other. The grammar rules of this section are derived from [11, section C.3].

Declarations: -- Empty
  | Declarations Declaration

Listing 3.1: Declarations grammar rules

Declarations with a semantic value of type LIST [CIL_ENTITY] (see section 3.1) are either empty (first rule) or, recursively, Declarations followed by a Declaration (second rule). The second rule is called left recursive because the recursion appears on the left hand side. Left recursive rules allow the parser algorithm to parse the corresponding input with bounded space (see [6, Recursive Rules]).

While a grammar rule specifies the syntax, actions specify the semantics of a nonterminal symbol. A semantic value of a particular Eiffel type is associated with each nonterminal and a few terminal symbols (see section 3.1 on the preceding page). The semantic values of the components of a rule are used for computing the semantic value of the corresponding result. Actions are specified in Eiffel code snippets.

NOTE 3.1 Preliminary runs on the complete disassembled mscorlib.dll of the .NET framework indicate that some of the keywords can also be used as identifiers. However, the rule for identifiers Identifier does not take this into account. The Identifier rule needs to be extended to handle “some” of the keyword tokens when an identifier token is expected.
3.3 User Code

The user code includes various attributes that are used as placeholders of semantic values until the parser is able to set the actual semantic value. These attributes are used only in the actions of the rules corresponding to CIL instructions. The grammar rules could be rearranged in order to avoid these placeholders, but this would lead to further deviation from the original grammar [11, section C.3].

3.4 Shift/Reduce Conflicts

There remain four unresolved shift/reduce conflicts in the grammar. Details of the conflicts are available through the --verbose option of geypacc. [6, Parser Algorithm] includes a description of the different kinds of conflicts. The following is a short description of the sources of the remaining shift/reduce conflicts.

Native_type and Variant_type

At positions (1) on listing 3.3 on the next page the parser cannot decide if a ‘[’ should result in a reduce (continuing at one of the ‘[’ of the Native_type rules) or in a shift.

Suppose, for example, the parser reads the following sequence of tokens at a point where a Native_type symbol can occur: CIL_SAFE_ARRAY, CIL_NULL, ‘[‘. When the parser reads the ‘[‘ token, it is not clear if it should shift in the Variant_type: Variant_type ‘[‘ ‘]’ rule or, without using this rule, reduce with the Variant_type: CIL_NULL rule and then shift in the Native_type: Native_type ‘[‘ [...] rules. A similar problem as with ‘[‘ exists with ‘*’ at the positions indicated by (2) on listing 3.3 on the following page.

The default action of geypacc is to shift (see [6, Parser Algorithm]), which is the desired action in these two cases (the grammar rules Variant_type: ‘*’ and Variant_type: Variant_type ‘[‘ ‘]’ would be unused otherwise).
Listing 3.3: Fragments of the rules for `Native_type` and `Variant_type` symbols

**External_source_specification and Identifier**

If the parser is at positions (3) or (4) on listing 3.4 on the next page, it cannot make the decision if a `Singly_quoted_string` results in a *shift* or a *reduce*. To *reduce* would mean to start a label (*Method_declaration*: `Identifier`: `:`). It is necessary to look two tokens ahead in order to see if there is a colon (indicating a label) or not, but it is not clear if this is sufficient.

Suppose, for example, the parser reads the following sequence of tokens at a point where a `Method_declaration` can occur: `CIL_DOT_LINE, Integer, Singly_quoted_string`. When the parser reads the `Singly_quoted_string`, it is not clear if it should *shift* in the first or *reduce* with the second rule of `External_source_specification` and then *shift* in the label rule from the `Method_declaration`.

This is a defect. Because *ildasm* [16] does not use singly quoted label identifiers no further effort apart from documenting has been made to resolve it.
Method_declaration: [...]
  | Identifier ':'
  | External_source_specification
[...]

Identifier: [...]  
  | Singly_quoted_string
          
;  

External_source_specification:  
  CIL_DOT_LINE Integer (3) Singly_quoted_string  
    | CIL_DOT_LINE Integer (3)  
    | CIL_DOT_LINE Integer ':'  
          Integer (4) Singly_quoted_string  
    | CIL_DOT_LINE Integer ':' Integer (4)
[...]

Listing 3.4: Fragments of the rules for Method_declaration, Identifier and External_source_specification
Chapter 4

Abstract Syntax Tree

The parser (section 3 on page 13) creates an abstract syntax tree (AST) of the CIL source code by using the AST factory (section 4.12 on page 46). This chapter documents the Eiffel classes that composed the AST, including classes tightly coupled to the AST, like the factory or the Visitors (section 4.13 on page 47). The classes are found in the ast cluster of the delivery. See also [8], [9] and [11, section C.3. and C.4.].

Note 4.1 The current implementation of the “AST” still resembles a parse tree. For example, the module as implemented in the class CIL_MODULE (see section 4.1) has a list of entities but no queries to retrieve specific entities like types, custom_attributes, manifest, ...

Annotation of the AST (see section 4.14 on page 48) provides one possibility to resolve this limitation. For example, adding attributes classes, manifest, custom_attributes, ... and is_annotated of type BOOLEAN to CIL_MODULE, where the annotation Visitor attaches the former attributes to their corresponding lists (in the case of types and custom_attributes) and instances of helper classes (in the case of manifest) and set is_annotated to true.

When examining this issue from the parser’s (see 3 on page 13) perspective, suppose, for example, the parser should be modified to extend a list of classes on the module each time it encounters a type definition in the Declarations rules of the grammar. The problem with this approach is that at the time when the action of the custom attributes rule of Declarations is executed, the instance of CIL_MODULE is not created yet—the parser creates the “AST” in bottom-up manner. Solving this issue by modifying the parser is not straightforward.

4.1 Module

The unit of deployment in .NET is the assembly. An assembly is a set of files, one or more of these files are modules and zero or more of them are additional resource files. A module is a file containing metadata and executable content (see [8, section 6]). Exactly one module of an assembly contains an assembly manifest (see section 4.2 on the following page).
The root node of the AST represents a module, implemented in the class CIL_MODULE. A module is composed of a sequence of entities (see [8, section 5.10], whereby entities are called declarations). CIL_MODULE has an attribute entities of type LIST [CIL_ENTITY] and a corresponding setter set_entities, a creation routine make taking a list of entities as argument and a command extend for extending the list of entities.

Note 4.2 See note 4.1 on the page before.

4.2 Entities

A module comprises a list of entities. Entities are represented by classes inheriting from CIL_ENTITY. This section documents entities inheriting neither from CIL_TYPE_MEMBER nor from CIL_METHOD_BODY_ITEM (see sections 4.4 on page 23 and 4.6 on page 37). See also [8, section 5.10].

Assembly Manifest

The assembly manifest is part of exactly one module of an assembly. It declares the resources of the assembly and specifies additional information about the assembly. See also [8, section 6.2].

Note 4.3 The assembly manifest is currently not supported by AST classes.

Image Base

The .imagebase directive is supported by the grammar in [11, section C.3.] but not documented in the standard (see [8, section 5.10]).

Note 4.4 The image base entity is currently not implemented in the AST classes.

Module Declaration

The .module directive declares the module’s file name (see [8, section 6.4]).

Note 4.5 The module entity is currently not implemented in the AST classes.

Namespaces

The .namespace directive is supported by the grammar in [11, section C.3.] but not documented in the standard (see [8, section 5.10]). (The last paragraph in [8, section 6] explicitly states that namespaces are not supported. ildasm [16] however makes use of them.) The relevant portions of the grammar suggest that the .namespace header defines a prefix for all names of the entities the namespace contains.

Virtual Table Declaration

The .vtable directive is supported by the grammar in [11, section C.3.] but not documented in the standard (see [8, section 5.10]).

Note 4.6 The virtual table entity is currently not implemented in the AST classes.
A namespace has a Namespace_header and a list of Declarations where the header has an identifier and the grammar rules for the Declarations are the same rules as used by the AST root rule Module.

Virtual Table Fixup Declaration

The .vtfixup directive supports calls from unmanaged into managed code. The virtual fixup declaration specifies the memory location of a table of method tokens. The tokens are converted to method pointers when the module is loaded into memory for execution (see [8, section 14.5.1]). This directive is implemented in the class CIL_VIRTUAL_FIXUP_DECLARATION where the attribute table_label of type CIL_IDENTIFIER refers to the memory location and table_count of type INTEGER declares the number of affected entries in the table. The feature attributes of type LIST [CIL_VIRTUAL_FIXUP_ATTRIBUTE] holds a list of virtual table fixup attributes.

Note 4.7 The effective descendant of CIL_VIRTUAL_FIXUP_ATTRIBUTE, CIL_GENERAL_VIRTUAL_FIXUP_ATTRIBUTE, inherits its implementation from CIL_GENERAL_ATTRIBUTE (see section 4.8 on page 41).

Note 4.8 The virtual table fixup declaration and its attributes are currently not supported by the factory (documented in section 4.12 on page 46) and the parser (documented in section 3 on page 13). The implementation is therefore not in use.

4.3 Type Members

A type (see 4.4 on page 23) comprises—among other attributes—a list of type members. Type members are represented by classes inheriting from the deferred class CIL_TYPE_MEMBER. This section documents type members inheriting neither from CIL_ENTITY nor from CIL_METHOD_BODY_ITEM (see sections 4.4 on page 23 and 4.6 on page 37). See also [8, section 9.2].

Events

An event of type CIL_EVENT has a header of type CIL_EVENT_HEADER and a list of members of type LIST [CIL_EVENT_MEMBER]. See also [8, section 17].

Note 4.9 The event declaration is currently not supported by the factory (documented in section 4.12 on page 46) and the parser (documented in section 3 on page 13). The implementation is therefore not in use.
Event Header

The event header is implemented in CIL\_EVENT\_HEADER and specifies an identifier of type CIL\_IDENTIFIER and an optional delegate of type CIL\_TYPE\_SPECIFICATION for the event. The status report attribute is\_special\_name marks the event name for tools, is\_rt\_special\_name marks the event for the runtime environment. See also [8, section 17].

**Note 4.10** See 4.9 on the preceding page, which also applies here.

Event Members

The deferred class CIL\_EVENT\_MEMBER is ancestor to all event members (e.g., CIL\_EXTERNAL\_SOURCE\_LINE documented in section 4.6 on page 38 and CIL\_CUSTOM\_ATTRIBUTE documented in section 4.6 on page 37).

**Note 4.11** In the current implementation CIL\_EXTERNAL\_SOURCE\_LINE and CIL\_CUSTOM\_ATTRIBUTE are the only descendant classes of CIL\_EVENT\_MEMBER. The other event members, introduced by the keywords .addon, .fire, .other and .removeon, are not implemented.

**Note 4.12** See 4.9 on the page before, which also applies here.

Properties

Properties are implemented by the class CIL\_PROPERTY which comprises the attributes: header of type CIL\_PROPERTY\_HEADER and a list of members of type LIST [CIL\_PROPERTY\_MEMBER]. See also [8, section 16].

**Note 4.13** The property declaration is currently not supported by the factory (documented in section 4.12 on page 46) and the parser (documented in section 3 on page 13). The implementation is therefore not in use.

Property Header

The property header is implemented in CIL\_PROPERTY\_HEADER and specifies an identifier of type CIL\_IDENTIFIER and a signature of type CIL\_METHOD\_SIGNATURE for the property. The attribute is\_special\_name of type BOOLEAN marks the property name for tools, is\_rt\_special\_name marks the property for the runtime environment. See also [8, section 16].

**Note 4.14** See 4.13, which also applies here.

Property Members

The deferred class CIL\_PROPERTY\_MEMBER is ancestor to all property members (e.g., CIL\_EXTERNAL\_SOURCE\_LINE, which is documented in section 4.6 on page 38 and CIL\_CUSTOM\_ATTRIBUTE, which is documented in section 4.6 on page 37).

**Note 4.15** In the current implementation CIL\_EXTERNAL\_SOURCE\_LINE and CIL\_CUSTOM\_ATTRIBUTE are the only descendants of class CIL\_PROPERTY\_MEMBER. The other property members, introduced by the keywords .get, .other and .set, are not implemented.
Note 4.16 See 4.13 on the page before, which also applies here.

Explicit Memory Alignment of Fields
The .pack directive specifies the alignment of the enclosing type’s fields to be at multiples of the declared value in bytes or the field’s type natural alignment, whichever is less (see [8, section 9.7]).

Note 4.17 The .pack directive for explicit memory alignment of fields is currently not implemented in the AST classes.

Explicit Memory Requirement of Instances
The .size directive specifies in bytes the amount of memory allocated for the type’s instances. The specified size has to be greater than or equal to the calculated size of the type’s instances (see [8, section 9.7]).

Note 4.18 The .size directive for explicitly specifying memory requirement of instances is currently not implemented in the AST classes.

Export Declaration
The .export directive is supported by the grammar in [11, section C.3.] but not documented in the standard (see [8, section 9.2]). See also [8, section 6.7] for a seemingly—when looking at the grammar—related entity.

Note 4.19 The export declaration is currently not implemented in the AST classes.

Overriding Methods
The .override directive specifies a method from the current type that should override another method (see [8, section 9.3.2]). This member declaration is implemented by the class CIL_OVERRIDE_MEMBER implements this type member, it inherits from CIL_OVERRIDE (see section 4.8 on page 42) the attributes type and method name and the corresponding setters. The class CIL_OVERRIDE_MEMBER also has to specify the overriding method (unlike the second descendant of CIL_OVERRIDE, CIL_OVERRIDE_BODY_ITEM, for which this information is given by the context of the overriding method).

The overriding method is given by the attributes overriding type of type CIL_TYPE_SPECIFICATION, overriding method name of type CIL_NAME and signature of type CIL_METHOD_SIGNATURE, where signature also completes the specification of the overridden method in case of overloading. CIL_OVERRIDE_MEMBER also implements setters for the latter three attributes and provides a corresponding creation routine for initialization of the attributes.

The class CIL_OVERRIDE_MEMBER also effects the accept command of the type member Visitor pattern (see section 4.13 on page 47).
4.4 Entities and Type Members

This section documents the elements of the AST that occur as entities (see also 4.2 on page 19) and type members (see also 4.3 on page 20) but not as method body items (see 4.6 on page 37).

Type Definition

Type definitions are implemented in the class CIL_TYPE_DEFINITION, which comprises a header of type CIL_TYPE_HEADER and a list of members of type LIST[CIL_TYPE_MEMBER]. See also [8, section 9].

The class CIL_TYPE_DEFINITION also effects the accept commands of the entity Visitor pattern inherited from CIL_ENTITY (which is renamed as accept_entity_visitor, see section 4.13 on page 47) and the type member Visitor pattern inherited from CIL_TYPE_MEMBER (which is renamed as accept_type_member_visitor, see section 4.13 on page 47).

Type Header

The type header is implemented in the class CIL_TYPE_HEADER and has the attributes: attributes of type LIST[CIL_TYPE_ATTRIBUTE], identifier of type CIL_IDENTIFIER, base_type of type CIL_TYPE_REFERENCE and interfaces of type LIST[CIL_TYPE_REFERENCE]. These are accompanied by corresponding setters and two creation routines, make without and make_with_base_type with a base_type argument. See also [8, section 9.1].

Note 4.20 CIL_GENERAL_TYPE_ATTRIBUTE is currently the only effective descendant of CIL_TYPE_ATTRIBUTE and inherits its implementation from CIL_GENERAL_ATTRIBUTE (see section 4.8 on page 41).

Type Members

The deferred class CIL_TYPE_MEMBER is ancestor to all type members documented in sections 4.3 on page 20, 4.4 and 4.6 on page 37. See also [8, section 9.2].

Method Definition

A method definition of type CIL_METHOD_DEFINITION comprises a header of type CIL_METHOD_HEADER and a list of body_items (inherited from the class CIL_METHOD_BODY_ITEM_CONTEXT) of type LIST[CIL_METHOD_BODY_ITEM]. See also [8, section 14].

The annotation (see section 4.14 on page 48) is supported by the attributes label_table of type HASH_TABLE[INTEGER, CIL_IDENTIFIER] mapping from code label identifiers to indexes in the list of body_items and is.annotated of type BOOLEAN.

The four attributes are accompanied by corresponding setters set_header, set_body_items, set_label_table and set_is_annotated.

The class CIL_METHOD_DEFINITION also effects the accept commands of the entity Visitor pattern inherited from CIL_ENTITY (which is renamed as accept_entity_visitor, see section 4.13 on page 47) and the type member
Visitor pattern inherited from CIL\_TYPE\_MEMBER (which is renamed as
accept\_type\_member\_visitor, see section 4.13 on page 47).

The class CIL\_METHOD\_DEFINITION is the direct ancestor of
CIL\_CONSTRUCTOR\_DEFINITION and CIL\_INITIALIZER\_DEFINITION,
both of these descendants only add a class invariant clause that puts a restriction
on the method\_name.

Method Header

The method header implemented in CIL\_METHOD\_HEADER comprises the
attributes signature of type CIL\_METHOD\_SIGNATURE, method\_attributes of
type LIST [CIL\_METHOD\_ATTRIBUTE], return\_type\_attributes of type
LIST [CIL\_PARAMETER\_ATTRIBUTE], native\_return\_type of type
CIL\_NATIVE\_TYPE, parameters of type LIST [CIL\_PARAMETER] and a
list of implementation\_attributes of type
LIST [CIL\_IMPLEMENTATION\_ATTRIBUTE]. These are accompa-
nied by corresponding setters and two creation routines, make and
make\_with\_signature, where make\_with\_signature receives an argument of
type CIL\_METHOD\_SIGNATURE instead of—like make—receiving all at-
tributes of CIL\_METHOD\_SIGNATURE separately. There are the following
queries for conveniently accessing the attributes in signature: calling\_convention
of type CIL\_CALLING\_CONVENTION, return\_type of type CIL\_TYPE and
parameters of LIST [CIL\_PARAMETER]. native\_return\_type is an optional
attribute, where the query is\_native\_return\_type\_set reports on the status. See
also [8, section 14.4].

NOTE 4.21 The only effective descendant of CIL\_METHOD\_ATTRIBUTE,
CIL\_GENERAL\_METHOD\_ATTRIBUTE, inherits its implementation from
CIL\_GENERAL\_ATTRIBUTE (see section 4.8 on page 41).
The same applies to CIL\_PARAMETER\_ATTRIBUTE and
CIL\_IMPLEMENTATION\_ATTRIBUTE.

NOTE 4.22 See 4.39 on page 42 on CIL\_NATIVE\_TYPE.

Method Body Items

The deferred class CIL\_METHOD\_BODY\_ITEM is ancestor to all type members
documented in sections 4.5 on the following page and 4.6 on page 37. See
also [8, section 14.4.1].

Field Definition

A field definition of type CIL\_FIELD\_DEFINITION comprises an optional
byte\_offset of type INTEGER with an accompanying status attribute
is\_byte\_offset\_set, attributes of type LIST [CIL\_FIELD\_ATTRIBUTE], a type of
type CIL\_TYPE, an identifier of type CIL\_IDENTIFIER, an optional field\_init
of type CIL\_FIELD\_INIT with a corresponding status query is\_field\_init\_set
and an optional data\_label of type CIL\_IDENTIFIER with the status query
is\_data\_label\_set. Only one of field\_init and data\_label can be set at the same
time. See also [8, section 15].
The attributes are accompanied by corresponding setters `set_byte_offset`, `set_is_byte_offset_set`, `set_type`, `set_identifier`, `set_field_init` and `set_data_label`.

The class CIL\_FIELD\_DEFINITION also effects the `accept` commands of the entity Visitor pattern inherited from CIL\_ENTITY (which is renamed as `accept_entity_visitor`, see section 4.13 on page 47) and the type member Visitor pattern inherited from CIL\_TYPE\_MEMBER (which is renamed as `accept_type_member_visitor`, see section 4.13 on page 47).

Note 4.23 The only effective descendant of CIL\_FIELD\_ATTRIBUTE, CIL\_GENERAL\_FIELD\_ATTRIBUTE, inherits its implementation from CIL\_GENERAL\_ATTRIBUTE (see section 4.8 on page 41).

Note 4.24 The field definition is currently not supported by the AST factory (see 4.12 on page 46) and by the parser (see 3 on page 13). It is therefore not in use.

### 4.5 Method Body Items

A method’s body comprises a list of method body items. Method body items are represented by classes inheriting from CIL\_METHOD\_BODY\_ITEM. See section 4.6 on page 37 for body items also inheriting from CIL\_ENTITY and CIL\_TYPE\_MEMBER. See also [8, section 14.4.1] on the method body and [9] and [11, section C.4.] on the instruction set and syntax.

#### Labels

A code label has an identifier and is followed by an instruction of which it represents the address (see [8, section 5.4]). Code labels are implemented by the class CIL\_LABEL, which has an attribute `identifier` of type CIL\_IDENTIFIER (see section 4.11 on page 46) and a corresponding setter `set_identifier` and creation routine `make`.

CIL\_LABEL also effects the `accept` command of the method body item Visitor pattern (see section 4.13 on page 47).

#### Emitting Bytes

The `.emitbyte` directive (see [8, section 14.4.1.1]) emits an unsigned 8-bit integer at its position into the binary CIL stream. This functionality is provided for testing purposes. The class CIL\_EMIT\_BYTE implements a corresponding method body item and provides access to its attribute `value` of type INTEGER\_8 by the creation routine `make` and the corresponding setter `set_value`.

CIL\_EMIT\_BYTE also effects the `accept` command of the method body item Visitor pattern (see section 4.13 on page 47).

#### Entry Point

Every executable module has to define a method as its entry point of execution. The class CIL\_ENTRY\_POINT represents the corresponding method body item. CIL\_ENTRY\_POINT does not change or add any features apart from effecting the `accept` command of the method body item Visitor pattern (see section 4.13 on page 47).
NOTE 4.25 An attribute `is_entry_point` on methods would be more convenient than a separate method body item class.

**Exception blocks**

An exception block (see [8, section 18]) starts with a protected (.try directive) block and is followed by one or more handlers (catch, fault, filter and finally keywords). The class CIL_EXCEPTION_BLOCK implements this. It has the attributes `try_block` of type CIL_TRY_BLOCK and `handler_clauses` of type LIST [CIL_HANDLER_CLAUSE].

NOTE 4.26 The lack of support for exception handling by the AST is one of the important limitations of the AST regarding the analysis for precondition extraction.

**Export Directive**

The `.export` directive is supported by the grammar in [11, section C.3.] but not documented in the standard (see [8, section 9.2]).

NOTE 4.27 The export type member is currently not implemented in the AST classes.

**Local variables declaration**

Local variables are declared by the `.locals` directive (see [8, section 14.4.1.3]) in the method body. The class CIL_LOCALS implements a corresponding item by providing the attributes `variables` of type LIST [CIL_LOCAL_VARIABLE] (see section 4.9 on page 43) and `is_initialized` of type BOOLEAN. `is_initialized` indicates if the variables should be initialized to their default values (null for reference types, zero for value types) by the runtime environment.

CIL_LOCALS also provides the necessary creation routines, setters for the attributes and a command `extend` for conveniently extending the list of local variable declarations. It also effects the `accept` command of the method body item Visitor pattern (see section 4.13 on page 47).

NOTE 4.28 An attribute `locals` on methods would be more convenient than a separate method body item class. (`ilasm` [16] and the grammar accept several locals for one method, `ilasm`, however, appears to merge them into one, if one of them requires the variables to be initialized, then the merged locals declaration will also require an initialization.)

**Instructions**

All instruction classes are descendants of the deferred class CIL_INSTRUCTION. There are roughly one hundred effective instruction classes in the cluster `ast/method_body/instruction` of the delivery. They are organized into three sub-clusters: `base`, `object_model` and `prefix`, reflecting the partitioning of the instructions into these three groups in [9].

26
A second partitioning of the effective instruction classes is given by the inheritance hierarchy, where the effective instruction classes inherit all attributes and routines they are made of and only implement the visit command for the method body item Visitor pattern (see section 4.13 on page 47). The following subsections make use of this second partitioning by documenting the direct ancestors of the effective instruction classes.

**CIL_DATA_INSTRUCTION**

The effective descendants of CIL_DATA_INSTRUCTION represent instructions having some associated data.

CIL_DATA_INSTRUCTION directly inherits from CIL_INSTRUCTION and is itself a direct ancestor of the deferred classes CIL_NUMERIC_INSTRUCTION and CIL_STRING_INSTRUCTION. CIL_DATA_INSTRUCTION defines an attribute data of type ANY and a corresponding creation routine make and setter set_data for initializing and setting the attribute. Its (direct and indirect) deferred descendants redefine the attribute data to be of the type corresponding to them. CIL_STRING_INSTRUCTION has no deferred descendant, CIL_NUMERIC_INSTRUCTION has four deferred descendants.

<table>
<thead>
<tr>
<th>Class</th>
<th>Type of data</th>
<th>Ancestora</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_NUMERIC_INSTRUCTION</td>
<td>NUMERIC</td>
<td>DATA b</td>
</tr>
<tr>
<td>CIL_FLOAT_INSTRUCTION</td>
<td>REAL_REF</td>
<td>NUMERICc</td>
</tr>
<tr>
<td>CIL_FLOAT_64_INSTRUCTION</td>
<td>DOUBLE_REF</td>
<td>NUMERICc</td>
</tr>
<tr>
<td>CIL_INTEGER_INSTRUCTION</td>
<td>INTEGER_REF</td>
<td>NUMERICc</td>
</tr>
<tr>
<td>CIL_INTEGER_64_INSTRUCTION</td>
<td>INTEGER_64_REF</td>
<td>NUMERICc</td>
</tr>
<tr>
<td>CIL_STRING_INSTRUCTION</td>
<td>STRING</td>
<td>DATA b</td>
</tr>
</tbody>
</table>

a Abbreviated
b Abbreviation for CIL_DATA_INSTRUCTION
c Abbreviation for CIL_NUMERIC_INSTRUCTION

Table 4.2: Effective descendants of CIL_DATA_INSTRUCTION

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Sectiona</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_LDC_I4_INSTRUCTION</td>
<td>Load 32-bit integer constant</td>
<td>3.40</td>
</tr>
<tr>
<td>CIL_LDC_I8_INSTRUCTION</td>
<td>Load 64-bit integer constant</td>
<td>3.40</td>
</tr>
<tr>
<td>CIL_LDC_R4_INSTRUCTION</td>
<td>Load 32-bit floating-point constant</td>
<td>3.40</td>
</tr>
<tr>
<td>CIL_LDC_R8_INSTRUCTION</td>
<td>Load 64-bit floating-point constant</td>
<td>3.40</td>
</tr>
<tr>
<td>CIL_LDSTR_INSTRUCTION</td>
<td>Load string constant</td>
<td>4.15</td>
</tr>
<tr>
<td>CIL_UNALIGNED_INSTRUCTION</td>
<td>Unaligned pointer instruction</td>
<td>2.2</td>
</tr>
</tbody>
</table>

a Section of [9]

**Note 4.29** CIL_UNALIGNED_INSTRUCTION represents a prefix instruction, it would be convenient to have a boolean query is_unaligned for the instructions
that can follow the unaligned prefix instead of a separate class for the prefix itself.

**CIL_LABEL_LIST_INSTRUCTION**

CIL_LABEL_LIST_INSTRUCTION directly inherits from CIL_INSTRUCTION and is the deferred ancestor of all effective instruction classes having a list of labels. It implements a creation routine `make`, an attribute `references` and a corresponding setter `set_references`.

The only descendant of CIL_SIGNATURE_INSTRUCTION, CIL_SWITCH_INSTRUCTION (see [9, section 3.66]), effects the command `visit` of the method body item Visitor pattern (see section 4.13 on page 47) and does not add or change any other feature.

**CIL_LOCAL_REFERENCE_INSTRUCTION**

CIL_LOCAL_REFERENCE_INSTRUCTION directly inherits from the class CIL_INSTRUCTION and is itself a direct ancestor of the deferred classes CIL_BRANCH_INSTRUCTION and CIL_VARIABLE_INSTRUCTION. It defines an attribute `reference` of type CIL_LOCAL_REFERENCE and a corresponding creation routine `make` and setter `set_reference` for initializing and setting the attribute. Its descendants covariantly redefine the attribute’s type.

The effective descendants of CIL_LOCAL_REFERENCE_INSTRUCTION represent instructions having a reference to a label in the method body (see section 4.5), a local variable (see section 4.5 on the next page) or a parameter variable (see section 4.5 on page 30).

**CIL_BRANCH_INSTRUCTION**

CIL_BRANCH_INSTRUCTION is the deferred ancestor of all effective branch instruction classes, CIL_LOCAL_REFERENCE_INSTRUCTION is its direct ancestor of which it redefines the attribute `reference` of type CIL_LOCAL_REFERENCE to type CIL_LABEL_REFERENCE.

The descendants of CIL_BRANCH_INSTRUCTION effect the command `visit` of the method body item Visitor pattern (see section 4.13 on page 47) and do not add or change any other feature.
Table 4.3: Descendants of CIL\_BRANCH\_INSTRUCTION

<table>
<thead>
<tr>
<th>Class</th>
<th>Branch Condition</th>
<th>Section^a</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_BEQ_INSTRUCTION</td>
<td>if equal</td>
<td>3.5</td>
</tr>
<tr>
<td>CIL_BGE_INSTRUCTION</td>
<td>if greater or equal (signed)</td>
<td>3.6</td>
</tr>
<tr>
<td>CIL_BGE_UN_INSTRUCTION</td>
<td>if greater or equal (unsigned/unordered)</td>
<td>3.7</td>
</tr>
<tr>
<td>CIL_BGT_INSTRUCTION</td>
<td>if greater (signed)</td>
<td>3.8</td>
</tr>
<tr>
<td>CIL_BGT_UN_INSTRUCTION</td>
<td>if greater (unsigned/unordered)</td>
<td>3.9</td>
</tr>
<tr>
<td>CIL_BLE_INSTRUCTION</td>
<td>if less or equal (signed)</td>
<td>3.10</td>
</tr>
<tr>
<td>CIL_BLE_UN_INSTRUCTION</td>
<td>if less, equal (unsigned/unordered)</td>
<td>3.11</td>
</tr>
<tr>
<td>CIL_BLT_INSTRUCTION</td>
<td>if less (signed)</td>
<td>3.12</td>
</tr>
<tr>
<td>CIL_BLT_UN_INSTRUCTION</td>
<td>if less (unsigned/unordered)</td>
<td>3.13</td>
</tr>
<tr>
<td>CIL_BNE_UN_INSTRUCTION</td>
<td>if unequal or unordered</td>
<td>3.14</td>
</tr>
<tr>
<td>CIL_BR_INSTRUCTION</td>
<td>unconditional</td>
<td>3.15</td>
</tr>
<tr>
<td>CIL_BRFALSE_INSTRUCTION</td>
<td>if false</td>
<td>3.17</td>
</tr>
<tr>
<td>CIL_BRTRUE_INSTRUCTION</td>
<td>if true</td>
<td>3.18</td>
</tr>
<tr>
<td>CIL_LEAVE_INSTRUCTION</td>
<td>unconditional^b</td>
<td>3.46</td>
</tr>
</tbody>
</table>

^a Section of [9]
^b Leaves a protected region of code

**CIL\_VARIABLE\_INSTRUCTION**

The class CIL\_VARIABLE\_INSTRUCTION is the deferred ancestor of CIL\_LOCAL\_VARIABLE\_INSTRUCTION (see section 4.5) and CIL\_PARAMETER\_VARIABLE\_INSTRUCTION (see section 4.5 on the next page). The class CIL\_LOCAL\_REFERENCE\_INSTRUCTION is its direct ancestor of which it redefines the attribute reference of type CIL\_LOCAL\_REFERENCE to type CIL\_VARIABLE\_REFERENCE.

The direct descendants of classes CIL\_VARIABLE\_INSTRUCTION, CIL\_LOCAL\_VARIABLE\_INSTRUCTION and CIL\_PARAMETER\_VARIABLE\_INSTRUCTION, redefine the type of the attribute reference and leave the other inherited features unchanged.

**CIL\_LOCAL\_VARIABLE\_INSTRUCTION**

The class CIL\_LOCAL\_VARIABLE\_INSTRUCTION is the deferred ancestor of all effective instruction classes referring to a local variable, CIL\_VARIABLE\_INSTRUCTION is its direct ancestor of which it redefines the attribute reference of type CIL\_VARIABLE\_REFERENCE to type CIL\_LOCAL\_VARIABLE\_REFERENCE.

The descendants of CIL\_LOCAL\_VARIABLE\_INSTRUCTION effect the command visit of the method body item Visitor pattern (see section 4.13 on page 47) and do not add or change any other feature.
Table 4.4: Descendants of CIL\_LOCAL\_VARIABLE\_INSTRUCTION

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_LDLOC_INSTRUCTION</td>
<td>Load local variable on stack</td>
<td>3.43</td>
</tr>
<tr>
<td>CIL_LDLOCA_INSTRUCTION</td>
<td>Load address of local variable on stack</td>
<td>3.44</td>
</tr>
<tr>
<td>CIL_STLOC_INSTRUCTION</td>
<td>Store stack item to local variable</td>
<td>3.63</td>
</tr>
</tbody>
</table>

\(^a\) Section of [9]

CIL\_PARAMETER\_VARIABLE\_INSTRUCTION

CIL\_PARAMETER\_VARIABLE\_INSTRUCTION is the deferred ancestor of all effective instruction classes referring to a parameter variable (i.e., routine argument in Eiffel), CIL\_VARIABLE\_INSTRUCTION is its direct ancestor of which it redefines the attribute reference of type CIL\_VARIABLE\_REFERENCE to type CIL\_PARAMETER\_VARIABLE\_REFERENCE.

The descendants of CIL\_PARAMETER\_VARIABLE\_INSTRUCTION effect the command visit of the method body item Visitor pattern (see section 4.13 on page 47) and do not add or change any other feature.

Table 4.5: Descendants of CIL\_PARAMETER\_VARIABLE\_INSTRUCTION

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_LDARG_INSTRUCTION</td>
<td>Load parameter on stack</td>
<td>3.38</td>
</tr>
<tr>
<td>CIL_LDARGA_INSTRUCTION</td>
<td>Load address of parameter on stack</td>
<td>3.39</td>
</tr>
<tr>
<td>CIL_STARG_INSTRUCTION</td>
<td>Store stack item to parameter</td>
<td>3.61</td>
</tr>
</tbody>
</table>

\(^a\) Section of [9]

CIL\_NONE\_INSTRUCTION

CIL\_NONE\_INSTRUCTION \(^1\) directly inherits from CIL\_INSTRUCTION and is the deferred ancestor of all effective instruction classes without additional attributes. As such it serves as a structural part within the inheritance hierarchy and does not change or add any features.

The descendants of CIL\_NONE\_INSTRUCTION effect the command visit of the method body item Visitor pattern (see section 4.13 on page 47) and do not add or change any other feature.

Table 4.6: Descendants of CIL\_NONE\_INSTRUCTION

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_ADD_INSTRUCTION</td>
<td>Add two values</td>
<td>3.1(^b)</td>
</tr>
<tr>
<td>CIL_ADD_OVF_INSTRUCTION</td>
<td>Add two signed integers with overflow check</td>
<td>3.2(^b)</td>
</tr>
</tbody>
</table>

\(^1\) The class name CIL\_NONE\_INSTRUCTION comes from the naming in [11, section C.3. and C.4.2.]. The corresponding nonterminal symbol of the parser description file (see section 3.1 on page 14) is Instruction\_none. The class name CIL\_NONE\_INSTRUCTION is not related to the Eiffel class NONE.
### Descendants of CIL_NONE_INSTRUCTION (continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_ADD_OVF_UN_INSTRUCTION</td>
<td>Add two unsigned values with overflow check</td>
<td>3.28</td>
</tr>
<tr>
<td>CIL_AND_INSTRUCTION</td>
<td>Bitwise AND of two integers</td>
<td>3.3n</td>
</tr>
<tr>
<td>CIL_ARGLIST_INSTRUCTION</td>
<td>Load argument list</td>
<td>3.4</td>
</tr>
<tr>
<td>CIL_BREAK_INSTRUCTION</td>
<td>Breakpoint</td>
<td>3.16</td>
</tr>
<tr>
<td>CIL_CEQ_INSTRUCTION</td>
<td>Compare for equality</td>
<td>3.21p</td>
</tr>
<tr>
<td>CIL_CGT_INSTRUCTION</td>
<td>Compare for greater (signed / ordered)</td>
<td>3.22p</td>
</tr>
<tr>
<td>CIL_CGT_UN_INSTRUCTION</td>
<td>Compare for greater (unsigned / unordered)</td>
<td>3.23p</td>
</tr>
<tr>
<td>CIL_CKFINITE_INSTRUCTION</td>
<td>Check for a finite floating-point number</td>
<td>3.24</td>
</tr>
<tr>
<td>CIL_CLT_INSTRUCTION</td>
<td>Compare for less (signed / ordered)</td>
<td>3.25p</td>
</tr>
<tr>
<td>CIL_CLT_UN_INSTRUCTION</td>
<td>Compare for less (unsigned / unordered)</td>
<td>3.26p</td>
</tr>
<tr>
<td>CIL_CONV_R_UN_INSTRUCTION</td>
<td>Convert unsigned integer to floating-point number</td>
<td>3.27p</td>
</tr>
<tr>
<td>CIL_CPBLK_INSTRUCTION</td>
<td>Copy data in memory</td>
<td>3.30</td>
</tr>
<tr>
<td>CIL_DIV_INSTRUCTION</td>
<td>Divide two signed numbers</td>
<td>3.31p</td>
</tr>
<tr>
<td>CIL_DIV_UN_INSTRUCTION</td>
<td>Divide two unsigned integers</td>
<td>3.32p</td>
</tr>
<tr>
<td>CIL_DUP_INSTRUCTION</td>
<td>Duplicate stack item</td>
<td>3.33</td>
</tr>
<tr>
<td>CIL_ENDFILTER_INSTRUCTION</td>
<td>End filter clause</td>
<td>3.34</td>
</tr>
<tr>
<td>CIL_ENDFINALLY_INSTRUCTION</td>
<td>End finally or fault clause</td>
<td>3.35</td>
</tr>
<tr>
<td>CIL_INITBLK_INSTRUCTION</td>
<td>Initialize data in memory</td>
<td>3.36</td>
</tr>
<tr>
<td>CIL_LDLEN_INSTRUCTION</td>
<td>Length of array</td>
<td>4.11</td>
</tr>
<tr>
<td>CIL_LDNULL_INSTRUCTION</td>
<td>Load null pointer</td>
<td>3.45</td>
</tr>
<tr>
<td>CILToLocalALLOC_INSTRUCTION</td>
<td>Allocate space in local memory pool</td>
<td>3.47</td>
</tr>
<tr>
<td>CIL_MUL_INSTRUCTION</td>
<td>Multiply two signed numbers</td>
<td>3.48p</td>
</tr>
<tr>
<td>CIL_MUL_OVF_INSTRUCTION</td>
<td>Multiply two signed integers with overflow check</td>
<td>3.49p</td>
</tr>
<tr>
<td>CIL_MUL_OVF_UN_INSTRUCTION</td>
<td>Multiply two unsigned integers with overflow check</td>
<td>3.49p</td>
</tr>
<tr>
<td>CIL_NEG_INSTRUCTION</td>
<td>Negate a number</td>
<td>3.50p</td>
</tr>
<tr>
<td>CIL_NOP_INSTRUCTION</td>
<td>No operation</td>
<td>3.51</td>
</tr>
</tbody>
</table>
Descendants of CIL_NONE_INSTRUCTION (continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_NOT_INSTRUCTION</td>
<td>Bitwise complement of an integer</td>
<td>3.52^a</td>
</tr>
<tr>
<td>CIL_OR_INSTRUCTION</td>
<td>Bitwise OR of two integers</td>
<td>3.53^a</td>
</tr>
<tr>
<td>CIL_POP_INSTRUCTION</td>
<td>Remove stack item</td>
<td>3.54</td>
</tr>
<tr>
<td>CIL_REFANYTYPE_INSTRUCTION</td>
<td>Type of typed reference</td>
<td>4.21</td>
</tr>
<tr>
<td>CIL_REM_INSTRUCTION</td>
<td>Remainder of two signed numbers</td>
<td>3.55^b</td>
</tr>
<tr>
<td>CIL_REM_UN_INSTRUCTION</td>
<td>Remainder of two unsigned integers</td>
<td>3.56^b</td>
</tr>
<tr>
<td>CIL_RET_INSTRUCTION</td>
<td>Return from method</td>
<td>3.57</td>
</tr>
<tr>
<td>CIL_RETHROW_INSTRUCTION</td>
<td>Re-throw current exception</td>
<td>4.23</td>
</tr>
<tr>
<td>CIL_SHL_INSTRUCTION</td>
<td>Shift bits of an integer left</td>
<td>3.58^f</td>
</tr>
<tr>
<td>CIL_SHR_INSTRUCTION</td>
<td>Shift bits of a signed integer right</td>
<td>3.59^g</td>
</tr>
<tr>
<td>CIL_SHR_UN_INSTRUCTION</td>
<td>Shift bits of an unsigned integer right</td>
<td>3.60^g</td>
</tr>
<tr>
<td>CIL_SUB_INSTRUCTION</td>
<td>Subtract two values</td>
<td>3.64^b</td>
</tr>
<tr>
<td>CIL_SUB_OVF_INSTRUCTION</td>
<td>Subtract two signed integers with overflow check</td>
<td>3.65^g</td>
</tr>
<tr>
<td>CIL_SUB_OVF_UN_INSTRUCTION</td>
<td>Subtract two unsigned values with overflow check</td>
<td>3.65^g</td>
</tr>
<tr>
<td>CIL_THROW_INSTRUCTION</td>
<td>Throw an exception</td>
<td>4.29</td>
</tr>
<tr>
<td>CIL_VOLATILE_INSTRUCTION</td>
<td>Volatile pointer reference</td>
<td>2.3</td>
</tr>
<tr>
<td>CIL_XOR_INSTRUCTION</td>
<td>Bitwise XOR of two integers</td>
<td>3.67</td>
</tr>
</tbody>
</table>

^a Section of [9]
^b See also table 2: Binary Numeric Operations of [9]
^c See also table 3: Unary Numeric Operations of [9]
^d See also table 4: Binary Comparison or Branch Operations of [9]
^e See also table 5: Integer Operations of [9]
^f See also table 6: Shift Operations of [9]
^g See also table 7: Overflow Arithmetic Operations of [9]
^h See also table 8: Conversion Operations of [9]

Note 4.30 CIL_VOLATILE_INSTRUCTION represents a prefix instruction, it would be convenient to have a boolean query is_volatile for the instructions that can follow the volatile prefix instead of a separate class for the prefix itself.
CIL\_SIGNATURE\_INSTRUCTION

CIL\_SIGNATURE\_INSTRUCTION directly inherits from CIL\_INSTRUCTION and is the deferred ancestor of all effective instruction classes having a method signature. It implements a creation routine \textit{make}, an attribute \textit{signature} and a corresponding setter \textit{set\_signature}.

The descendants of CIL\_SIGNATURE\_INSTRUCTION effect the command \textit{visit} of the method body item Visitor pattern (see section 4.13 on page 47) and do not add or change any other feature.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_CALL_INSTRUCTION</td>
<td>Indirect method call</td>
<td>3.20</td>
</tr>
<tr>
<td>CIL_TAIL_CALL_INSTRUCTION</td>
<td>Indirect method call, return to caller on return from callee</td>
<td>3.20\textsuperscript{b}</td>
</tr>
</tbody>
</table>

\textsuperscript{a} Section of [9]  
\textsuperscript{b} See also [9, section 2.1]

CIL\_TOKEN\_INSTRUCTION

CIL\_TOKEN\_INSTRUCTION directly inherits from CIL\_INSTRUCTION and is the deferred ancestor of all effective instruction classes with a token owner type. It implements a creation routine \textit{make}, an attribute \textit{reference} and a corresponding setter \textit{set\_reference}.

The only effective direct descendant of CIL\_TOKEN\_INSTRUCTION, CIL\_LDTOKEN\_INSTRUCTION (see [9, section 4.16]), effects the command \textit{visit} of the method body item Visitor pattern (see section 4.13 on page 47) and does not add or change any other feature.

The only deferred direct descendant of CIL\_TOKEN\_INSTRUCTION, CIL\_MEMBER\_INSTRUCTION, redefines \textit{reference} of type CIL\_TOKEN\_OWNER to type CIL\_MEMBER\_REFERENCE. The class CIL\_MEMBER\_INSTRUCTION has two direct descendants, the first is CIL\_FIELD\_INSTRUCTION (see section 4.5) and the second is CIL\_METHOD\_INSTRUCTION (see section 4.5 on the next page), both are deferred.

CIL\_FIELD\_INSTRUCTION

The class CIL\_FIELD\_INSTRUCTION directly inherits from the class CIL\_MEMBER\_INSTRUCTION and is the deferred ancestor of all effective instruction classes referring to a field. CIL\_FIELD\_INSTRUCTION redefines \textit{reference} of type CIL\_MEMBER\_REFERENCE to type CIL\_FIELD\_REFERENCE.

The descendants of CIL\_FIELD\_INSTRUCTION effect the command \textit{visit} of the method body item Visitor pattern (see section 4.13 on page 47) and do not add or change any other feature.
### Table 4.8: Descendants of CIL_FIELD_INSTRUCTION

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_LDFLD_INSTRUCTION</td>
<td>Load from a field of an object</td>
<td>4.9</td>
</tr>
<tr>
<td>CIL_LDFLDA_INSTRUCTION</td>
<td>Load address of a field of an</td>
<td>4.10</td>
</tr>
<tr>
<td></td>
<td>object</td>
<td></td>
</tr>
<tr>
<td>CIL_LDSFLD_INSTRUCTION</td>
<td>Load from a static field of a</td>
<td>4.13</td>
</tr>
<tr>
<td></td>
<td>class</td>
<td></td>
</tr>
<tr>
<td>CIL_LDSFLDA_INSTRUCTION</td>
<td>Load address of a static field of a</td>
<td>4.14</td>
</tr>
<tr>
<td></td>
<td>class</td>
<td></td>
</tr>
<tr>
<td>CIL_STFLD_INSTRUCTION</td>
<td>Store to a field of an object</td>
<td>4.26</td>
</tr>
<tr>
<td>CIL_STSFLD_INSTRUCTION</td>
<td>Store to a static field of a class</td>
<td>4.28</td>
</tr>
</tbody>
</table>

*a* Section of [9]

### CIL_METHOD_INSTRUCTION

The class CIL_METHOD_INSTRUCTION directly inherits from the class CIL_MEMBER_INSTRUCTION and is the deferred ancestor of all effective instruction classes referring to a method. CIL_METHOD_INSTRUCTION redefines the attribute `reference` of type CIL_MEMBER_REFERENCE to be of type CIL_METHOD_REFERENCE.

The descendants of CIL_METHOD_INSTRUCTION effect the command visit of the method body item Visitor pattern (see section 4.13 on page 47) and do not add or change any other feature.

### Table 4.9: Descendants of CIL_METHOD_INSTRUCTION

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_CALL_INSTRUCTION</td>
<td>Call a method</td>
<td>3.19</td>
</tr>
<tr>
<td>CIL_CALLVIRT_INSTRUCTION</td>
<td>Call a dynamically linked method</td>
<td>4.2</td>
</tr>
<tr>
<td>CIL_JMP_INSTRUCTION</td>
<td>Jump to a method</td>
<td>3.37</td>
</tr>
<tr>
<td>CIL_LDFTN_INSTRUCTION</td>
<td>Load a pointer to a method</td>
<td>3.41</td>
</tr>
<tr>
<td>CIL_LDFVIRTFTN_INSTRUCTION</td>
<td>Load a pointer to a dynamically</td>
<td>4.17</td>
</tr>
<tr>
<td></td>
<td>linked method</td>
<td></td>
</tr>
<tr>
<td>CIL_NEWOBJ_INSTRUCTION</td>
<td>Create a new object</td>
<td>4.20</td>
</tr>
</tbody>
</table>
| CIL_TAIL_CALL_INSTRUCTION | Call a method, return to caller on return from callee | 3.19
| CIL_TAIL_CALLVIRT_INSTRUCTION | Call a dynamically linked method, return to caller on return from callee | 4.2

*a* Section of [9]

*b* See also [9, section 2.1]
CIL_TYPE_INSTRUCTION

CIL_TYPE_INSTRUCTION directly inherits from CIL_INSTRUCTION and is the deferred ancestor of all effective instruction classes having a type. It implements a creation routine `make`, an attribute `type` and a corresponding setter `setType`.

The descendants of CIL_TYPE_INSTRUCTION effect the command `visit` of the method body item Visitor pattern (see section 4.13 on page 47) and do not add or change any other feature.

Table 4.10: Descendants of CIL_TYPE_INSTRUCTION

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_BOX_INSTRUCTION</td>
<td>Create object reference from value type</td>
<td>4.1</td>
</tr>
<tr>
<td>CIL_CASTCLASS_INSTRUCTION</td>
<td>Type cast</td>
<td>4.3</td>
</tr>
<tr>
<td>CIL_CONV_INSTRUCTION</td>
<td>Convert between numeric types</td>
<td>3.27</td>
</tr>
<tr>
<td>CIL_CONV_OVF_INSTRUCTION</td>
<td>Convert between signed numeric types with overflow check</td>
<td>3.28</td>
</tr>
<tr>
<td>CIL_CONV_OVF_UN_INSTRUCTION</td>
<td>Convert between unsigned numeric types with overflow check</td>
<td>3.29</td>
</tr>
<tr>
<td>CIL_CPOBJ_INSTRUCTION</td>
<td>Copy value type in memory</td>
<td>4.4</td>
</tr>
<tr>
<td>CIL_INITOBJ_INSTRUCTION</td>
<td>Initialize value type in memory</td>
<td>4.5</td>
</tr>
<tr>
<td>CIL_ISINST_INSTRUCTION</td>
<td>Test object for instance of class or interface</td>
<td>4.6</td>
</tr>
<tr>
<td>CIL_LDELEM_INSTRUCTION</td>
<td>Load element from array</td>
<td>4.7</td>
</tr>
<tr>
<td>CIL_LDELEMA_INSTRUCTION</td>
<td>Load address of array element</td>
<td>4.8</td>
</tr>
<tr>
<td>CIL_LDIND_INSTRUCTION</td>
<td>Load value from address</td>
<td>3.42</td>
</tr>
<tr>
<td>CIL_LDOBJ_INSTRUCTION</td>
<td>Load value type from address</td>
<td>4.12</td>
</tr>
<tr>
<td>CIL_MKREFANY_INSTRUCTION</td>
<td>Create a typed reference</td>
<td>4.18</td>
</tr>
<tr>
<td>CIL_NEWARR_INSTRUCTION</td>
<td>Create an array</td>
<td>4.19</td>
</tr>
<tr>
<td>CIL_REFANYVAL_INSTRUCTION</td>
<td>Address value of typed reference</td>
<td>4.22</td>
</tr>
<tr>
<td>CIL_SIZEOF_INSTRUCTION</td>
<td>Size of a value type in bytes</td>
<td>4.24</td>
</tr>
<tr>
<td>CIL_STELEM_INSTRUCTION</td>
<td>Store element in array</td>
<td>4.25</td>
</tr>
<tr>
<td>CIL_STIND_INSTRUCTION</td>
<td>Store value to address</td>
<td>3.62</td>
</tr>
<tr>
<td>CIL_STOBJ_INSTRUCTION</td>
<td>Store value type to address</td>
<td>4.27</td>
</tr>
</tbody>
</table>
Descendants of CIL\_TYPE\_INSTRUCTION (continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Sectiona</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_UNBOX_INSTRUCTION</td>
<td>Create value type from object reference</td>
<td>4.30</td>
</tr>
</tbody>
</table>

a Section of [9]

b See also table 8: Conversion Operations of [9]

**NOTE 4.31** As CIL\_CONV\_INSTRUCTION, CIL\_CONV\_OVF\_INSTRUCTION and CIL\_CONV\_OVF\_UN\_INSTRUCTION only handle numeric types a descendant CIL\_NUMERIC\_TYPE\_INSTRUCTION of CIL\_TYPE\_INSTRUCTION would be convenient. The class CIL\_NUMERIC\_TYPE\_INSTRUCTION would redefine the attribute type to be of type CIL\_NUMERIC\_TYPE and the three conversion instructions could inherit from it. Similarly for other instructions.

**Maximum Stack Count**

Each method body has an item .maxstack (see [8, section 14.4.1]) that defines the maximum stack count during execution of the method. This body item is implemented by the class CIL\_MAX\_STACK, which provides an attribute value along with a creation routine and a setter for initialization and status setting.

CIL\_MAX\_STACK also effects the accept command of the method body item Visitor pattern (see section 4.13 on page 47).

**Overriding methods**

The .override (see [8, section 14.4.1 and 9.3.2]) method body item specifies that the current method should override a specified method. The class CIL\_OVERRIDE\_BODY\_ITEM represents these body items, it inherits from CIL\_OVERRIDE (see section 4.8 on page 42 and CIL\_METHOD\_BODY\_ITEM). The class CIL\_OVERRIDE provides the attributes type and method name and corresponding setters, CIL\_OVERRIDE\_BODY\_ITEM adds a creation routine for initializing both attributes.

The class CIL\_OVERRIDE\_BODY\_ITEM also effects the accept command of the method body item Visitor pattern (see section 4.13 on page 47).

**Parameter Associated Values**

The .param directive (see [8, section 14.4.1.4]) associates a parameter index with a value. The index 0 whereby refers to the return value (unlike in instructions where it refers to the this object reference). This method body item is implemented by the class CIL\_PARAMETER\_VALUE with the attributes index of type INTEGER and value of type CIL\_FIELD\_INIT accompanied with a corresponding creation routine and setters. Because the value is optional CIL\_PARAMETER\_VALUE also provides a status report query is value set.

The class CIL\_PARAMETER\_VALUE effects the accept command of the method body item Visitor pattern (see section 4.13 on page 47).
Scope Blocks

A scope block contains a sequence of method body items and is itself a method body item. However scope blocks as method body item are not documented in the standard but supported by the grammar in [11, section C.3.]. The use of scope blocks as part of an exception block is documented in section 4.5 on page 26 and [8, section 18]. Scope blocks are implemented by the class CIL_SCOPE_BLOCK which inherits from CIL_METHOD_BODY_ITEM and CIL_METHOD_BODY_ITEMSCONTEXT (see section 4.8 on page 42).

CIL_SCOPE_BLOCK additionally provides a creation routine make for initializing body_items from CIL_METHOD_BODY_ITEMSCONTEXT and effects the accept command of the method body item Visitor pattern (see section 4.13 on page 47).

Virtual Table Entries

The .vtentry directive is supported by the grammar in [11, section C.3.] but not documented in the standard (see [8, section 14.4.1]).

Note 4.32 The .vtentry method body item is currently not implemented in the AST classes.

Zero Initialization

The .zeroinit method body item is an alternative for specifying that the local variables should be initialized to their default values (see section 4.5 on page 26). However this directive is not documented in the standard but is supported by the grammar in [11, section C.3.] and ilasm [16].

The class CIL_ZERO_INIT implements this body item and has no features but the command accept from the method body item Visitor pattern (see section 4.13 on page 47), which it effects.

4.6 Entities, Type Members and Method Body Items

There are a few metadata constructs that can occur as entities, type members and method body items. These are documented in this section.

Custom Attributes

Custom attributes as documented in [8, section 20] and supported by the grammar in [11, section C.3.] are implemented in CIL_CUSTOM_ATTRIBUTE. Note that the grammar in [11, section C.3.] supports more options than is documented in the standard in [8, section 20].

Custom attribute declarations occur as entities, type members, method body items, event members (see section 4.3 on page 20), property members (see section 4.3 on page 21), assembly declaration members (see section 4.2 on page 19), external type declaration members (see section 4.2 on page 19), export declaration members (see section 4.3 on page 22) and manifest resource declaration members (see section 4.2 on page 19).
The class CIL\_CUSTOM\_ATTRIBUTE effects the \texttt{accept} commands of the entity Visitor pattern inherited from CIL\_ENTITY (which is renamed as \texttt{accept\_entity\_visitor}, see section 4.13 on page 47), the type member Visitor pattern inherited from CIL\_TYPE\_MEMBER (which is renamed as \texttt{accept\_type\_member\_visitor}, see section 4.13 on page 47) and the method body item Visitor pattern inherited from CIL\_METHOD\_BODY\_ITEM (which is renamed as \texttt{accept\_method\_body\_item\_visitor}, see section 4.13 on page 47).

\textbf{Note 4.33} While CIL\_CUSTOM\_ATTRIBUTE also inherits from the class CIL\_EVENT\_MEMBER and CIL\_PROPERTY\_MEMBER, the parser’s actions (see section 3.2 on page 14) currently only supports custom attributes as members of the top-level module, type definitions and method bodies.

\textbf{Embedded Data}

Data that is embedded in a module is documented in [8, section 15.3] and can occur as an entity, a type member and a method body item. Embedded data is implemented by the class CIL\_DATA with the attributes \texttt{identifier} of type CIL\_IDENTIFIER denoting the optional data label and \texttt{items} of type \texttt{LIST [CIL\_DATA\_ITEM]}. There is an undocumented flag \texttt{tls} in the grammar (see [11, section C.3,]) that appears to be relevant for thread local storage and is represented by the status report attribute \texttt{is\_tls}.

CIL\_DATA also has the setters \texttt{set\_is\_tls}, \texttt{set\_identifier} and \texttt{set\_items} for setting its attributes and a status report query \texttt{is\_identifier\_set}.

The class CIL\_DATA also effects the \texttt{accept} commands of the entity Visitor pattern inherited from CIL\_ENTITY (renamed as \texttt{accept\_entity\_visitor}, see section 4.13 on page 47), the type member Visitor pattern inherited from CIL\_TYPE\_MEMBER (renamed as \texttt{accept\_type\_member\_visitor}, see section 4.13 on page 47) and the method body item Visitor pattern inherited from CIL\_METHOD\_BODY\_ITEM (renamed as \texttt{accept\_method\_body\_item\_visitor}, see section 4.13 on page 47).

\textbf{Data Items}

Data items are implemented by the deferred class CIL\_DATA\_ITEM and its effective descendants where the numeric data items inherit from the intermediate deferred descendant of CIL\_DATA\_ITEM—CIL\_NUMERIC\_DATA\_ITEM.

\textbf{External Source Positions}

The standard allows to specify information about the lexical scope of variables and a mapping from source line numbers to CIL instructions by the \texttt{.line} directive. See also [8, section 5.7].

The \texttt{.line} directive for specifying external source information is implemented in the class CIL\_EXTERNAL\_SOURCE\_LINE, which provides the attributes \texttt{line\_number} of type \texttt{INTEGER}, an optional \texttt{column\_number} of type \texttt{INTEGER} and an optional \texttt{referred\_file\_name} of type \texttt{CIL\_QUOTED\_STRING} (see section 4.11 on page 45) accompanied by the corresponding setters. CIL\_EXTERNAL\_SOURCE\_LINE also includes the status report attribute \texttt{is\_column\_number\_set} with the corresponding setter \texttt{set\_is\_column\_number\_set}
and the query `is_referred_file_name_set`. The creation routines `make` and `make_with_column` are also provided.

External source line declarations occur as entities, type members, method body items, event members (see section 4.3 on page 20) and property members (see section 4.3 on page 21).

The class `CIL_EXTERNALSOURCE_LINE` effects the `accept` commands of the entity Visitor pattern inherited from `CIL_ENTITY` (which is renamed as `accept_entity_visitor`, see section 4.13 on page 47), the type member Visitor pattern inherited from `CIL_TYPE_MEMBER` (which is renamed as `accept_type_member_visitor`, see section 4.13 on page 47) and the method body item Visitor pattern inherited from `CIL_METHOD_BODY_ITEM` (which is renamed as `accept_method_body_item_visitor`, see section 4.13 on page 47).

**Note 4.34** While `CIL_EXTERNALSOURCE_LINE` also inherits from `CIL_EVENT_MEMBER` and `CIL_PROPERTY_MEMBER`, the parser’s actions (see section 3.2 on page 14) currently only supports custom attributes as members of the top-level module, type definitions and method bodies.

**Language Declarations**

The grammar allows to specify language information by the `.language` directive, this is not documented by the standard (see also [8, section 6.2.1.2] about specifying an assembly wide culture). See also [11, section C.3].

The `.language` directive for specifying language information corresponds to the class `CIL_LANGUAGE_DECLARATION`, which provides the attribute `languages` of type `LIST [CIL_SINGLY_QUOTED_STRING]` (see section 4.11 on page 45) with a corresponding setter `set_languages` and creation routine `make`.

Language declarations occur as entities, type members, method body items, event members (see section 4.3 on page 20) and property members (see section 4.3 on page 21).

The class `CIL_LANGUAGE_DECLARATION` effects the `accept` commands of the entity Visitor pattern inherited from `CIL_ENTITY` (which is renamed as `accept_entity_visitor`, see section 4.13 on page 47), the type member Visitor pattern inherited from `CIL_TYPE_MEMBER` (which is renamed as `accept_type_member_visitor`, see section 4.13 on page 47) and the method body item Visitor pattern inherited from `CIL_METHOD_BODY_ITEM` (which is renamed as `accept_method_body_item_visitor`, see section 4.13 on page 47).

**Note 4.35** While `CIL_LANGUAGE_DECLARATION` also inherits from `CIL_EVENT_MEMBER` and `CIL_PROPERTY_MEMBER`, the parser’s actions (see section 3.2 on page 14) currently only supports custom attributes as members of the top-level module, type definitions and method bodies.

**Security Attributes**

A security attribute declares an action and settings associated with that action. There are two variants for declaring the settings (see sections 4.6 on the next page and 4.6 on the following page). See also [8, section 19].

Security attributes are implemented in the deferred class `CIL_SECURITY_ATTRIBUTE` and its effective descendants (see subsec-
tions below). The class CIL_SECURITY_ATTRIBUTE has an attribute action of type CIL_SECURITY_ACTION and a corresponding setter set_action.

Security attributes appear as entities, type members, method body items and assembly declaration members (see section 4.2 on page 19).

Note 4.36 The only effective descendant of the deferred class CIL_SECURITY_ACTION, CILGENERAL_SECURITY_ACTION, inherits its implementation from CILGENERAL_ATTRIBUTE (see section 4.8 on the following page).

Manifest Security Attributes

Manifest security attributes are implemented by the class CIL_MANIFEST_SECURITY_ATTRIBUTE, which provides the attributes type of type CIL_TYPE_SPECIFICATION and settings of type LIST [CIL_NAME_VALUE_PAIR]. type specifies the permission type. The class is complemented by the setters set_type and set_settings and the creation routine make.

The class CIL_MANIFEST_SECURITY_ATTRIBUTE effects the accept commands of the entity Visitor pattern inherited from CIL_ENTITY (which is renamed as accept_entity_visitor, see section 4.13 on page 47), the type member Visitor pattern inherited from CIL_TYPE_MEMBER (which is renamed as accept_type_member_visitor, see section 4.13 on page 47) and the method body item Visitor pattern inherited from CIL_METHOD_BODY_ITEM (which is renamed as accept_method_body_item_visitor, see section 4.13 on page 47).

Serialized Security Attributes

Serialized security attributes are implemented by the class CIL_SERIALIZED_SECURITY_ATTRIBUTE, which provides the attribute serialized_settings of type LIST [INTEGER_8], a corresponding setter set_type and a creation routine make.

The class CIL_SERIALIZED_SECURITY_ATTRIBUTE effects the accept commands of the entity Visitor pattern inherited from CIL_ENTITY (which is renamed as accept_entity_visitor, see section 4.13 on page 47), the type member Visitor pattern inherited from CIL_TYPE_MEMBER (which is renamed as accept_type_member_visitor, see section 4.13 on page 47) and the method body item Visitor pattern inherited from CIL_METHOD_BODY_ITEM (which is renamed as accept_method_body_item_visitor, see section 4.13 on page 47).

4.7 Method Signatures

As method signatures are used at various places in the AST they have been factored out and implemented in the class CIL_METHOD_SIGNATURE. The class CIL_METHOD_SIGNATURE has the attributes calling_convention of type CIL_CALLING_CONVENTION (see section 4.7 on the next page), return_type of type CIL_TYPE and parameters of type LIST [CIL_PARAMETER]. These are accompanied by corresponding setters and a creation routine make.
Calling Convention

Calling conventions for methods are implemented in the class CIL\_CALLING\_CONVENTION. If the status report attribute \texttt{is\_instance} is set to \texttt{true} the corresponding method receives a \texttt{this} pointer. If additionally \texttt{is\_explicit} is set to \texttt{true} the type of the \texttt{this} pointer is explicitly given by the first item of the parameter list of the associated method. There is also an optional attribute \texttt{kind} of type CIL\_CALLING\_KIND, a corresponding status report query \texttt{is\_kind\_set}, the setters \texttt{set\_kind}, \texttt{set\_is\_instance}, \texttt{set\_is\_explicit} and a creation routine \texttt{make}. See also [8, section 14.3].

Note 4.37 The only effective descendant of the deferred class CIL\_CALLING\_KIND, CIL\_GENERAL\_CALLING\_KIND, inherits its implementation from CIL\_GENERAL\_ATTRIBUTE (see section 4.8).

Method Parameters

Method parameters are implemented by descendants of CIL\_PARAMETER: CIL\_PARAMETER\_ELLIPSIS and CIL\_PARAMETER\_VARIABLE. CIL\_PARAMETER and its descendant CIL\_PARAMETER\_ELLIPSIS both define no features since an ellipsis parameter stands for a variable number of otherwise unspecified actual method parameters.

The class CIL\_PARAMETER\_VARIABLE also inherits from CIL\_VARIABLE (see section 4.9 on page 43) and adds the attributes \texttt{attributes} of type \texttt{LIST[CIL\_PARAMETER\_ATTRIBUTE]} and the optional \texttt{native\_type} of type CIL\_NATIVE\_TYPE (see section 4.8 on the following page) and a corresponding status report query \texttt{is\_native\_type\_set}. CIL\_PARAMETER\_VARIABLE also provides the setters \texttt{set\_attributes} and \texttt{set\_native\_type} and a creation routine \texttt{make}. See also [8, section 14.4].

Note 4.38 The only effective descendants of the deferred class CIL\_PARAMETER\_ATTRIBUTE are CIL\_GENERAL\_PARAMETER\_ATTRIBUTE inheriting its implementation from CIL\_GENERAL\_ATTRIBUTE (see section 4.8) and CIL\_INTEGER\_PARAMETER\_ATTRIBUTE having an attribute \texttt{value} of type INTEGER. CIL\_INTEGER\_PARAMETER\_ATTRIBUTE is not documented in the standard but supported by the grammar in [8, section C.3.].

4.8 Miscellaneous

General Attributes

The deferred class CIL\_GENERAL\_ATTRIBUTE provides an implementation for attributes that are not fully implemented yet. It provides an attribute \texttt{definition} of type \texttt{STRING} and a corresponding setter \texttt{set\_definition} and creation routine \texttt{make}. Various effective attribute classes inherit from this class and clients set the \texttt{definition} on instances of these descendants of CIL\_GENERAL\_ATTRIBUTE to the CIL keyword corresponding to the represented attribute. This is only a provisional solution.
Method Body Items context

The deferred class CIL_METHOD_BODY_ITEMS_CONTEXT has an attribute \texttt{body\_items} of type \texttt{LIST [CIL\_METHOD\_BODY\_ITEM]}, a corresponding setter \texttt{set\_body\_items} and a convenience command \texttt{extend} for extending \texttt{body\_items}. CIL_METHOD_BODY_ITEMS_CONTEXT is the base class for body item contenders like: CIL_METHOD_DEFINITION (see section 4.4 on page 23) and CIL_SCOPE_BLOCK (see section 4.5 on page 37). CIL_METHOD_BODY_ITEMS_CONTEXT also serves as the type of context of CIL_METHOD_BODY_ITEM_VISITOR (see 4.13 on page 47).

Native Types

Native types provide optional marshaling information for method return values, method parameters and fields. Native types are represented by instances of type CIL_NATIVE_TYPE.

Note 4.39 The class CIL\_NATIVE\_TYPE is currently empty and as such unimplemented, it also has no descendants.

Overriding Methods

The standard specifies two possibilities for overriding a method by a method with a different name: The first by a type member implemented by the class CIL\_OVERRIDE\_MEMBER (see section 4.3 on page 22) and the second by a method body item implemented by the class CIL\_OVERRIDE\_BODY\_ITEM (see section 4.5 on page 36). Both of these classes inherit from the hereby documented deferred class CIL\_OVERRIDE. CIL\_OVERRIDE has the attributes \texttt{type} of type CIL\_TYPE\_SPECIFICATION, specifying the type of which the overridden method is a member, and \texttt{method\_name} of type CIL\_NAME, specifying the overridden method’s name. CIL\_OVERRIDE also provides corresponding setters \texttt{set\_type} and \texttt{set\_method\_name}. See also [8, section 9.3.2 and 14.4.1].

4.9 Referable Nodes

There are various kinds of AST nodes that can be referred to, the important referable AST nodes all inherit—usually indirectly—from the empty deferred class CIL\_REFERENT. Subsequently, there is a parallel inheritance hierarchy of referencing AST nodes inheriting from the deferred class CIL\_REFERENCE (see section 4.10 on the next page).

There are currently two important categories of descendants of CIL\_REFERENT documented in the following subsections.

Token Owner Nodes

Token owners inherit from CIL\_TOKEN\_OWNER, a direct descendant of CIL\_REFERENT.

The important descendants of CIL\_TOKEN\_OWNER are CIL\_TYPE\_DEFINITION (see section 4.4 on page 23),
CIL_METHOD_DEFINITION (see section 4.4 on page 23) and CIL_FIELD_DEFINITION (see section 4.4 on page 24). CIL_TYPE_DEFINITION and CIL_MEMBER_DEFINITION are direct descendants of CIL_TOKEN_OWNER, where CIL_MEMBER_DEFINITION is the direct ancestor of CIL_METHOD_DEFINITION and CIL_FIELD_DEFINITION.

Referable Local Nodes

Referable local nodes inherit from the deferred class CIL_LOCAL, a direct descendant of CILREFERENT. CIL_LOCAL has an attribute identifier of class CIL_IDENTIFIER, which may be void if the deferred status report query may_identifier_be_void is true. There is also a setter set_identifier and another status report query is_identifier_set.

The direct descendants of CIL_LOCAL are the effective class CIL_LABEL, whose identifier “may not be void”, and the deferred class CIL_VARIABLE, whose identifier “may be void”. CIL_VARIABLE also defines an attribute type of type CIL_TYPE, specifying the type of the variable, with a corresponding setter set_type.

The direct effective descendants of CIL_VARIABLE are CIL_PARAMETER_VARIABLE (see section 4.7 on page 41) and CIL_LOCAL_VARIABLE, the latter only adding a creation routine make to the inherited features.

4.10 Referencing Nodes

The counterpart to referable nodes (see section 4.9 on the preceding page) are the referencing nodes inheriting from the deferred class CIL_REFERENCE. CIL_REFERENCE defines an attribute referent of type CIL_REFERENT, which is covariantly redefined in the descendants, step by step following the inheritance hierarchy of CIL_REFERENT. CIL_REFERENCE also supplies a setter set_referent and a status report query is_resolved.

There are currently two important categories of descendants of CIL_REFERENCE documented in the following subsections.

Tokens

Tokens reference token owners (see section 4.9 on the page before) and are implemented by the deferred class CIL_TOKEN. CIL_TOKEN directly inherits from CIL_REFERENCE and redefines referent to be of type CIL_TOKEN_OWNER. See also section 4.5 on page 33 and [8, section C.4.13.] on instructions taking a token as their argument.

CIL_TOKEN has two direct descendants: CIL_TYPE_SPECIFICATION (see below) and CIL_MEMBER_REFERENCE. CIL_MEMBER_REFERENCE is the direct ancestor of CIL_METHOD_REFERENCE and CIL_FIELD_REFERENCE. Each of these descendants of CIL_TOKEN covariantly redefine the type of referent (i.e., CIL_TYPE_DEFINITION, CIL_MEMBER_DEFINITION, CIL_METHOD_DEFINITION and CIL_FIELD_DEFINITION).
Type Specifications

There are two possibilities to specify a type, both of them are captured by the deferred class CIL\_TYPE\_SPECIFICATION, which inherits from CIL\_TOKEN and redefines referent to be of type CIL\_TYPE\_DEFINITION and does not add or change any other feature (note that descendants of CIL\_TYPE\_SPECIFICATION no longer redefine referent). See also the last paragraph and table of [8, section 7.1].

The more verbose construct for specifying types is represented by the deferred class CIL\_TYPE. CIL\_TYPE directly inherits its features from CIL\_TYPE\_SPECIFICATION without additions or changes. Various kinds of type specifications are represented by the descendants of CIL\_TYPE and documented in [8, section 7.1].

In some situations a slightly simpler construct, implemented by the effective class CIL\_TYPE\_REFERENCE, suffices for specifying a type. CIL\_TYPE\_REFERENCE has the attributes type\_name of type CIL\_TYPE\_NAME (see section 4.11 on page 46), an optional module\_name of type CIL\_NAME (see section 4.11 on page 46) and an optional assembly\_name of type CIL\_NAME. CIL\_TYPE\_REFERENCE also provides setters for its attributes, the status report queries is\_module\_name\_set and is\_assembly\_name\_set and a creation routine make. See also [8, section 7.3].

Referencing Local Nodes

Local nodes are referred to by instructions in the method body (see section 4.5 on page 28). The deferred class CIL\_LOCAL\_REFERENCE captures the abstraction of a local reference with the attributes identifier of type CIL\_IDENTIFIER and value of type INTEGER. At least one of these two attributes has to be set at a time. The status report attribute is\_value\_set and query is\_identifier\_set, the setters set\_identifier, set\_value and set\_is\_value\_set and the creation routines make\_with\_identifier and make\_with\_value are also provided. CIL\_LOCAL\_REFERENCE also redefines referent to be of type CIL\_LOCAL (see section 4.9 on the preceding page).

The direct effective descendant CIL\_LABEL\_REFERENCE of CIL\_LOCAL\_REFERENCE redefines referent to be of type CIL\_LABEL (see section 4.5 on page 25) and renames value (including make\_with\_value, set\_value and is\_value\_set) to offset, which offers the alternative method of referencing a label by a byte offset in the instruction stream.

The direct deferred descendant CIL\_VARIABLE\_REFERENCE of CIL\_LOCAL\_REFERENCE redefines referent to be of type CIL\_VARIABLE (see section 4.9 on the preceding page) and renames value (including make\_with\_value, set\_value and is\_value\_set) to index, which offers the alternative method of referencing a variable by its index in the corresponding list of variables.

There are two effective descendants of the class CIL\_VARIABLE\_REFERENCE: CIL\_LOCAL\_VARIABLE\_REFERENCE and CIL\_PARAMETER\_VARIABLE\_REFERENCE. Both redefine the type of referent to their respective counterparts: CIL\_LOCAL\_VARIABLE and CIL\_PARAMETER\_VARIABLE.
4.11 Names, Identifiers and Strings

String Terminals

The abstract notion of a string terminal is implemented in the deferred class \texttt{CIL\_STRING\_TERMINAL}. \texttt{CIL\_STRING\_TERMINAL} provides access to the underlying string by the deferred queries \texttt{encoded\_string} and \texttt{decoded\_string} of type \texttt{STRING}. It also has the secret deferred setters \texttt{set\_encoded\_string} and \texttt{set\_decoded\_string}, the deferred status report queries \texttt{is\_valid\_encoded\_string} and \texttt{is\_valid\_decoded\_string} and the secret deferred queries \texttt{encode} and \texttt{decode}, these are effected by descendants. The underlying string has to be supplied on creation and should not be changed afterwards. See also [8, section 5.2].

Simple Strings

The effective class \texttt{CIL\_SIMPLE\_STRING} inherits from \texttt{CIL\_STRING\_TERMINAL} and joins the pairs of “decoded” and “encoded” features by renaming them as \texttt{string}, \texttt{set\_string} and \texttt{is\_valid\_string}, since simple strings have no encoding. A simple string starts with an alphabetic character, “_”, “$”, “@” or “?” and continues with any number of alphanumeric characters, “_”, “$”, “@” or “?” (see paragraph about \texttt{<ID>} in [8, section 5.2]). \texttt{CIL\_SIMPLE\_STRING} also provides a creation routine \texttt{make}, which allows to set the string on creation.

The effective descendant \texttt{CIL\_SPECIAL\_STRING} of \texttt{CIL\_SIMPLE\_STRING} has no restrictions on the characters it contains and is used, for example, for holding the constructor and initializer name strings \texttt{.ctor} and \texttt{.cctor}.

Note 4.41 \texttt{is\_valid\_string} of \texttt{CIL\_SIMPLE\_STRING} is currently not correctly implemented.

Quoted Strings

The deferred descendant \texttt{CIL\_QUOTED\_STRING} of class \texttt{CIL\_STRING\_TERMINAL} effects its ancestor’s deferred features and adds the secret attributes \texttt{internal\_encoded\_string} and \texttt{internal\_decoded\_string}, one of them is always set, the other is computed and set on demand. The deferred query \texttt{quotation\_character} of type \texttt{CHARACTER} is effected by descendants. \texttt{CIL\_QUOTED\_STRING} also provides the creation routines \texttt{make\_with\_encoded} and \texttt{make\_with\_decoded} for setting the underlying string on creation. See also the paragraphs in [8, section 5.2] on \texttt{<QSTRING>} and \texttt{<SQSTRING>}.

The class \texttt{CIL\_QUOTED\_STRING} has two effective descendants: \texttt{CIL\_DOUBLY\_QUOTED\_STRING} and \texttt{CIL\_SINGLY\_QUOTED\_STRING}, which effect \texttt{quotation\_character} as a constant attribute with character: ‘”’ and ‘’’ respectively.

Note 4.42 The features \texttt{is\_valid\_encoded\_string}, \texttt{is\_valid\_decoded\_string}, \texttt{encode} and \texttt{decoded} are currently not correctly implemented.
Note 4.43 The commands *append_encoded_string* and *append_decoded_string* violate the requirement that the underlying string mustn’t change after creation. These commands are currently in use by the parser for performance and simplicity reasons.

**Identifiers**

Identifiers are implemented by the class CIL_IDENTIFIER, which has an attribute *string_terminal* of type CIL_STRING_TERMINAL and a corresponding creation routine *make* for setting the string terminal on creation. The string terminal should not be changed after the identifiers creation. *string_terminal* should only be attached to instances of type CIL_SIMPLE_STRING or CIL_SINGLY_QUOTED_STRING. See also [8, section 5.3].

**Composite Identifiers**

Composite identifiers are implemented by the deferred generic class CIL_COMPOSITE_IDENTIFIER [G]. CIL_COMPOSITE_IDENTIFIER has an attribute *components* of type LIST [G], a deferred query *separator* of type CHARACTER and creation routines *make*, *make_with_capacity* and *make_with_component*.

The class CIL_NAME is a direct descendant of CIL_COMPOSITE_IDENTIFIER [CIL_IDENTIFIER] and effects *separator* as a constant attribute with character `. `. See also [8, section 5.3] on `<dottedname>`.

The class CIL_TYPE_NAME is a direct descendant of CIL_COMPOSITE_IDENTIFIER [CIL_NAME] and effects *separator* to a constant attribute with character `'/` '. See also [8, section 7.3] on the grammar fragment `<dottedname> [ / <dottedname>]*.`

**4.12 Factory**

The creation of the AST is accomplished by an Abstract Factory pattern with the deferred class CIL_AST_FACTORY providing deferred factory routines.

The effective descendant CIL_DEFAULT_AST_FACTORY of CIL_AST_FACTORY effects all inherited factory routines. It is important to note that some of the factory routines are *once* routines and therefore implement a Flyweight pattern. This is especially important in the case of the instructions inheriting from CIL_NONE_INSTRUCTION, which are all implemented as Flyweights. See also section 4.13 on the following page about the *context* attribute of type CIL_METHOD_BODY_ITEMS_CONTEXT, which makes the index of the current method body item available during a traversal (the *extrinsic* information of a Flyweight method body item, see [13]).

CIL_DEFAULT_AST_FACTORY also provides debugging clauses for short and verbose debugging output on the console.
4.13 Visitors

Entity Visitors

The deferred class CIL_ENTITY_VISITOR provides a deferred visit command for each effective descendant of CIL_ENTITY and the apply command, which takes an argument of type CIL_ENTITY on which the Visitor should be applied. For example, apply can be used to create an agent to be passed to do_all of class LINEAR.

CILEMPTY_ENTITY_OR_TYPE_MEMBER_VISITOR is currently the only direct descendant of CIL_ENTITY_VISITOR (see below).

Type Member Visitors

The deferred class CIL_TYPE_MEMBER_VISITOR provides a deferred visit command for each effective descendant of CIL_TYPE_MEMBER and the apply command, which takes an argument of type CIL_TYPE_MEMBER on which the Visitor should be applied. For example, apply can be used to create an agent to be passed to do_all of class LINEAR.

CILEMPTY_ENTITY_OR_TYPE_MEMBER_VISITOR is currently the only direct descendant of CIL_TYPE_MEMBER_VISITOR (see below).

Entity and Type Member Visitors

The effective class CILEMPTY_ENTITY_OR_TYPE_MEMBER_VISITOR inherits from CIL_ENTITY_VISITOR and CIL_TYPE_MEMBER_VISITOR and effects all inherited visit commands. The implementation of each visit command includes debugging clauses for short and verbose output but is otherwise empty. This class allows its descendants to redefine only the necessary routines instead of effecting all of them. Additionally the command apply from CIL_ENTITY_VISITOR is renamed as apply_to_entity and apply from CIL_TYPE_MEMBER_VISITOR is renamed as apply_to_type_member.

The descendant CILMETHOD_FILTER_VISITOR of class CILEMPTY_ENTITY_OR_TYPE_MEMBER_VISITOR has an attribute method_agent of type PROCEDURE [ANY, TUPLE [CIL_METHOD_DEFINITION]] with a corresponding setter and creation routine make. The method agent is called when a method (including constructors and initializers) is visited. The Visitor applies itself to the members of a type definition if one is visited.

See also section 4.14 on the next page on the descendant CIL_ENTITY_OR_TYPE_MEMBER_ANNOTATION_VISITOR of class CILEMPTY_ENTITY_OR_TYPE_MEMBER_VISITOR.

Method Body Item Visitors

The deferred class CIL_METHOD_BODY_ITEM_VISITOR provides a deferred visit command for each effective descendant of CIL_METHOD_BODY_ITEM. The Visitor has a secret attribute context of type CIL_METHOD_BODY_ITEMS_CONTEXT with the item of its body_items set to the currently visited method body item during iteration.
with apply_all and apply_all_reverse. CIL_METHOD_BODY_ITEM_VISITOR also implements the apply command, which takes an argument of type CIL_TYPE_MEMBER on which the Visitor should be applied and the apply_all and apply_all_reverse commands, which take an argument of type like context on whose body_items they iteratively invoke the apply command.

The apply_all and apply_all_reverse commands call the commands start_traversal at the begin of the traversal, start_visit before and finish_visit after each call to apply and finish_traversal at the end of the traversal. start_traversal, start_visit, finish_visit and finish_traversal are empty and are redefined by descendants for advanced uses (see, e.g., section 5 on the following page on CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR).

CIL_METHOD_BODY_ITEM_VISITOR also provides a status report attribute is_stopped and the corresponding setter set_is_stopped for early termination of a traversal by apply_all and apply_all_reverse.

The effective class CIL EMPTY_METHOD_BODY_ITEM_VISITOR inherits from CIL_METHOD_BODY_ITEM_VISITOR and effects all inherited visit commands. The implementation of each visit command includes debugging clauses for short and verbose output but is otherwise empty. This class allows its descendants to redefine only the necessary routines instead of effecting all of them.

The effective class CIL_DEFAULT_METHOD_BODY_ITEM_VISITOR inherits from CIL_METHOD_BODY_ITEM_VISITOR and effects all inherited visit commands. The implementation of each visit command includes debugging clauses for short and verbose output and calls the deferred command visit_method_body_item with the visited body item as argument. This class allows its descendants to redefine routines corresponding to specific body items and to provide a default implementation for all others.

4.14 Annotation

The annotation algorithms make use of the Visitors (see section 4.13 on the page before) for processing a newly parsed AST.

CIL_ENTITY_OR_TYPE_MEMBER_ANNOTATION_VISITOR inherits from CIL EMPTY_ENTITY_OR_TYPE_MEMBER_VISITOR and annotates all methods (including constructors and initializers) by the application of a list of method body item Visitors attached to the attribute method_body_annotation_visitors. It recursively applies itself to the members of a visited type definition.

CIL_BODY_ITEM_ANNOTATION_VISITOR inherits from CIL EMPTY_METHOD_BODY_ITEM_VISITOR and resolves the references of descendants of CIL VARIABLEREFERENCE_INSTRUCTION. It also extends the label_table of the enclosing method with the label identifier as keys and the index in the method body as values.

Note 4.44 CIL_SCOPE_BLOCK and CIL EXCEPTION_BLOCK are currently not supported by the method body annotation Visitor.
Chapter 5

Contract Extraction

Various algorithms participate in the contract extraction but currently only precondition extraction is supported. The primary assumption in the precondition extraction is that an explicit exception in a method corresponds to an incorrect use of the method, meaning that the conditions under which the exception is thrown should be part of a method precondition, which the client has to satisfy.

The analysis of a method, for which preconditions should be extracted, reveals the method implementation’s code paths that finish at an explicit exception and whose instructions are tractable by the succeeding algorithms. A symbolic execution algorithm then extracts the branch conditions on all such code paths. Finally a precondition extraction algorithm forms precondition clauses by examining each considered code path and combining the branch conditions that would lead to an actual execution of the code to the explicitly thrown exception.

The next section provides a high-level description with pseudocode of the algorithms (section 5.1). The succeeding sections then document classes for directed graphs (section 5.2 on page 71) and expressions (section 5.3 on page 74), which are both used by the algorithms that follow. The analysis of the method’s body starts with partitioning the sequential list of method body items into basic blocks where the flow of execution within a basic block is sequential (section 5.4 on page 84). Each basic block is then classified according to the side-effects of the instructions it contains, currently only side-effects on the evaluation stack can be addressed unless the basic block ends with an exception (section 5.5 on page 88). Branch conditions of the addressable basic blocks are then extracted into an expression by a symbolic execution algorithm (section 5.7 on page 91) and combined to form precondition clauses of the method (section 5.8 on page 94). The last section presents and discusses the results for the classes ArrayList, Stack and Queue from the .NET libraries [16] (section 5.9 on page 96).

5.1 High-Level Description

The section provides a high-level description with pseudocode of the algorithms for precondition extraction. The description and the pseudocode enable verifying the chosen approach in the absence of implementation details. Listings
Basic Block Analysis

The basic block analysis algorithm partitions the method body into basic blocks. A basic block is a sequence of method body items where any execution is sequential. Only the first body item of a basic block can be a branch target and only the last body item can be a branch instruction or an instruction that causes the execution to leave the method. The result of the algorithm is a basic block graph where a vertex is a basic block and an edge is a possible transition during execution, called basic block link in what follows.

A basic block graph is represented by several classes. The listings 5.1, 5.2 on the next page, 5.3 on the following page and 5.4 on page 52 provide an overview of their attributes. Basic blocks and basic block links are each split into two classes. The classes BASIC_BLOCK and BASIC_BLOCK_LINK are concerned with the connections between basic blocks and basic block links, while the classes BASIC_BLOCK_CONTENT and BASIC_BLOCK_LINK_CONTENT represent the content of basic blocks and basic block links. The purpose of each attribute is explained later.

The basic block analysis proceeds in two passes over the method’s body items. The result of the first pass (see listings 5.5 on page 53 and 5.6 on page 54) is a mapping from indices of the list of method body items to basic blocks that start at the corresponding index. The second pass (see listings 5.7 on page 56 and 5.8 on page 57) uses this mapping to fill the basic blocks with the corresponding method body items and connecting the basic blocks with the appropriate basic block links.

```plaintext
class BASIC_BLOCK

feature -- Access

  predecessors: LIST [BASIC_BLOCK_LINK]  -- Preceding links

  successors: LIST [BASIC_BLOCK_LINK]  -- Succeeding links

  item: BASIC_BLOCK_CONTENT  -- Content

end

Listing 5.1: Basic Block
```
Listing 5.2: Basic Block Content

Listing 5.3: Basic Block Link

First Pass

Listing 5.5 on page 53 shows part of the first pass. The attribute `body_items` is set to the considered method’s list of body items. The algorithm in the command `basic_blocks_first_pass` iterates over the list of method body items and introduces basic blocks at their start indices (with the command `introduce_basic_block`, see listing 5.6 on page 54) as appropriate for each body item. If the item is a (conditional or unconditional) branch instruction, after which the execution continues at the next body item or at the branch instruction’s target body item, basic blocks starting at the next and at the target body items are introduced. In
the case of an instruction at which the execution leaves the method (e.g., `ret`, `throw`, . . .) a basic block starting at the next body item is inserted. Similarly, when a switch instruction is encountered basic blocks at the next and at all target body items are introduced.

The implementation of `introduce_basic_block` taking the start index of the inserted basic block as its argument is shown in listing 5.6 on page 54. The listing also shows the attribute `basic_block_table`, which is initially set to an empty hash table and extended by the command `introduce_basic_block` with mappings from start indices to basic blocks. In the implementation of `introduce_basic_block` the argument `a_start_index` is first checked for its validity and whether no basic block has been inserted at this start index. Only then is a new basic block created and `basic_block_table` extended with the key `a_start_index` and the new basic block as the value. The check avoids the introduction of a basic block after the last body item (e.g., after a final `ret` instruction) and the insertion of multiple basic blocks for the same index (e.g., when multiple branch instructions target the same body item).

**Note 5.1** The classes for basic block graphs in the actual implementation are descendants of the generic classes for graphs (CIL_VERTEX, a descendant of CILGENERALMULTILINKABLE which is a descendant of CELL of the EiffelBase library and CILEDGE, which is also a descendant of CELL). In the implementation, method body items do not have the queries `is_branch`, `is_leaving_method` and `is_switch`. The functionality is instead implemented by CIL_BASIC_BLOCK_ANALYSIS_VISITOR and the actions in the if-clauses in `basic_blocks_first_pass` are implemented by its descendant CIL_FIRST_BASIC_BLOCK_ANALYSIS_VISITOR (see section 5.4 on page 85). The attributes `target` on branch instructions and `targets` on switch instructions are called `reference` and `references` in the actual implementation and there is no `index` query on references, the actual implementation uses the attribute `label_table` from CIL_METHOD_DEFINITION for looking up a target’s index in the list of method body items (see section 4.4 on page 23).
\texttt{body\_items}: \text{LIST [METHOD\_BODY\_ITEM]}
-- Body of the analyzed method
-- Initialized by the client

\texttt{basic\_blocks\_first\_pass is}
-- First pass of the basic block analysis
-- Find start indices of basic blocks
\textbf{do}
-- Introduce basic block at index 1
\texttt{introduce\_basic\_block (1)}

-- Iterate over method body items and
-- introduce basic blocks when discovered
\texttt{from}
\texttt{body\_items.start}
\textbf{until}
\texttt{body\_items.after}
\textbf{loop}
\textbf{if} body\_items.item.is\_branch \textbf{then}
-- For branch instructions:
-- Introduce basic blocks at the next and
-- at the target instruction
\texttt{introduce\_basic\_block (body\_items.index + 1)}
\texttt{introduce\_basic\_block (body\_items.item.target.index)}
\textbf{elseif} body\_items.item.is\_leaving\_method \textbf{then}
-- For instructions leaving the method:
-- Introduce basic block at the next instruction
\texttt{introduce\_basic\_block (body\_items.index + 1)}
\textbf{elseif} body\_items.item.is\_switch \textbf{then}
-- For switch instructions:
-- Introduce basic blocks at the next and
-- at each target instruction
\texttt{introduce\_basic\_block (body\_items.index + 1)}
\texttt{from}
\texttt{body\_items.item.targets.start}
\textbf{until}
\texttt{body\_items.item.targets.after}
\texttt{do}
\texttt{introduce\_basic\_block (body\_items.item.targets.item.index)}
\texttt{body\_items.item.targets.forth}
\texttt{end}
\texttt{end}
\texttt{body\_items.forth}
\texttt{end}

Listing 5.5: Basic Blocks Analysis - First Pass (Part 1 of 2)
**Second Pass**

The first part of the second pass of the basic block analysis is shown in listing 5.7 on page 56. The algorithm in the command `basic_block_second_pass` operates on the considered method’s list of body items `body_items` and the mapping from start indices to basic blocks `basic_block_table` computed by the first pass. The attribute `basic_blocks` is initially set to an empty list and extended by the algorithm with basic blocks in the order of their appearance in the method’s body.

The algorithm in `basic_block_second_pass` iterates over the method body items in `body_items`. In each iteration step the algorithm first updates the locals `a_block` to be attached to the basic block containing the current body item and `a_next_block` to be attached to the basic block starting at the next body item, if there is such a basic block. `a_next_block` can therefore be `void`. Whenever `a_block` is attached to a different basic block `basic_blocks` is extended with that basic block. The remaining part of the iteration step extends the content of `a_block` with the current body item and, if the current body item is the last of the current basic block, computes the links starting at `a_block` (with `compute_links`, see listing 5.8 on page 57).

The command `compute_links` in listing 5.8 on page 57 receives the current basic block `a_block` and the next basic block `a_next_block` as its arguments and introduces the basic block links (with the command `link_basic_blocks`, see below) originating at `a_block`. The argument `a_next_block` is `void` when `a_block` is the method’s last basic block. The computation of the links depends on the last body item of the current basic block. If the last item is a conditional branch instruction, links from the current to the next and the target body item’s basic blocks are introduced.
block are introduced. In the case of an unconditional branch instruction only a
basic block link to the target body item’s basic block is added to the basic block
graph. If the last instruction of the current basic block lets the execution leave
the method no link is introduced. For switch instructions links from the current
to the next and each target body item’s basic block are added. All other body
items are handled by connecting the current with the next basic block with a
basic block link.

The command `link_basic_blocks` receives an origin and a target basic block.
It creates a basic block link with the given origin and target basic blocks and
extends the list of succeeding and preceding links of the origin and target re-
spectively with the newly created link. The creation routine of the basic block
link takes an expression as its third argument that has been left out. In the
actual implementation the expression passed at a link’s creation is the condition
under which the link will be followed during execution. However, at this point
of the precondition extraction that condition is not completely known yet and
the implementation makes use of a placeholder for the part of the expression
that is not known (using a delayed expression, see 5.3 on page 79). For example,
the condition of a link, which is followed if a branch condition does not hold, is
“not *”, where ‘*’ is the still unknown branch condition. Branch conditions are
extracted by the symbolic execution algorithm.

Note 5.2 Similar to the first pass: In the actual implementation method body
items do not have the queries `is_conditional_branch`, `is_unconditional_branch`,
`is_leaving_method` and `is_switch`. The functionality is instead imple-
mented by CIL_BASIC_BLOCK_ANALYSIS_VISITOR and the actions in
the if-clauses in `compute_links` are implemented by the its descendant
CIL_SECOND_BASIC_BLOCK_ANALYSIS_VISITOR (see section 5.4 on
page 85). The attributes `target` on branch and `targets` on switch instructions
are called `reference` and `references` in the actual implementation and there is
no index query on references, the actual implementation uses the attribute
`label_table` from CIL_METHOD_DEFINITION for looking up a target’s index
in the list of method body items (see section 4.4 on page 23).
\[body\_items\]: LIST [METHOD\_BODY\_ITEM]
-- Body of the considered method
-- Initialized by the client

\[basic\_block\_table\]: HASH\_TABLE [BASIC\_BLOCK, INTEGER]
-- Basic block table mapping an index of a body item
to the basic block starting at this item
-- Result of the first pass of the basic block analysis

\[basic\_blocks\]: LIST [BASIC\_BLOCK]
-- Basic blocks in the order of their appearance in the method
-- Initialized to an empty list

\[basic\_blocks\_second\_pass\] is
-- Second pass of the basic block analysis
-- Fill each basic block's content with the body items of
-- that basic block and connect the basic blocks with links
-- representing the control flow

local
\[a\_block\]: BASIC\_BLOCK
-- Current basic block
\[a\_next\_block\]: BASIC\_BLOCK
-- Basic block starting at the next body item, if any

do
-- Set 'a\_next\_block' to the first basic block
\[a\_next\_block\] := basic\_block\_table\_item (1)

from
\[body\_items\].start
until
\[body\_items\].after

loop
-- If a basic block starts at the current body item
if \[a\_next\_block\] /= void then
-- Update 'a\_next\_block' and extend 'basic\_blocks'
-- with the new basic block
\[a\_block\] := \[a\_next\_block\]
basic\_blocks\_extend (\[a\_block\])
end
-- Update 'a\_next\_block' to the basic block
-- starting at the next body item, if any
\[a\_next\_block\] := basic\_block\_table\_item (\[body\_items\].index + 1)

-- Extend 'body\_items' of the current basic block
-- with the current body item
\[a\_block\].item.body\_items\_extend (\[body\_items\].item)

-- If the iteration is at the end of 'a\_block'
if \[a\_next\_block\] /= void or else \[body\_items\].is\_last then
-- Compute the links originating at the current basic block
compute\_links (\[a\_block\], \[a\_next\_block\])
end

end

\[body\_items\].forth
end

Listing 5.7: Basic Blocks Analysis - Second Pass (Part 1 of 2)
compute_links (a_block: BASIC_BLOCK; a_next_block: BASIC_BLOCK) is
  -- Compute links from the current 'a_block' to other basic
  -- blocks, where 'a_next_block' starts at the next body item
  do
    if body_items.item.is_conditional_branch then
      -- For conditional branch instructions:
      -- Link the current with the next and the target basic blocks
      link_basic_blocks (a_block, a_next_block)
      link_basic_blocks (a_block, basic_block_table.item (body_items.item.target.index))
    elseif body_items.item.is_unconditional_branch then
      -- For unconditional branch instructions:
      -- Link the current with the target basic block
      link_basic_blocks (a_block, basic_block_table.item (body_items.item.target.index))
    elseif body_items.item.is_leaving_method then
      -- For instructions leaving the method: Do nothing
    elseif body_items.item.is_switch then
      -- For switch instructions:
      -- Link the current with the next and the target basic blocks
      link_basic_blocks (a_block, a_next_block)
      from
      body_items.item.targets.start
      until
      body_items.item.targets.after
      loop
        link_basic_blocks (a_block, basic_block_table.item (body_items.item.targets.item.index))
        body_items.item.targets.forth
      end
    else
      -- For all other body items:
      -- Link the current with the next basic block
      link_basic_blocks (a_block, a_next_block)
    end
  end
link_basic_blocks (an_origin: BASIC_BLOCK; a_target: BASIC_BLOCK) is
  -- Link 'an_origin' to 'a_target'
  local
    a_link: BASIC_BLOCK_LINK
  do
    create a_link.make (an_origin, a_target, ...)
    an_origin.successors.extend (a_link)
    a_target.predecessors.extend (a_link)
  end

Listing 5.8: Basic Blocks Analysis - Second Pass (Part 2 of 2)
Basic Block Classification

The basic block classification algorithm assigns a category to each basic block depending on the body items it contains. The classification follows a total order: “unknown” < “normal” < “difficult” < “intractable” < “exception”. Later algorithms will ignore code paths containing “difficult” or “intractable” basic blocks because they contain instructions whose effects are not tractable by the current implementation.

The algorithm in the command classify of listing 5.9 iterates over the body items of the considered basic block’s content and updates (with the command update_classification) at each step the basic block’s classification with the current body item’s classification. The command update_classification receives the current body item’s classification as argument and replaces the basic block’s classification with the received classification, if the received one is considered greater (or stronger) than the basic block’s current classification. See also table 5.7 on page 89 for the classifications associated with body items.

Note 5.3 In the actual implementation method body items do not have a query classification, the body items are associated with classifications by means of a visitor: CIL_BASIC_BLOCK_CLASSIFICATION_VISITOR.

```c
content: BASIC_BLOCK_CONTENT
   -- Considered basic block’s content
   -- Initialized by the client

classify is
   -- Classify ‘content’.
   local
      some_items: LIST [METHOD_BODY_ITEM]
   do
      some_items := content.body_items
      from
      until
      loop
         update_classification (some_items.item.classification)
         some_items.forth
      end
   end

update_classification (a_classification: BASIC_BLOCK_CLASSIFICATION) is
   -- Set ‘classification’ of ‘content’ to ‘a_classification’,
   -- if ‘a_classification’ is ‘greater than’ the current.
   do
      if content.classification < a_classification then
         content.set_classification (a_classification)
      end
   end
```

Listing 5.9: Basic Blocks Classification
Code Path Traversal

The algorithm for code path traversal is used as part of the later described code path extraction and precondition extraction algorithms. The traversal starts at the first basic block of a method and traverses all paths in the basic block graph. The client algorithms can exclude specific basic blocks from the traversal and the traversal algorithm will ignore paths that include such basic blocks. The traversal algorithm allows the client algorithms to incrementally compute results on the traversed paths in depth-first order. That is, the algorithm will ask its clients for the result associated with a basic block given the results from the succeeding basic blocks in all considered paths. The algorithm also detects loops in the graph and does not follow links that would create a circle in the traversed path.

The algorithm in the recursively invoked query traversal_result of listing 5.10 on the next page traverses the paths originating at the argument a_block and returns the result of the traversal of generic type G. The attribute path keeps track of the currently considered part of a path and is used for loop detection. The code in traversal_result first queries the client algorithm with is_traversable, supplying the current basic block’s a_block content as argument, if that basic block should be traversed. If the current basic block is not traversable the result is void, otherwise the algorithm recursively invokes itself on each succeeding basic block that does not introduce a loop in the currently considered path. The results from the recursive invocations on the succeeding basic blocks and the corresponding basic block link contents are collected in the lists some_results and some_link_items respectively.

The current basic block is considered to be the end of a path if all succeeding basic blocks would introduce a loop. After the iteration over all succeeding basic blocks the local variable a_is_end is true if the current basic block is the end of a path. To that end, a_is_end is set to true before the iteration and set back to false during the iteration, if a succeeding basic block not introducing a loop in the current path is found.

The current basic block lies on at least one path containing only traversable (as specified by the client algorithm) basic blocks if it is traversable itself and if it is the end of a path or if at least one of the recursive invocations of the algorithm returned a result. If the current basic block lies on at least one traversable path the client algorithm is queried with derived_result for the (assumed to be not void) result associated with the current basic block. The query derived_result receives the content of the current basic block, the results from the recursion and the corresponding basic block link contents as its arguments.

A client algorithm of the code path traversal algorithm has to specify the actual generic parameter for the formal generic parameter G of the algorithm’s result and implement the queries is_traversable and derived_result.
path: STACK [BASIC_BLOCK]
-- Currently examined blocks (for loop detection)
-- Initialized to an empty stack

traversal_result (a_block: BASIC_BLOCK): G is
-- Result from traversal of paths starting at 'a_block'
local
a_is_end: BOOLEAN
a_next_block: BASIC_BLOCK
a_next_result: G
some_results: LIST [G]
some_link_items: LIST [BASIC_BLOCK_LINKCONTENT]
do
-- Examine 'a_block', if it is traversable
if is_traversable (a_block.item) then
  -- Extend the currently examined basic blocks
  path.extend (a_block)
  -- Create temporary lists
  create some_results.make (a_block.successors.count)
  create some_link_items.make (a_block.successors.count)
  -- Assume 'a_block' is the end of a path
  a_is_end := true
  -- Iterate over the succeeding basic block links
  from
    a_block.successors.start
  until
    a_block.successors.after
  loop
    -- Retrieve the succeeding basic block
    a_next_block := a_block.successors.item.target
    -- If the succeeding block does not introduce a circle
    if not path.has (a_next_block) then
      -- 'a_block' is not the end of a path
      a_is_end := false
      -- Recursively compute result from 'a_next_block'
      a_next_result := traversal_result (a_next_block)
      -- If a result was found
      if a_next_result /= void then
        -- Extend the temporary results and links
        some_results.extend (a_next_result)
        some_link_items.extend (a_block.successors.item.item)
      end
    end
  end
a_block.successors.forth
end
-- If 'a_block' is the end of or in a traversable path
if a_is_end or else not some_results.is.empty then
  -- Compute result for paths starting at 'a_block'
  Result := derived_result (a_block.item,
    some_results, some_link_items)
end
-- Remove 'a_block' from the currently examined basic blocks
path.remove
end

Listing 5.10: Code Path Traversal
Note 5.4 The actual implementation encapsulates the queries \texttt{is\_traversable} and \texttt{derived\_result} in a Strategy pattern (\texttt{CIL\_PATH\_TRAVERSAL\_STRATEGY} and its descendents). The implementation of the algorithm (in the deferred class \texttt{CIL\_PATH\_TRAVERSAL}) also allows for a traversal in the opposite direction (by the descendant \texttt{CIL\_FORWARD\_PATH\_TRAVERSAL} as opposed to the described \texttt{CIL\_BACKWARD\_PATH\_TRAVERSAL}), where all paths finishing at a given basic block are traversed and the results are computed starting at the first basic block of each path. The implemented algorithm also stores the computed result of each basic block and reuses it if the same basic block is reached by other paths. The implementation relies on the more general classes \texttt{CIL\_VERTEX} and \texttt{CIL\_EDGE} instead of only basic blocks and basic block links.

**Code Path Extraction**

The code path extraction algorithm creates a subgraph of the original basic block graph. The subgraph contains only code paths with basic blocks classified as “normal” (or weaker) or “exception” (or stronger), code paths in the original graph containing “difficult” or “intractable” basic blocks are omitted in the subgraph.

The algorithm uses the code path traversal algorithm with the actual generic parameter \texttt{BASIC\_BLOCK} for the formal generic parameter \texttt{G} of the traversal’s result. The implementation of the queries \texttt{is\_traversable} and \texttt{derived\_result} is shown in listing 5.11 on the following page. The query \texttt{is\_traversable} returns \texttt{true} only if the argument basic block is classified as “normal” (or weaker) or “exception” (or stronger).

The query \texttt{derived\_result} receives the currently traversed basic block’s content \texttt{a\_block\_item}, a list of the results from the already traversed succeeding basic blocks \texttt{some\_results} (the results are basic blocks themselves) and a list of corresponding basic block link contents \texttt{some\_link\_items}. The routine \texttt{derived\_result} first creates its result, which is a new basic block with the received basic block content, and then iterates over the succeeding links’ contents and the results. Each iteration step connects the newly created basic block by a new link with the current link content to the corresponding result.

The repeated invocation of \texttt{derived\_result} by the code path traversal algorithm therefore builds a new basic block graph with the same basic block contents and basic block link contents as the original graph. The traversal algorithm’s \texttt{traversal\_result} returns, when invoked on the first basic block in the method, the first basic block in the method from the newly created subgraph. The result is \texttt{void} only if all code paths include a “difficult” or “intractable” basic block.

Note 5.5 The implementation of the algorithm in \texttt{CIL\_CODE\_PATH\_EXTRACTOR} uses an older interface for the traversal strategy. This interface is provided by its ancestor \texttt{CIL\_DEFAULT\_PATH\_TRAVERSAL\_STRATEGY} which inherits and effects the query \texttt{derived\_result} using the older interface.
is_traversable (a_block_item: BASIC_BLOCK_CONTENT): BOOLEAN is
    -- Should the traversal proceed with 'a_block_item'?
    do
        -- Traverse 'normal' and 'exception' basic blocks only
        Result := a_block_item.\_classification <= normal\_classification
        or else
        a_block_item.\_classification >= exception\_classification
    end

derived_result (a_block_item: BASIC_BLOCK_CONTENT;
    some_results: LIST [BASIC_BLOCK];
    some_link_items: LIST [BASIC_BLOCK\_LINK\_CONTENT]
): BASIC\_BLOCK is
    -- Result of 'a_block_item' derived from recursively computed
    -- 'some_results' and the corresponding 'some_link_items'
    require
        equal_counts: some_results.\_count = some_link_items.\_count
    local
        a_new_link: BASIC\_BLOCK\_LINK
    do
        -- Create a 'new' basic block with the current content
        create Result.make (a_block_item)
        -- Iterate over succeeding 'new' basic blocks
        from
            some_results.start
            some_link_items.start
        until
            some_results.after
        do
            -- Link the current 'new' basic block with the
            -- succeeding 'new' basic block
            create a_new_link.make (Result, some_results.item,
                some_link_items.item)
            Result.successors.extend (a_new_link)
            some_results.item.predecessors.extend (a_new_link)
            some_results.forth
            some_link_items.forth
        end
    end
end

Listing 5.11: Code Path Extraction

Symbolic Execution

The subgraph from the code path extraction is input to the algorithm for sym-
bolic execution. The symbolic execution algorithm is applied to each basic block
which finishes with a conditional branch or a switch instruction. The algorithm
extracts an expression for the branch condition or the integer value in the case
of switch instructions and assigns the extracted expression to the basic block
under consideration. The current implementation does not support all instruc-
tions that can occur in basic blocks classified as “normal”. A corresponding extension of the algorithm itself is straightforward but would also involve the development additional expression classes.

The first part of the algorithm in the command `extract` of listing 5.12 on the next page iterates backwards over the body items of the considered basic block’s content until the iteration has gone before the first body item or the attribute `isStopped` is set to `true`. `isStopped` is set to `true` when the extraction of the basic block’s expression has finished or when the algorithm encounters a body item it does not support. In each iteration step the command `execute` of listing 5.13 on page 65 and the command `finishExecute` of listing 5.14 on page 67 are called.

The iteration and therefore the execution of the body items are carried out backwards. However, the implementation of `execute` pretends it is carried out forwards. The implementation of `finishExecute` does part of the necessary work to make this setting work. This allows the implementation of `execute` to be simpler than it would be if it had to implement the execution backwards and it offers the possibility to reuse part of the algorithm for forward execution.

Listing 5.13 on page 65 shows the implementation of `execute`, which receives the current body item `a_body_item` as its argument. The command `execute` accesses the execution stack with the routines `item`, `remove` and `extend` shown in listings 5.14 on page 67 and 5.15 on page 68. These routines have to take into account that `execute` is implemented as if the execution would proceed forwards in the basic block’s body items, not backwards. In the current implementation there are three restrictions for `execute` during a single invocation: it can make at most one call to `extend`, it has to make its call to `extend`, if any, only after all calls to `item` and `remove` and it has to `remove` all stack items it looks up with `item`. The implementation of these routines is discussed below.

The listing of `execute` shows, as an example, the relevant code for the three instructions `add`, `bgt` (branch on greater than) and `ldarg` (load argument). If `a_body_item` is an `add` instruction the first two items are read from the stack and combined to an “add” expression (with the Factory Query `infix “+” from EXPRESSION, see section 5.3 on page 74), which is then put on the stack. Similar actions are carried out for a `bgt` instruction, where the resulting expression becomes associated with the considered basic block content. If the symbolic execution of a body item is not implemented the attribute `isStopped` is set to `true` and the iteration over the body items will stop without having extracted a complete expression.

**Note 5.6** The class `CIL_SYMBOLIC_EXTRACTOR` in the actual implementation encapsulates the algorithm. The if-clauses for the method body items in `execute` are implemented by the deferred class `CIL_SYMBOLIC_EXECUTION_VISITOR`. 
Listing 5.12: Symbolic Execution (Part 1 of 4)
execute (a_body_item: METHOD_BODY_ITEM) is
   -- Symbolically execute ‘a_body_item’
   -- Implemented as if execution would proceed forwards
   local
   an_expression, an_item: EXPRESSION
   do
     if a_body_item.is_add then
       -- For ‘add’ instructions:
       -- Retrieve and remove the first item from the stack
       an_item := item remove
       -- Retrieve and remove the second item and
       -- form an ‘add’ expression with both items
       an_expression := item + an_item remove
       -- Extend the stack with the ‘add’ expression
       extend (an_expression)
     elseif a_body_item.is_bgt then
       -- For ‘bgt’ (branch on greater than) instructions:
       -- Retrieve and remove the first item from the stack
       an_item := item remove
       -- Retrieve and remove the second item and form
       -- a ‘greater than’ expression with both items
       an_expression := item > an_item remove
       -- Set the expression of the content to the
       -- ‘greater than’ expression
       content.set_expression (an_expression)
     elseif a_body_item.is_ldarg then
       -- For ‘ldarg’ (load argument) instructions:
       -- Extend the stack with a ‘variable’ expression
       extend (create {VARIABLE_EXPRESSION}.make (a_body_item.identifier))
     else
       -- For unimplemented body items:
       -- Stop the traversal
       is_stopped := true
     end
   end
end

Listing 5.13: Symbolic Execution (Part 2 of 4)

Execution Stack Access Routines

The attribute unknown_stack of listing 5.14 on page 67 is attached to a stack which contains all unknown expressions whose existence is known before a call to execute and after a call to finish.execute. An unknown expression is represented by an identity expression (see section 5.3 on page 79) whose operand is eventually set to an actual expression.

The attributes unknown_item and unknown_items are used during a call to
execute by item and remove and by finish_execute. unknown_items is the list of items that have been read from the stack during a single call to execute. unknown_item is the top of the stack after an invocation of item and before the corresponding call to remove, it is void otherwise.

The command extend can be called at most once during a single invocation of execute, at this point the attribute unknown_items contains all items drawn from the stack during the current call to execute and the last item read from the stack has been removed, that is, unknown_item is void. The unknown_stack still contains the unknown expressions whose existence was known after (because the execution is done backwards) the current body item. Therefore the argument expression passed to extend matches the current item on top of unknown_stack and this item will not be on the stack before the current body item. extend first sets the operand on the unknown_stack’s item and then removes it from the stack.

When finish_execute is called, the attribute unknown_items contains all items drawn from the stack during the call to execute, the last item read from the stack has been removed, that is, unknown_item is void and the optional call to extend has removed the item from the unknown_stack that was there after but not before the current body item is executed. Conversely the unknown_items were not on the unknown_stack after the current body item but they will be there before its execution. finish_execute appends unknown_items to unknown_stack in the reverse order and then wipes out unknown_items. finish_execute then checks if there are any remaining unknown expressions on the stack and if the stack is empty is_stopped is set to true; in this case the extraction has succeeded.

When the routines item and remove in listing 5.15 on page 68 are invoked, the attribute unknown_item is attached to an unknown expression if item has been called before without a corresponding call to remove, it is void otherwise. item and remove both call create_item, which creates a new unknown_item and extends unknown_items, if unknown_item is void. The query item then sets its result to unknown_item and remove sets unknown_item to void. A close look at create_item shows that what has been called “unknown expression” so far would more accurately be called “identity expression with an unknown expression as its operand”. The operand of the identity expression will, as described above, eventually be replaced with an actual expression by a call to extend.

Note 5.7 The routines for accessing the execution stack are implemented in the class CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR, which is a descendant of CIL_SYMBOLIC_EXECUTION_VISITOR. The command finish_execute is thereby named finish_visit.
unknown_stack: STACK [IDENTITY_EXPRESSION]
   -- Execution stack, contains all unknown expressions
   -- of the current extraction before an 'execute' and
   -- after a 'finish_execute'
   -- Initialized to an empty stack

unknown_item: IDENTITY_EXPRESSION
   -- Unknown expression on top of stack during an 'execute'
   -- (accessed/set through 'item', set to 'void' by 'remove')

unknown_items: LIST [IDENTITY_EXPRESSION]
   -- List of new unknown expressions during an 'execute'
   -- (extended by 'item', wiped out by 'finish_execute')
   -- Initialized to an empty list

extend (an_expression: EXPRESSION) is
   -- Extend stack with 'an_expression'
   require
   retrieved_items_removed: unknown_item = void
   do
      if not unknown_stack. is_empty then
         -- Set the operand of the identity expression
         unknown_stack.item.set_operand (an_expression)
         -- Remove the identity expression from the stack
         unknown_stack.remove
      end
   end

finish_execute is
   -- Finish execution
   require
   retrieved_items_removed: unknown_item = void
   do
      -- Append the removed unknown expressions to the stack
      -- of unknown expressions
      unknown_stack.append (unknown_items)
      -- Remove all items from 'unknown_items'
      unknown_items.wipe_out

      -- If the stack of unknown items is empty
      if unknown_stack.is_empty then
         -- The extraction of the branch/switch condition
         -- is finished, stop the traversal
         is_stopped := true
      end
   end
item: EXPRESSION is
  -- Expression on top of stack
  do
    -- Create 'unknown_item', if it does not exist
    create_item
    -- Return the current 'identity of an unknown expression'
    Result := unknown_item
  end

remove is
  -- Remove expression on top of stack
  do
    -- Create 'unknown_item', if it does not exist
    create_item
    -- “Remove top of stack”
    unknown_item := void
  end

create_item is
  -- Create ‘unknown_item’, if it does not exist
  do
    -- If the ‘expression on top of stack’ does not exists yet
    if unknown_item = void then
      -- Create an ‘identity of an unknown expression’
      unknown_item := unknown_expression.identity
      -- Extend the list of unknown expressions
      unknown_items.extend(unknown_item)
    end
  end

Listing 5.15: Symbolic Execution (Part 4 of 4)

Precondition Extraction

The precondition extraction algorithm uses the subgraph from the code path extraction with the extracted branch conditions and switch value expressions from the symbolic execution as its input. Its result is a list of routine precondition expressions. The algorithm associates the precondition expression `false` to “exception” basic blocks and incrementally deduces the precondition expressions for the basic blocks preceding the “exception” basic blocks. Preconditions for intermediary basic blocks are computed by using the observation that for each succeeding basic block of an intermediary basic block the condition under which that succeeding block is reached during execution has to imply the succeeding basic blocks precondition.

The algorithm for precondition extraction uses the code path traversal algorithm with the actual generic parameter LIST [EXPRESSION] for the formal generic parameter G of the traversal’s result. The implementation of the queries `is_traversable` and `derived_result` is shown in listing 5.16 on page 70. The query `is_traversable` always returns `true` such that the whole subgraph is traversed.

The query `derived_result` receives the currently traversed basic block’s con-
tent a_block_item, a list of the results from the already traversed succeeding basic blocks some_results and a list of corresponding basic block link contents some_link_items. Each of the received results and the result of derived_result is a list of expressions. The routine derived_result first creates an empty list of expressions as its result. If the current basic block is the end of a path and is classified as exception basic block the result is extended with a precondition expression of "false". If the current basic block is not the end of a path the algorithm iterates over the succeeding links’ contents and the received results. At each iteration step the resulting list of precondition expressions is updated by the command update_result.

Listing 5.17 on page 71 shows the implementation of update_result. The routine receives the current resulting list of precondition expressions a_result, a list of precondition expressions from a succeeding basic block a_succeeding_result and the content of the basic block link leading to this succeeding basic block a_link_item. The routine update_result iterates over the precondition expressions of the succeeding basic block. Each iteration step extends the resulting list of precondition expressions with the expression: “the link condition implies the current precondition expression of the succeeding basic block”. This basic block graph traversal’s result is therefore a list of precondition expressions deduced from explicit exception cases.

Note 5.8 The actual implementation of the algorithm in class CIL_PRECONDITION_EXTRACTOR includes an optimization in terms of the construction of the precondition expressions in update_result. If all succeeding basic blocks have a precondition expression of “false” the result for the current basic block is also a precondition expression of “false”. If exactly one of the succeeding basic blocks does not have a precondition expression of “false”, that basic block’s precondition expressions are included in the result for the current basic block without prepending the link condition with an “implies” operator. This simplifies the extracted expressions.
is_traversable (a_block_item: BASIC_BLOCK_CONTENT): BOOLEAN is
   -- Should the traversal proceed with 'a_block_item'? 
   do
      -- Traverse all paths on the subgraph of tractable paths
      Result := true
   end

derived_result (a_block_item: BASIC_BLOCK_CONTENT;
   some_results: LIST [LIST [EXPRESSION]];
   some_link_items: LIST [BASIC_BLOCK_LINK_CONTENT] ) : LIST [EXPRESSION] is
   -- Result of 'a_block_item' derived from recursively computed
   -- 'some_results' and the corresponding 'some_link_items'
   require
      equal_counts: some_results. count = some_link_items. count
   do
      -- Create the resulting list of precondition clauses
      create Result. make (10)

      -- If 'a_block_item' is at the end of a path
      if some_results. is_empty then
         -- If the instructions in 'a_block_item' throw an exception
         if a_block_item. is_throwing then
            -- Extend the list of precondition clauses with 'false'
            Result. extend (false_constant_expression)
         end
      else
         -- Compute the precondition clauses of the current basic block
         -- from the clauses of the succeeding basic blocks: Iterate over
         -- the links to and the results of the succeeding basic blocks
         from
            some_results. start
            some_link_items. start
         until
            some_results. after
         do
            -- Include the 'link condition' and the result from
            -- the succeeding basic block in the current result
            update_result (Result, some_results. item,
               some_link_items. item)
            some_results. forth
            some_link_items. forth
         end
      end
   end

Listing 5.16: Precondition Extraction (Part 1 of 2)
update_result (a_result: LIST [EXPRESSION];
    a_succeeding_result: LIST [EXPRESSION];
    a_link_item: BASIC_BLOCK_LINKCONTENT) is
    -- Include the ‘link condition’ and the result from
    -- the succeeding basic block in the current result
    do
        -- Iterate over precondition clauses of the succeeding basic block
        from
        a_succeeding_result.start
        until
        a_succeeding_result.after
        loop
            -- Extend the result with
            -- “link condition implies succeeding precondition clause”
            a_result.extend (a_link_item.condition implies
                a_succeeding_result.item.expression)
        end
        a_succeeding_result.forth
    end
Listing 5.17: Precondition Extraction (Part 2 of 2)

5.2 Directed Graphs

The classes for directed graphs are used to represent basic block (see section 5.4 on page 84) graphs.

Note 5.9 Graphs are currently not represented by their own class (e.g., CIL_GRAPH).

A General Abstraction

The generic class CIL_GENERAL_MULTILINKABLE [G, H] inherits from CELL [G] (see figure 5.1 on the following page) the attribute item of type G and the equivalent commands put and replace to set the item. CIL_GENERAL_MULTILINKABLE adds the attributes predecessors and successors of type LIST [H] with corresponding setters and a creation routine make setting item to the received argument and initializing predecessors and successors to empty lists.

The generic descendant CIL_MULTI_LINKABLE [G] of CIL_GENERAL_MULTI_LINKABLE [G, CIL_MULTI_LINKABLE [G]] is no longer in use. It can be used for graphs where the edges do not carry their own attributes.

Vertices

Vertices of a graph are represented by the generic class CIL_VERTEX [G, H]. CIL_VERTEX [G, H] inherits from the generic class CIL_GENERAL_MULTILINKABLE [G, CIL_EDGE [H, G]]
The yellow ellipses represent classes and the red single arrows are inheritance relations.

(see figure 5.1). The inherited attribute `item` of CIL\textunderscore VERTEX [G, H] therefore is of type G while the attributes `predecessors` and `successors` are of type LIST [CIL\textunderscore EDGE [H, G]]. CIL\textunderscore VERTEX also adds the status report queries `has_successor` and `has_predecessor`, which take an argument of type like Current.

The descendant CIL\textunderscore BASIC\textunderscore BLOCK inherits from class CIL\textunderscore VERTEX with generic parameters CIL\textunderscore BASIC\textunderscore BLOCK\textunderscore CONTENT and CIL\textunderscore BASIC\textunderscore BLOCK\textunderscore LINK\textunderscore CONTENT and redefines `predecessors` and `successors` to be of type LIST [CIL\textunderscore BASIC\textunderscore BLOCK\textunderscore LINK] (see also section 5.4 on page 84). The class CIL\textunderscore BASIC\textunderscore BLOCK avoids unnecessary long type specifications when dealing with basic block graphs.

**Edges**

Edges of a graph are represented by the generic class CIL\textunderscore EDGE [G, H], which inherits from CELL [G] (see figure 5.2 on the next page) the attribute `item` of type G and the equivalent commands `put` and `replace` to set the `item`. An edge additionally has the attributes `origin` and `target` of type CIL\textunderscore VERTEX [H, G] with corresponding setters and a creation routine `make` receiving an origin, a target and an item as arguments. CIL\textunderscore EDGE also provides the status report queries `is_target` and `is_origin` with one argument of type like `target` and like `origin` respectively.

The class CIL\textunderscore BASIC\textunderscore BLOCK\textunderscore LINK inherits from class CIL\textunderscore EDGE with generic parameters CIL\textunderscore BASIC\textunderscore BLOCK\textunderscore LINK\textunderscore CONTENT and CIL\textunderscore BASIC\textunderscore BLOCK\textunderscore CONTENT and redefines `origin` and `target` to be of type CIL\textunderscore BASIC\textunderscore BLOCK (see also section 5.4 on page 84). The class CIL\textunderscore BASIC\textunderscore BLOCK\textunderscore LINK avoids unnecessary long type specifications when dealing with basic block graphs.
Path Traversal Algorithms

The generic deferred class CIL\_PATH\_TRAVERSAL \([G, H, K]\) implements a path traversal algorithm for graphs with vertices of type CIL\_VERTEX \([G, H]\) and edges of type CIL\_EDGE \([H, G]\) and results for each vertex of type \(K\). The effective descendant CIL\_FORWARD\_PATH\_TRAVERSAL \([G, H, K]\) (see figure 5.3 on the next page) implements the deferred routines such that the algorithm starts at a client-supplied end-vertex of the paths and then searches for possible start-vertices. Circles in the graph are detected and remain untraversed. Once a start-vertex of a path has been found, the algorithm queries the strategy attached to the attribute strategy of type CIL\_PATH\_TRAVERSAL\_STRATEGY for a result of this vertex. The algorithm queries the strategy again for the results of the inner vertices of the found paths, where it additionally supplies the results from all adjacent vertices closer to start-vertices of paths with that inner vertex. This continues until the result for the originally supplied vertex has been computed. The effective descendant CIL\_BACKWARD\_PATH\_TRAVERSAL \([G, H, K]\) starts at a start-vertex of the considered paths and starts querying the strategy when it found an end-vertex of a path. The strategy is set by the creation routine make or the command set\_strategy. The command traverse receives an argument of type CIL\_VERTEX \([G, H]\) and performs the traversal. The results are made available after the traversal through the query results of type LIST \([K]\). traversal can be called several times, the results are accumulated for all calls and reset again by the reset command.

A path traversal strategy inherits from the generic deferred class CIL\_PATH\_TRAVERSAL\_STRATEGY \([G, H, K]\) and effects the query derived\_result of type \(K\), which receives as its first argument the vertex item of type \(G\) and a list of tuples of an edge item of type \(H\), and a corresponding previous result of type \(H\) as its second argument. The strategy can additionally redefine the status report query is\_traversable with a vertex item of type \(G\) as its only argument in order to control the traversal of vertices.

Simpler path traversal strategies can inherit from the generic deferred class CIL\_DEFAULT\_PATH\_TRAVERSAL\_STRATEGY \([G, H, K]\), which itself inherits from CIL\_PATH\_TRAVERSAL\_STRATEGY \([G, H, K]\) (see figure 5.4 on page 75) and effects the derived\_result query. Descendants effect the query...
Expressions are represented by objects of type CIL_EXPRESSION. The deferred class CIL_EXPRESSION defines the interface of expressions and includes factory queries corresponding to several operations. Its most important deferred descendants are (see figure 5.5 on page 77): CIL_UNARY_EXPRESSION, CIL_BINARY_EXPRESSION and CIL_TERMINAL_EXPRESSION. Expressions with a corresponding effective descendant of CIL_EXPRESSION are usually semantically equivalent or closely related to a CIL instruction.
Figure 5.4: Path Traversal Strategy Class Diagram

Table 5.1: Deferred routines of CIL\_EXPRESSION

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>children</td>
<td>A list of child expressions</td>
</tr>
<tr>
<td>arity</td>
<td>The number of child expressions</td>
</tr>
<tr>
<td>is_known</td>
<td>Is the expression (including its child expressions) known?(^a)</td>
</tr>
<tr>
<td>is_defined</td>
<td>Is the expression (including its child expressions) defined?(^a)</td>
</tr>
<tr>
<td>simplified</td>
<td>Simplified expression equivalent to the current expression</td>
</tr>
<tr>
<td>accept</td>
<td>Accept a Visitor of type CIL_EXPRESSION_VISITOR.(^b)</td>
</tr>
</tbody>
</table>

\(^a\) See section 5.3 on page 79.
\(^b\) See section 5.3 on page 81.
Table 5.2: Effective factory queries of CIL\textsc{Expression}

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>infix &quot;#&quot;</code></td>
<td>“Equal” expression of Current and other</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>/=</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>&gt;</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>&gt;</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>&lt;</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>&lt;</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>`infix &quot;</td>
<td>&lt;=</td>
</tr>
<tr>
<td><code>infix &quot;implies&quot;</code></td>
<td>“Implies” expression of Current and other</td>
</tr>
<tr>
<td><code>infix &quot;+&quot;</code></td>
<td>“Addition” expression of Current and other</td>
</tr>
<tr>
<td><code>infix &quot;-&quot;</code></td>
<td>“Subtraction” expression of Current and other</td>
</tr>
<tr>
<td><code>prefix &quot;not&quot;</code></td>
<td>“Bitwise inversion” expression of Current</td>
</tr>
<tr>
<td><code>identity</code></td>
<td>“Identity” expression of Current</td>
</tr>
<tr>
<td>`infix &quot;.</td>
<td>&quot;`</td>
</tr>
</tbody>
</table>

\(^a\) Where other refers to a single argument of type CIL\textsc{Expression}.

**Note 5.10** CIL\textsc{Expression} currently lacks an attribute type of the unimplemented type CIL\textsc{Expression\_Type} or similar. Knowledge of the type of an expression (32-bit integer, 64-bit floating point, \ldots see also [7, section 12.3.2.1]) would allow additional simplification of expressions and correct transformations into languages like Eiffel.

**Note 5.11** There is currently not an expression class for each CIL instruction where it would be appropriate. This needs to be extended as the symbolic execution becomes more complete (see note 5.15 on page 92).

The direct deferred descendant CIL\_DEFERRED\_UNARY\_EXPRESSION of CIL\_EXPRESSION affects the queries children to return a one-element list, arity to return one and is\_defined and is\_known to return the same result as its operand. The operand is accessible through the deferred query operand of type CIL\_EXPRESSION. CIL\_DEFERRED\_UNARY\_EXPRESSION was introduced for supporting CIL\_DELAYED\_EXPRESSION, which effects the query operand as a routine.

The direct deferred descendant CIL\_UNARY\_EXPRESSION of CIL\_DEFERRED\_UNARY\_EXPRESSION effects the queries simplified to return a clone of Current with a simplified operand and operand as an attribute with a corresponding setter and creation routine.

The direct deferred descendant CIL\_BINARY\_EXPRESSION of CIL\_UNARY\_EXPRESSION renames operand to left\_operand and set\_operand to set\_left\_operand and adds the attribute right\_operand of
type CIL_EXPRESSION with a corresponding setter and a creation routine receiving both operands as its arguments. CIL_BINARY_EXPRESSION also redefines the routines children, arity, is_known, is_defined and simplified to take the second operand into account in their implementation.

Figure 5.5: Expression Class Diagram

Logic Expressions

All currently implemented logic expression classes inherit from the deferred class CIL_BINARY_EXPRESSION and effect the command accept of the expression Visitor pattern.

The two classes CIL_EQUAL_EXPRESSION and CIL_UNEQUAL_UNORDERED_EXPRESSION additionally re-define the Factory Query prefix “not” to return an in-
stance of CIL\_UNEQUAL\_UNORDERED\_EXPRESSION and CIL\_EQUAL\_EXPRESSION respectively with the same operands.

Note 5.12 The relational expressions different from CIL\_EQUAL\_EXPRESSION and CIL\_UNEQUAL\_UNORDERED\_EXPRESSION cannot provide a specialized implementation of prefix “not” returning their logical inverse relational expression because the correct logical inverse depends on the operands type. The inverse of, for example, a CIL\_LESS\_UN\_EXPRESSION is a CIL\_GREATER\_EQUAL\_UN\_EXPRESSION for integer operands (treated as unsigned integers in this case) and a CIL\_GREATER\_EQUAL\_EXPRESSION for floating-point operands. However, expression types are not supported yet (see note 5.10 on page 76).

CIL\_HAS\_EXPRESSION additionally redefines the attribute left\_operand to be of type CIL\_SET\_EXPRESSION and the query simplified to return false\_constant\_expression of CIL\_EXPRESSION\_CONSTANTS if the set is empty, an “equal” expression if the set contains only one element and the precursor’s result otherwise.

CIL\_IMPLIES\_EXPRESSION also redefines the query simplified for handling special cases.

Table 5.3: Logic expression classes and related CIL instructions

<table>
<thead>
<tr>
<th>Class</th>
<th>Instructions</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIL_EQUAL_EXPRESSION</td>
<td>beq, ceq</td>
<td>3.5, 3.21</td>
</tr>
<tr>
<td>CIL_UNEQUAL_UNORDERED_EXPRESSION</td>
<td>bne.un</td>
<td>3.14</td>
</tr>
<tr>
<td>CIL_GREATER_EXPRESSION</td>
<td>bgt, cgt</td>
<td>3.8, 3.22</td>
</tr>
<tr>
<td>CIL_GREATER_UN_EXPRESSION</td>
<td>bgt.un,</td>
<td>3.9, 3.23</td>
</tr>
<tr>
<td></td>
<td>cgt.un</td>
<td></td>
</tr>
<tr>
<td>CIL_GREATER_EQUAL_EXPRESSION</td>
<td>bge</td>
<td>3.6</td>
</tr>
<tr>
<td>CIL_GREATER_EQUAL_UN_EXPRESSION</td>
<td>bge.un</td>
<td>3.7</td>
</tr>
<tr>
<td>CIL_LESS_EXPRESSION</td>
<td>bit, clt</td>
<td>3.12, 3.25</td>
</tr>
<tr>
<td>CIL_LESS_UN_EXPRESSION</td>
<td>bit.un,</td>
<td>3.13, 3.26</td>
</tr>
<tr>
<td></td>
<td>clt.un</td>
<td></td>
</tr>
<tr>
<td>CIL_LESS_EQUAL_EXPRESSION</td>
<td>ble</td>
<td>3.10</td>
</tr>
<tr>
<td>CIL_LESS_EQUAL_UN_EXPRESSION</td>
<td>ble.un</td>
<td>3.11</td>
</tr>
<tr>
<td>CIL_IMPLIES_EXPRESSION</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CIL_HAS_EXPRESSION</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

* Sections of [9]

**Bitwise Expressions**

The only implemented bitwise operator expression is CIL\_NOT\_EXPRESSION corresponding to the instruction not specified in [9, section 3.52]. CIL\_NOT\_EXPRESSION inherits from CIL\_UNARY\_EXPRESSION and redefines the Factory Query prefix “not” to return its operand and the query simplified to apply prefix “not” to its operand. CIL\_NOT\_EXPRESSION also effects the accept command of the expression Visitor pattern.
Arithmetic Expressions

The currently implemented arithmetic expressions are: CIL\_ADD\_EXPRESSION and CIL\_SUB\_EXPRESSION corresponding to the instructions \texttt{add} (see \cite[section 3.1]{9}) and \texttt{sub} (see \cite[section 3.64]{9}) respectively. Both classes inherit from CIL\_BINARY\_EXPRESSION and effect the accept command of the expression Visitor pattern.

Identity Expressions

The identity expression classes CIL\_IDENTITY\_EXPRESSION and CIL\_DELAYED\_EXPRESSION inherit from CIL\_DEFERRED\_IDENTITY\_EXPRESSION and implement a Proxy pattern for expressions. CIL\_DEFERRED\_IDENTITY\_EXPRESSION inherits from CIL\_DEFERRED\_UNARY\_EXPRESSION (see figure 5.6 on the next page) and effects the query simplified to return its simplified operand.

The class CIL\_IDENTITY\_EXPRESSION inherits from CIL\_DEFERRED\_IDENTITY\_EXPRESSION and CIL\_UNARY\_EXPRESSION whereby it undefines and joins simplified of the latter with simplified of the former. CIL\_IDENTITY\_EXPRESSION therefore implements a Proxy with its real subject being attached to its attribute operand. CIL\_IDENTITY\_EXPRESSION also effects the accept command of the expression Visitor pattern. The symbolic execution uses instances of CIL\_IDENTITY\_EXPRESSION during backward execution for evaluation stack items it does not known yet (see section 5.7 on page 91).

The class CIL\_DELAYED\_EXPRESSION inherits from CIL\_DEFERRED\_IDENTITY\_EXPRESSION and adds an attribute operand\_agent of type FUNCTION [ANY, TUPLE [], like operand] with a setter and a creation routine make. It effects the query operand as a routine that calls the operand\_agent and returns its result. CIL\_DELAYED\_EXPRESSION also effects the accept command of the expression Visitor pattern. CIL\_DELAYED\_EXPRESSION therefore implements a Proxy with its real subject being computed on demand. Instances of CIL\_DELAYED\_EXPRESSION are used by the basic block analysis when building the basic block link’s condition (see section 5.4 on page 84).

Terminal Expressions

The direct deferred descendant CIL\_TERMINAL\_EXPRESSION of CIL\_EXPRESSION effects the queries children to return an empty list, arity to return zero and simplified to return Current. Terminal expressions are the only expressions that return Current from simplified all other expressions return a new instance in order to satisfy query semantics.

Void Expressions

The deferred class CIL\_VOID\_EXPRESSION inherits from CIL\_TERMINAL\_EXPRESSION (see figure 5.7 on page 81) and effects the query is\_known as a constant with value false. It also adds the attribute name of type STRING with a corresponding setter and creation routine.
The effective descendant CIL.UNDEFINED_EXPRESSION of CIL_VOID_EXPRESSION effects the features is_defined as a constant with value false and accept of the expression Visitor pattern (see section 5.3 on the next page).

The effective descendant CIL.UNKNOWN_EXPRESSION of CIL_VOID_EXPRESSION effects the features is_defined as a constant with value true and accept of the expression Visitor pattern (see section 5.3 on the following page).

The two descendants of CIL_VOID_EXPRESSION implement Null Object patterns (see [17]), corresponding instances are made available in CIL_EXPRESSION_CONSTANTS by the once queries unknown_expression and undefined_expression.

Other Terminal Expressions

Other terminal expressions inherit from the deferred descendant CIL_NON_VOID_TERMINAL_EXPRESSION of CIL_TERMINAL_EXPRESSION. CIL_NON_VOID_TERMINAL_EXPRESSION effects the queries is_defined and is_known as constants with value true.

The effective class CIL_CONSTANT_EXPRESSION [G] inherits from CIL_NON_VOID_TERMINAL_EXPRESSION and effects the accept command of the expression Visitor pattern. CIL_CONSTANT_EXPRESSION adds the attribute value of type G with an accompanying setter and creation routine make. Instances of generically derived types of CIL_CONSTANT_EXPRESSION represent constant values.

The effective descendant CIL_VARIABLE_EXPRESSION of CIL_NON_VOID_TERMINAL_EXPRESSION effects the accept command of the expression Visitor pattern and adds the attribute identifier of type CIL_IDENTIFIER (see section 4.11 on page 46) with an accompanying setter
and creation routine make. Instances of type CIL_VARIABLE_EXPRESSION are used to represent local and parameter variables and field identifiers.

Miscellaneous Expressions

The class CIL_DEREFERENCE_EXPRESSION inherits from CIL_BINARY_EXPRESSION and effects the accept command of the expression Visitor pattern. Instances of CIL_DEREFERENCE_EXPRESSION are used to represent access to a member of a type.

The class CIL_SET_EXPRESSION is a descendant of CIL_EXPRESSION and adds an attribute items of type LIST [CIL_EXPRESSION] accompanied by a setter set_items, a creation routine make initializing items to an empty list and a creation routine make_with_items receiving a list of items as its argument. CIL_SET_EXPRESSION effects the queries children to return a clone of items, arity to return the number of items and is_known, is_defined and simplified to compute their result from the list of items. It additionally provides a command extend for extending items and a Factory Query has for creating a “does the current set have the following element?” expression of Current and the argument expression. CIL_SET_EXPRESSION also effects the accept command of the expression Visitor pattern.

Visitor

The deferred class CIL_EXPRESSION_VISITOR provides a deferred visit command for each effective descendant of CIL_EXPRESSION and the apply command, which takes an argument of type CIL_EXPRESSION on which the Visitor should be applied.

The effective class CIL_EMPTY_EXPRESSION_VISITOR inherits from CIL_EXPRESSION_VISITOR (see figure 5.8 on the following page) and effects
all inherited \textit{visit} commands. The implementation of each \textit{visit} command includes debugging clauses for short and verbose output but is otherwise empty. This class allows its descendants to redefine only the necessary routines instead of effecting all of them.

The effective class \texttt{CIL\_DEFAULT\_EXPRESSION\_VISITOR} inherits from \texttt{CIL\_EXPRESSION\_VISITOR} and effects all inherited \textit{visit} commands. The implementation of each \textit{visit} command includes debugging clauses for short and verbose output and calls the deferred command \textit{visit}\_\textit{expression} with the visited expression as argument. This class allows its descendants to redefine routines corresponding to specific body items and provide a default implementation for all others.

![Expression Visitor Class Diagram](image)

Figure 5.8: Expression Visitor Class Diagram
String Representations

The class CIL_EXPRESSION_TO_STRING implements an algorithm for computing a string representation of a given expression. CIL_EXPRESSION_TO_STRING has an attribute `strategy` of type CIL_EXPRESSION_TO_STRING_STRATEGY accompanied by a setter and creation routine `make` to set the attribute. The algorithm is made available by the query `string`, which receives an argument of type CIL_EXPRESSION and returns the corresponding STRING.

The deferred class CIL_EXPRESSION_TO_STRING_STRATEGY inherits from CIL_EXPRESSION_VISITOR. It requires its deferred `visit` commands to set the precedence of the operator corresponding to the visited expression with `set_precedence` and extend the list `strings` with the string representations of the operator. The number of string representations for an operator is its arity plus one.

The class CIL_DEFAULT_EXPRESSION_TO_STRING_STRATEGY is a direct descendant of CIL_EXPRESSION_TO_STRING_STRATEGY. It effects all `visit` commands and defines deferred queries for the precedence and the operator strings of each supported expression.

The class CIL_EXPRESSION_TO_PSEUDO_EIFFEL_STRATEGY inherits from CIL_DEFAULT_EXPRESSION_TO_STRING_STRATEGY and effects the precedence and operator string queries as constants.

The translation from CIL to Eiffel will likely pose a few problems, some are already apparent with the current subset of CIL instructions supported by the expression classes. For example, the VES uses integers as the result of relational operations clt, cgt, ... and arithmetic operations on the result of relational operations (e.g., \((a < b) + (c > d)\)) are possible, although expected to be rare. A more important problem is that the CIL supports a wider variety of relational operators. There are, for example, blt (branch on less than) and blt.un. In the case of integers blt is used for signed, blt.un for unsigned operands. With floating-point numbers blt does not branch, but blt.un does, if the operands are unordered (i.e., if one or both of the operands is NaN).

This and future difficulties in the translation from CIL to Eiffel may suggest extensions to the Eiffel expression language.

Note 5.13 The implementation and design of the expression-to-string algorithm is in an early state.

Factory

The deferred class CIL_EXPRESSION_FACTORY defines Factory Queries for expressions that do not have a corresponding Factory Query in CIL_EXPRESSION or one of its descendants.

CIL_DEFAULT_EXPRESSION_FACTORY inherits from CIL_EXPRESSION_FACTORY and effects the Factory Queries to a default implementation.
5.4 Basic Block Analysis

The first step towards contract extraction is to analyze the method body and partition it into basic blocks. A basic block is a sequence of method body items where the execution control flow can only branch to the first and from the last body items in the basic block. The algorithm for basic block analysis creates a basic block graph where a vertex is a basic block and an edge represents a possible control flow from one basic block to another.

Basic Blocks

Basic blocks are represented by the class CIL_BASIC_BLOCK (see section 5.2 on page 71) whose item is of type CIL_BASIC_BLOCK_CONTENT.

The class CIL_BASIC_BLOCKCONTENT inherits the attribute body_items of type LIST [CIL_METHOD_BODY_ITEM] and the corresponding setter and extend command from CIL_METHOD_BODY_ITEMS_CONTEXT. The class CIL_BASIC_BLOCKCONTENT adds the attributes classification of type CIL_BASIC_BLOCK_CLASSIFICATION (see section 5.5 on page 88) and expression of type CIL_EXPRESSION (see section 5.3 on page 74) accompanied by corresponding setters and a creation routine make with no arguments. The creation routine initializes body_items to an empty list, classification to unknown_classification from CIL_BASIC_BLOCK_CLASSIFICATION_CONSTANTS and expression to unknown_expression from CIL_EXPRESSION_CONSTANTS. CIL_BASIC_BLOCKCONTENT also provides the status report queries is_classified with an argument of type CIL_BASIC_BLOCK_CLASSIFICATION, is_conditional and is_throwing without arguments.

The attribute expression of the class CIL_BASIC_BLOCKCONTENT is initially set to unknown_expression from CIL_EXPRESSION_CONSTANTS by the creation routine. The basic block analysis algorithm then sets it to undefined_expression from CIL_EXPRESSION_CONSTANTS if the basic block does not end with a conditional branch or switch instruction. Algorithms for extracting branch or switch conditions can check if a basic block is_conditional, then attempt to extract the corresponding condition and finally set the extracted condition (i.e., instance of CIL_EXPRESSION) on the basic block for later use (see section 5.7 on page 91 on symbolic execution).

Basic Block Links

Links between basic blocks are represented by instances of the class CIL_BASIC_BLOCK_LINK (see section 5.2 on page 71) whose item is of type CIL_BASIC_BLOCK_LINK_CONTENT.

The class CIL_BASIC_BLOCK_LINK_CONTENT has an attribute condition of type CIL_EXPRESSION (see section 5.3 on page 74) accompanied by a corresponding setter and a creation routine make. The basic block analysis algorithm sets the condition under which the execution would follow this particular link. As the basic block analysis algorithm is not capable of supplying the complete condition for a link in general, the expression attached to condition...
will usually have a leaf node of dynamic type CIL\_DEFERRED\_EXPRESSION (see section 5.3 on page 74).

**Algorithm**

The class CIL\_BASIC\_BLOCK\_ANALYZER encapsulates the basic block analysis algorithm and has an attribute `expression\_factory` of type CIL\_EXPRESSION\_FACTORY (see 5.3 on page 74) with a corresponding setter and creation routine `make`. The query `basic\_blocks` receives an argument of type CIL\_METHOD\_DEFINITION and returns a list of the computed basic blocks of type LIST [CIL\_BASIC\_BLOCK].

The basic block analysis is accomplished by two passes iterating over the method body items in the method's body. The first pass creates the basic blocks and records their start indices. The second pass fills the still empty basic blocks with the method body items corresponding to each basic block and connects them with basic block links. Both passes of the algorithm are implemented in method body item Visitors (see section 4.13 on page 47) CIL\_FIRST\_BASIC\_BLOCK\_ANALYSIS\_VISITOR and CIL\_SECOND\_BASIC\_BLOCK\_ANALYSIS\_VISITOR, both inheriting from CIL\_BASIC\_BLOCK\_ANALYSIS\_VISITOR (see figure 5.9 on the following page).

The deferred class CIL\_BASIC\_BLOCK\_ANALYSIS\_VISITOR inherits from CIL\_DEFAULT\_METHOD\_BODY\_ITEM\_VISITOR and redefines the `visit` commands of instructions that control the flow of execution. Each redefined `visit` command calls one of the deferred commands: `visit\_branch\_instruction`, `visit\_unconditional\_branch\_instruction` or `visit\_leaving\_instruction`. CIL\_BASIC\_BLOCK\_ANALYSIS\_VISITOR also has two other deferred commands: the inherited `visit\_method\_body\_item` and the newly undefined `visit\_switch\_instruction`. All five deferred commands are effected by the descendants of CIL\_BASIC\_BLOCK\_ANALYSIS\_VISITOR.

CIL\_BASIC\_BLOCK\_ANALYSIS\_VISITOR also introduces the attribute `basic\_block\_table` of type HASH\_TABLE [CIL\_BASIC\_BLOCK, INTEGER], which maps an index to the basic block starting at this index. This hash table is filled during the first pass and is retrieved from during the second pass of the basic block analysis algorithm.

<table>
<thead>
<tr>
<th>Command</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>visit\_branch\_instruction</code></td>
<td>beq, bge, bge.un, bgt, bgt.un, ble, ble.un, blt, blt.un, bne.un, brfalse, brtrue</td>
</tr>
<tr>
<td><code>visit\_unconditional\_branch\_instruction</code></td>
<td>br, leave</td>
</tr>
<tr>
<td><code>visit\_leaving\_instruction</code></td>
<td>endfilter, endfinally, jmp, ret, tail.call, tail.callvirt, tail.calli, rethrow, throw</td>
</tr>
<tr>
<td><code>visit\_switch\_instruction</code></td>
<td>switch</td>
</tr>
<tr>
<td><code>visit\_method\_body\_item</code></td>
<td>all other instructions and method body items</td>
</tr>
</tbody>
</table>
First Pass

The first pass over the method body items in the basic block analysis algorithm is implemented in the Visitor CIL-FIRST_BASIC_BLOCK_ANALYSIS_VISITOR. It extends the hash table attached to basic_block_table of type HASH_TABLE [CIL_BASIC_BLOCK, INTEGER] with a mapping from indices of items in the method body to basic blocks that start at that particular index. basic_block_table is initialized to a hash table with one basic block at the first index when the traversal starts. Each of the five effected commands (inherited from the direct ancestor CIL_BASIC_BLOCK_ANALYSIS_VISITOR) introduce newly created basic blocks in basic_block_table only if there is not already one associated with the corresponding index. The basic blocks are created but left empty in the first pass.
Table 5.5: Introduced basic blocks and effected commands of CIL_FIRST_BASIC_BLOCK_ANALYSIS_VISITOR

<table>
<thead>
<tr>
<th>Command</th>
<th>Introduced Basic Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>visit_branch_instruction</td>
<td>at the next and at the target instruction</td>
</tr>
<tr>
<td>visit_unconditional_branch_instruction</td>
<td>at the next if existent and at the target instruction</td>
</tr>
<tr>
<td>visit_leaving_instruction</td>
<td>at the next if existent</td>
</tr>
<tr>
<td>visit_switch_instruction</td>
<td>at the next and at all target instructions</td>
</tr>
<tr>
<td>visit_method_body_item</td>
<td>none</td>
</tr>
</tbody>
</table>

Second Pass

The second pass over the method body items in the basic block analysis algorithm is implemented in CIL_SECOND_BASIC_BLOCK_ANALYSIS_VISITOR. It extends the basic blocks in `basic_block_table` with the corresponding body items, extends its attribute `basic_blocks` of type LIST [CIL_BASIC_BLOCK] with the basic blocks as it encounters them during iteration and connects the basic blocks with basic block links.

Each time before a method body item is visited, the current basic block `basic_block` and the basic block starting at the next body item `next_basic_block` (which is `void` if there is no such basic block) are updated. The current basic block is extended with the current method body item and if a new basic block starts at the current body item `basic_blocks` is extended with `basic_block`. Each of the five effected commands (inherited from the direct ancestor CIL_BASIC_BLOCK_ANALYSIS_VISITOR) connect the current basic block attached to `basic_block` with a basic block link to the corresponding target basic blocks, if any. The basic block links are created with the current basic block as their origin, their target basic block and an expression representing the condition under which the link is taken during execution. If a basic block has only one successor then the `expression` attribute is set to `undefined_expression` from CIL_EXPRESSION_CONSTANTS to reflect the fact that there is no associated branch condition for this block (`expression` is attached to `unknown_expression` by default).

Table 5.6: Introduced basic block link targets and effected commands of CIL_SECOND_BASIC_BLOCK_ANALYSIS_VISITOR

<table>
<thead>
<tr>
<th>Command</th>
<th>Introduced Basic Block Linksa</th>
</tr>
</thead>
<tbody>
<tr>
<td>visit_branch_instruction</td>
<td>to the next and the target basic blocks</td>
</tr>
<tr>
<td>visit_unconditional_branch_instruction</td>
<td>to the target basic block</td>
</tr>
<tr>
<td>visit_leaving_instruction</td>
<td>none</td>
</tr>
<tr>
<td>visit_switch_instruction</td>
<td>to the next and all target basic blocks</td>
</tr>
<tr>
<td>visit_method_body_item</td>
<td>to the next basic block if it starts at the next body item</td>
</tr>
</tbody>
</table>

a With the current basic block as their origin.
5.5 Basic Block Classification

When extracting contracts, one has to track the system’s state during a hypothetical execution. Consider, for example, a method \( f \), which first calls another method \( g \) on the current object (\texttt{this}) and then checks if a field \( a \) of numeric type of \texttt{this} is positive and throws an exception if it is not. Should the field \( a \) therefore be positive when the considered method \( f \) is called? Not necessarily, the implementation of method \( g \) may change the field’s value before method \( f \) loads it. On the other hand, the call to method \( g \) may as well leave the field’s value unchanged, but without looking into the implementation of \( g \) it is not possible to decide if the field \( a \)’s value has to be positive when calling the method \( f \). Also note that there might be more than one implementation for method \( g \), for example, a descendant of the corresponding class may redefine it with its own implementation.

The chosen approach for precondition extraction only looks at the implementation of the method \( f \) and does not consider the preconditions that might arise from exceptions after a call to another method and a few other instructions whose effects are deemed too difficult to track.

After the basic block analysis, each block is classified according to the instructions it contains. Classifications are represented by instances of the class \texttt{CIL\_BASIC\_BLOCK\_CLASSIFICATION}. \texttt{CIL\_BASIC\_BLOCK\_CLASSIFICATION} inherits from \texttt{COMPARABLE} and effects the \texttt{infix} operator \(<\). A classification is less (or weaker) than another classification if its attribute \texttt{value} of type \texttt{INTEGER} is less than the \texttt{value} of the other classification. A classification also has an attribute \texttt{name} of type \texttt{STRING}. Both attributes are set on creation and not changed afterwards.

The class \texttt{CIL\_BASIC\_BLOCK\_CLASSIFICATION\_CONSTANTS} provides the used classifications as \texttt{once} queries.

The method body item Visitor (see section 4.13 on page 47) \texttt{CIL\_BASIC\_BLOCK\_CLASSIFICATION\_VISITOR} is used for classification of a basic block by applying it to all body items in that basic block. Each \texttt{visit} command sets the \texttt{classification} attribute of the corresponding \texttt{CIL\_BASIC\_BLOCK\_CONTENT} to the body item’s corresponding classification unless \texttt{classification} is already set to a stronger (i.e., greater in terms of \texttt{COMPARABLE}) classification, in which case the attribute is left unchanged. Each method body item is associated with a classification that reflects the difficulty of tracking its effects (in terms of implementing the tracking of the effects) on the system and hence the difficulty of extracting contracts after it. The \texttt{throw} and \texttt{rethrow} instructions are special in that they have the strongest associated classification \texttt{exception\_classification} from \texttt{CIL\_BASIC\_BLOCK\_CLASSIFICATION\_CONSTANTS}. A basic block that unconditionally ends with an exception always has a precondition of \texttt{false} even if the block contains calls to other methods.
Table 5.7: Classifications defined as constants and the corresponding method body items

<table>
<thead>
<tr>
<th>Classification</th>
<th>Method body items</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception_classification</td>
<td>rethrow, throw</td>
</tr>
<tr>
<td>intractable_classification</td>
<td>call, calli, callvirt, cpblk, initblk,</td>
</tr>
<tr>
<td></td>
<td>newobj, stind, stobj, tail.call, tail.calli,</td>
</tr>
<tr>
<td></td>
<td>tail.callvirt</td>
</tr>
<tr>
<td>difficult_classification</td>
<td>starg, stfld, stloc, stsfld, exception block,</td>
</tr>
<tr>
<td></td>
<td>scope block</td>
</tr>
<tr>
<td>normal_classification</td>
<td>all others</td>
</tr>
<tr>
<td>unknown_classification</td>
<td>none</td>
</tr>
</tbody>
</table>

a In decreasing order (i.e., from the strongest to the weakest classification)
b Deemed too difficult for a first approach
c Need more infrastructure for tracking their effects
d Initial classification of basic blocks before application of the classification algorithm

The class CIL_BASIC_BLOCK_CLASSIFIER encapsulates the classification algorithm and makes it available by its command classify, which takes a list of basic blocks as its argument and applies the classification Visitor on each of these basic blocks.

Classification Statistics

The class CIL_CLASSIFICATION_STATISTICS_EXTRACTOR implements an algorithm for collecting statistics on the classification of code paths. The intermediate and final results are stored by instances of class CIL_CLASSIFICATION_STATISTICS.

The class CIL_CLASSIFICATION_STATISTICS has an attribute result_table of type HASH_TABLE [like item, CIL_BASIC_BLOCK_CLASSIFICATION], which maps classifications to their number of occurrences. The number of occurrence of a classification stands either for the number of code paths with this classification as their strongest classification of any of their basic blocks or for the number of basic blocks that have this classification. The meaning depends on how the statistics where computed.

The extend command of CIL_CLASSIFICATION_STATISTICS increases the number of occurrences associated with the argument classification by one or, if the classification hasn’t been encountered before, initializes the number of occurrences to one. The update command first adds the number of occurrences of classifications weaker than the argument classification to the number of occurrences associated with the argument classification. update then sets the number of occurrences associated with classifications weaker than the argument classification to zero. update is used for incrementally computing code path statistics. The command add cumulates the results of its argument classification statistics with the results of the current statistics. The query infix “+” returns a new classification statistics with the cumulated results of the argument and the current statistics. CIL_CLASSIFICATION_STATISTICS also provides make without arguments and one argument make_with_result_table creation routines,
the setter set_result_table and a few convenience routines for accessing the result table: item, count, is_empty and has.

CIL\_CLASSIFICATION\_STATISTICS\_EXTRACTOR has an attribute threshold of type CIL\_BASIC\_BLOCK\_CLASSIFICATION with a corresponding setter and creation routine make. Classifications equal to or stronger than the threshold are not taken into account for basic blocks other than the first. The statistics extraction algorithm is made available by the command extract, which takes a list of basic blocks as argument. The algorithm extracts the statistics for code paths that end at the basic blocks received by extract and makes the results available through its attributes of type CIL\_CLASSIFICATION\_STATISTICS: code_path_statistics, basic_block_statistics and conditional_basic_block_statistics.

Note 5.14 “Classifications equal to or stronger than the threshold are not taken into account for basic blocks other than the first” is not very intuitive. With threshold set to exception_classification of CIL\_BASIC\_BLOCK\_CLASSIFICATION\_CONSTANT the results for code paths will ignore the exception classification at the end of the paths unless the code path has only that one last basic block, in which case this information is made available by the statistics. This however only makes sense if the classification attached to threshold implies that a corresponding basic block is the last in a path.

CIL\_CLASSIFICATION\_STATISTICS\_EXTRACTOR is a descendant of CIL\_DEFAULT\_PATH\_TRAVERSAL\_STRATEGY and effects the routines initial_result and update_result and redefines result_for_first and finish_result. The extract command uses an instance of CIL\_FORWARD\_PATH\_TRAVERSAL for traversing the basic blocks graph. The result of the traversal is the code path statistics, the statistics for basic blocks and conditional basic blocks are directly computed with the instances attached to basic_block_statistics and conditional_basic_block_statistics. See also section 5.2 on page 73.

Table 5.8: Implementation of the code path traversal strategy by CIL\_CLASSIFICATION\_STATISTICS\_EXTRACTOR

<table>
<thead>
<tr>
<th>Routine</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>result_for_first</td>
<td>returns the result from initial_result, additionally extends the result with the classification of the currently traversed basic block</td>
</tr>
<tr>
<td>initial_result</td>
<td>returns an empty classification statistics, extends the basic block statistics and—if the basic block is conditional—the conditional basic blocks statistics with the classification of the currently traversed basic block</td>
</tr>
<tr>
<td>update_result</td>
<td>adds the preceding result to the current result</td>
</tr>
<tr>
<td>finish_result</td>
<td>updates the result with the classification of the currently traversed basic block</td>
</tr>
</tbody>
</table>

a See above about extend of CIL\_CLASSIFICATION\_STATISTICS
b See above about add of CIL\_CLASSIFICATION\_STATISTICS
c See above about update of CIL\_CLASSIFICATION\_STATISTICS
5.6 Code Path Extraction

Once all basic blocks have been classified it is convenient to create a graph of basic blocks that is a subgraph of the original basic blocks graph containing only code paths whose basic blocks are classified with *normal_classification* or *exception_classification*. Algorithms for extracting contracts can then operate on the subgraph instead of the complete basic blocks graph of a method.

The class CIL_CODE_PATH_EXTRACTOR implements such an algorithm and makes it available by the query *paths*, which takes a list of basic blocks as first argument and a boolean value indicating if the forward or the backward path traversal algorithm should be applied as second argument. *paths* returns a list of basic blocks of the newly created subgraph corresponding to the basic blocks supplied as first argument. The returned number of basic blocks can be less than the supplied. The created subgraph only contains basic blocks not exceeding (i.e., less or equal in terms of COMPARABLE) the classification attached to the attribute *threshold* or equal or stronger than the classification attached to *special_threshold*. *threshold* and *special_threshold* are set on creation by the creation routine *make* and can be set afterwards by corresponding setters. CIL_CODE_PATH_EXTRACTOR is currently used with *threshold* set to *normal_classification* and *special_threshold* set to *exception_classification* of CIL_BASIC_BLOCK_CLASSIFICATION_CONSTANTS.

The class CIL_CODE_PATH_EXTRACTOR inherits from CIL_DEFAULT_PATH_TRAVERSAL_STRATEGY with generic parameters CIL_BASIC_BLOCK_CONTENT, CIL_BASIC_BLOCK_LINK_CONTENT and CIL_BASIC_BLOCK and effects or redefines the routines *initial_result*, *update_result* and *is_traversable* for implementing the code path extraction. *paths* uses an instance of CIL_FORWARD_PATH_TRAVERSAL (if its second argument is *true*) or CIL_BACKWARD_PATH_TRAVERSAL (if its second argument is *false*) for traversing the original graph and invoking CIL_CODE_PATH_EXTRACTOR’s traversal strategy.

Table 5.9: Implementation of the code path traversal strategy by CIL_CODE_PATH_EXTRACTOR

<table>
<thead>
<tr>
<th>Routine</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>initial_result</em></td>
<td>returns a newly created basic block with the basic block content received as its argument</td>
</tr>
<tr>
<td><em>update_result</em></td>
<td>adds a newly created basic block edge with the basic block link content received as its argument between the previous result (a basic block) and the current result (also a basic block)</td>
</tr>
<tr>
<td><em>is_traversable</em></td>
<td>returns <em>true</em> if the received basic block content’s classification is less than or equal to <em>threshold</em> or greater than or equal to <em>special_threshold</em></td>
</tr>
</tbody>
</table>

5.7 Symbolic Execution

While the current approach only considers explicitly thrown exceptions, it already requires an algorithm that is able to determine under what conditions such an exception is thrown. The first step towards the exception condition—
whose logical inversion is a precondition—is the extraction of branch conditions. The current approach traverses the body items in a basic block that finishes in a conditional branch backwards starting at the final branch or switch instruction. During the traversal the current body item is examined by a method body item Visitor (see section 4.13 on page 47) and a subexpression (see section 5.3 on page 74) of the branch condition is built. The extracted subexpression is attached to the basic blocks expression attribute when the traversal is finished.

In the case of a basic block finishing with a branch instruction, the extracted subexpression represents the condition under which the branch is taken. In the case of a switch instruction at the end of the considered basic block the extracted subexpression represents the integer value on which the switch operation is executed. The basic block links from a conditional basic block to its successors complete the extracted subexpression with the expression attached to their attribute condition, which includes the subexpression as a child expression (see also section 5.4 on page 87).

The algorithm for symbolic execution is encapsulated in the class CIL_SYMBOLIC_EXTRACTOR, which makes it available by the command extract receiving a list of the basic blocks. extract iterates over the list of basic blocks and applies an instance of CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR to all basic blocks that are conditional and whose expression is still unknown.

The symbolic execution is implemented in the deferred method body item Visitor CIL_SYMBOLIC_EXECUTION_VISITOR and its effective descendant CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR (see figure 5.10 on the following page). The class CIL_SYMBOLIC_EXECUTION_VISITOR inherits from CIL_DEFAULT_METHOD_BODY_ITEM_VISITOR (see section 4.13 on page 47) and effects the command visit method body item to stop the traversal as for the default case the corresponding body item is not yet supported by the symbolic execution. CIL_SYMBOLIC_EXECUTION_VISITOR defines the attribute expression factory of type CIL_EXPRESSION_FACTORY (see section 5.3 on page 83). The attribute is set by the creation routine make receiving as argument and the setter set_expression_factory. CIL_SYMBOLIC_EXECUTION_VISITOR defines the deferred routines item, remove and extend, which are used by the currently implemented visit routines for accessing the VES evaluation stack (see also [7, section 12.3.2.1]). These routines are implemented by CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR.

The deferred class CIL_SYMBOLIC_EXECUTION_VISITOR redefines the visit commands corresponding to the method body items it currently supports. The implementation of these visit commands use the routines item, remove and extend for manipulating the evaluation stack as if execution would be carried out in the usual forward order. It is the implementation in CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR, which ensures that execution is in effect done in reverse order. This however imposes the restriction that extend is called only after all calls to item and remove during a single visit.

**Note 5.15** Currently only a few visit commands are redefined in CIL_SYMBOLIC_EXECUTION_VISITOR to implement symbolic execution of the corresponding method body item. See also note 5.11 on page 76.

**Note 5.16** An additional restriction with extend of
CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR is that it can only be called once in a single visit. This is currently sufficient but can be extended to handle several calls per visit.

CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR maintains a stack of still unknown expressions attached to the attribute unknown_stack of type STACK [CIL_IDENTITY_EXPRESSION]. When a visit command requests the item of the evaluation stack through item the first time or after remove has been called a new identity expression is created with the unknown_expression of CIL_EXPRESSION_CONSTANTS as its operand and attached to the attribute unknown_item of type CIL_IDENTITY_EXPRESSION. item always returns the expression attached to unknown_item as its result. When the item should be removed from the evaluation stack by remove the temporary list unknown_items of type LIST [CIL_IDENTITY_EXPRESSION] is extended with the current unknown_item and unknown_item is reset to void. When the evaluation stack is extended by a call to extend the operand of the identity expression on top of unknown_stack is set to the expression received as the argument of extend and the item of the stack is removed. At the end of a visit finish_visit appends the unknown_items to the unknown_stack.
listing 5.18: visit_bge_instruction of CIL_SYMBOLIC_EXECUTION_VISITOR

the visit command for the “branch on greater or equal” instruction first looks up the item of the stack and removes it from the stack. Then, it creates a “greater or equal” binary expression with the first and the current stack item and again removes the current stack item and finally sets the subexpression of the branch condition on the context basic block.

Table 5.10: Summary of the implementation of the evaluation stack in CIL_BACKWARD_SYMBOLIC_EXECUTION_VISITOR

<table>
<thead>
<tr>
<th>Feature</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown_stack</td>
<td>is the stack of unknowns and represents the state of the evaluation stack before and after but not during a visit</td>
</tr>
<tr>
<td>unknown_item</td>
<td>caches a Proxy expression of an unknown expression on top of the evaluation stack until it is removed</td>
</tr>
<tr>
<td>unknown_items</td>
<td>stores the Proxy expressions of a single visit</td>
</tr>
<tr>
<td>item</td>
<td>creates a Proxy for a still unknown expression</td>
</tr>
<tr>
<td>remove</td>
<td>adds the Proxy to the temporary list of unknowns</td>
</tr>
<tr>
<td>extend</td>
<td>resolves the operand of the Proxy on top of the stack of unknowns and removes the item from the stack</td>
</tr>
<tr>
<td>finish_visit</td>
<td>appends the temporary list of unknowns to the stack of unknowns</td>
</tr>
</tbody>
</table>

5.8 Precondition extraction

After the symbolic execution has extracted the branch conditions, the precondition extraction algorithm builds precondition expressions from these branch conditions. The algorithm basically starts at basic blocks throwing explicit exceptions and associates to them a precondition with the expression false. Then, it traverses the basic blocks backwards to the first basic block of the method and associates preconditions with each traversed basic block. The preconditions associated with each intermediate basic block of the method and associates preconditions with each traversed basic block. The preconditions associated with each intermediate basic block have expressions of the form “branch condition of succeeding link implies precondition of the same succeeding link’s target basic block”. If, for example, the branch condition of a link of the considered basic block is index < 0 and the target basic block of that link
throws an explicit exception, that is, its precondition is \texttt{false}, then the basic block under consideration is associated with a precondition \texttt{index < 0 implies false}. This is done for each basic block and for each succeeding link of that basic block.

Precondition clauses are implemented by the class \texttt{CIL\_PRECONDITION\_CLAUSE}, which has the two attributes \texttt{label} of type \texttt{STRING} and \texttt{expression} of the type \texttt{CIL\_EXPRESSION}. Both attributes are set by the creation routine \texttt{make} to its received arguments and can be set by their corresponding setter commands.

The precondition extraction is implemented in the descendant \texttt{CIL\_PRECONDITION\_EXTRACTOR} of \texttt{CIL\_PATH\_TRAVERSAL\_STRATEGY} (see section 5.2 on page 73). The path traversal runs on the complete basic blocks graph, not on the subgraph of tractable paths (see also section 5.6 on page 91); this allows for a few optimizations of the extracted preconditions. \texttt{CIL\_PRECONDITION\_EXTRACTOR} has attributes \texttt{threshold} and \texttt{special\_threshold}, which are set by the creation routine \texttt{make} and have corresponding setter routines. The algorithm computes the preconditions for all basic blocks with a \texttt{classification} less than or equal to the \texttt{threshold} or greater than or equal to the \texttt{special\_threshold}. The query \texttt{preconditions} takes the first basic block of a method as its argument, applies a backward path traversal to the basic block and returns the result of the traversal—the list of extracted precondition clauses.

The class \texttt{CIL\_PRECONDITION\_EXTRACTOR} effects the query \texttt{derived\_result} of its ancestor \texttt{CIL\_PATH\_TRAVERSAL\_STRATEGY} with generic parameters \texttt{CIL\_BASIC\_BLOCK\_CONTENT}, \texttt{CIL\_BASIC\_BLOCK\_LINK\_CONTENT} and \texttt{LIST \{CIL\_PRECONDITION\_CLAUSE\}}. \texttt{derived\_result} first of all creates an empty \texttt{ARRAYED\_LIST \{CIL\_PRECONDITION\_CLAUSE\}} as its \texttt{Result}. If the classification of the query argument \texttt{a\_vertex\_item}—the current basic block content—is greater than the \texttt{threshold} and less than the \texttt{special\_threshold} the basic block is deemed intractable and the result is already complete. Otherwise the argument \texttt{some\_previous\_results} is queried if it is empty and the current basic block content if it throws an explicit exception, if both is true the \texttt{Result} is extended by a precondition clause with the expression set to \texttt{false\_constant\_expression} of \texttt{CIL\_EXPRESSION\_CONSTANTS} (the label is unused and set to a fixed string “unknown\_label”). If, on the other hand, \texttt{some\_previous\_results} is not empty, the algorithm instead proceeds by counting the number of previous lists of precondition clauses that do not contain the \texttt{false\_constant\_expression}—and therefore their corresponding basic blocks do not necessarily lead to an exception. If none of the succeeding basic blocks may succeed the \texttt{Result} is extended by a precondition clause with the expression set to \texttt{false\_constant\_expression}. If only one of the succeeding basic blocks may succeed, the \texttt{Result} is extended by the logical inverse of all branch conditions corresponding to target basic blocks that surely fail and the precondition clauses of the only succeeding basic block that may succeed. If more than one of the succeeding basic blocks may succeed, the \texttt{Result} is extended by precondition clauses with expressions “branch condition \texttt{implies} precondition clause expression” for all basic block link branch condition and all corresponding precondition clause expressions of the target basic block.
5.9 Results

The current implementation for contract extraction has been run on ArrayList, Stack and Queue from the .NET libraries in [16]. The following subsections present the results from the classification statistics extraction (see section 5.5 on page 89) and symbolic execution (see section 5.7 on page 91) and discusses the interpretation and the relevance of the extracted preconditions.

Classification Statistics

Table 5.11: Number of code paths finishing at an exception basic block and their classification

<table>
<thead>
<tr>
<th>Classification</th>
<th>ArrayList</th>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>114</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Difficult</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Intractable</td>
<td>92</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Exception(a)</td>
<td>36</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>246</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

\(a\) Code paths of only one basic block.

There are 246 code paths in methods of ArrayList or one of its nested classes that finish at an exception basic block (see table 5.9). The current implementation for contract extraction addresses the 150 normal and exception code paths. The other 96 difficult or intractable code paths are not considered in the current approach.

Table 5.12: Number of basic blocks on normal and exception code paths finishing at an exception

<table>
<thead>
<tr>
<th></th>
<th>ArrayList</th>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>227</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Conditional</td>
<td>99</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Extracted Branch Conditions</td>
<td>98</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

The symbolic execution extracts 98 of 99 branch conditions in ArrayList (see table 5.9). The extraction of the only unextracted branch condition stops at a conv instruction.
### Extracted Preconditions

Table 5.13: Extracted precondition clauses and their number of occurrence in `ArrayList`

<table>
<thead>
<tr>
<th>Precondition Clause</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>this</td>
<td>/=</td>
</tr>
<tr>
<td>c</td>
<td>/=</td>
</tr>
<tr>
<td>type</td>
<td>/=</td>
</tr>
<tr>
<td>array</td>
<td>/=</td>
</tr>
<tr>
<td>not (index &lt; 0)</td>
<td>22</td>
</tr>
<tr>
<td>count &gt;= 0</td>
<td>14</td>
</tr>
<tr>
<td>(this._size - index) &gt;= count</td>
<td>7</td>
</tr>
<tr>
<td>index &lt; this._size</td>
<td>3</td>
</tr>
<tr>
<td>index &lt;= this._size</td>
<td>2</td>
</tr>
<tr>
<td>value &gt;= 0</td>
<td>1</td>
</tr>
<tr>
<td>arrayIndex &gt;= 0</td>
<td>1</td>
</tr>
<tr>
<td>(startIndex + count) &lt;= this._size</td>
<td>1</td>
</tr>
<tr>
<td>not (startIndex &lt; 0)</td>
<td>1</td>
</tr>
<tr>
<td>startIndex &lt; this._size</td>
<td>1</td>
</tr>
<tr>
<td>startIndex &lt;= this._size</td>
<td>1</td>
</tr>
<tr>
<td>this._remaining &gt;= 0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>this.version = this.list._version</td>
<td>4</td>
</tr>
<tr>
<td>this._baseVersion = this._baseList._version</td>
<td>1</td>
</tr>
<tr>
<td>this._firstCall = 0</td>
<td>1</td>
</tr>
<tr>
<td>this._size</td>
<td>/=</td>
</tr>
<tr>
<td>this._size</td>
<td>/=</td>
</tr>
<tr>
<td>this._size</td>
<td>/=</td>
</tr>
<tr>
<td>this._size</td>
<td>/=</td>
</tr>
<tr>
<td>Total</td>
<td>115</td>
</tr>
</tbody>
</table>
Table 5.14: Selection of extracted precondition clauses from ArrayList

<table>
<thead>
<tr>
<th>Method / Precondition Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 LastIndexOf(object value, int32 startIndex, int32 count):</td>
</tr>
<tr>
<td>• this._size /= 0 implies count &gt;= 0</td>
</tr>
<tr>
<td>• this._size /= 0 implies count &lt;= (startIndex + 1)</td>
</tr>
<tr>
<td>• this._size /= 0 implies not (startIndex == this._size)</td>
</tr>
<tr>
<td>• this._size /= 0 implies not (startIndex &lt; 0)</td>
</tr>
<tr>
<td>void ReadOnlyArrayList::Insert(int32 index, object obj):</td>
</tr>
<tr>
<td>• 0</td>
</tr>
<tr>
<td>object IListWrapper/ILListWrapperEnumWrapper::get_Current():</td>
</tr>
<tr>
<td>• this._firstCall = 0</td>
</tr>
<tr>
<td>• this._remaining &gt;= 0</td>
</tr>
<tr>
<td>int32 BinarySearch(int32 index, int32 count, object value, IComparer c):</td>
</tr>
<tr>
<td>• count &gt;= 0</td>
</tr>
<tr>
<td>• (this._size - index) &gt;= count</td>
</tr>
<tr>
<td>• not (index &lt; 0)</td>
</tr>
</tbody>
</table>

Note that due to optimizations in the precondition extraction there are fewer precondition clauses than code paths that end at an exception basic block (see table 5.9 on the preceding page). Also note that the extracted preconditions are not compilable Eiffel code, for example: Eiffel identifiers cannot start with an underscore, this is Current in Eiffel and there is no infix “/= |” feature in standard Eiffel classes.

The most apparent limitation of the extracted preconditions is that the numeric value zero, the boolean value false and the reference value null are all represented by zero. This is due to the expressions not carrying type information and the VES using integers to represent boolean values.

The precondition clause this /= 0, where /= means “unequal or unordered” (see also section 5.3 on page 74), states that the current object this should not be null. This clause only appears for static methods, where this can be null.

An often occurring kind of precondition clauses is concerned with argument indices. The 36 clauses 0 should be read as “false”, which simply means that the corresponding method should not be called at all—an exception would be raised immediately (e.g., see ArrayList::ReadOnlyArrayList::Insert in table 5.9).

The four precondition clauses for ArrayList::LastIndexOf in table 5.9 show some of the flexibility of the current implementation. The four clauses impose restrictions on the arguments startIndex and count only if this._size is not equal to zero.
Table 5.15: Extracted precondition clauses and their number of occurrence in Stack

<table>
<thead>
<tr>
<th>Precondition Clause</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>this._size</td>
<td>/=</td>
</tr>
<tr>
<td>this._version = this._stack._version</td>
<td>2</td>
</tr>
<tr>
<td>array</td>
<td>/=</td>
</tr>
<tr>
<td>this</td>
<td>/=</td>
</tr>
<tr>
<td>this._index</td>
<td>/=</td>
</tr>
<tr>
<td>this._index</td>
<td>/=</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>8</strong></td>
</tr>
</tbody>
</table>

Table 5.16: Extracted precondition clauses and their number of occurrence in Queue

<table>
<thead>
<tr>
<th>Precondition Clause</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>this._version = this._q._version</td>
<td>2</td>
</tr>
<tr>
<td>this._size</td>
<td>/=</td>
</tr>
<tr>
<td>array</td>
<td>/=</td>
</tr>
<tr>
<td>this</td>
<td>/=</td>
</tr>
<tr>
<td>this.currentElement</td>
<td>/=</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>7</strong></td>
</tr>
</tbody>
</table>

Relevance of the Extracted Preconditions

The relevance of the extracted preconditions is dependent on whether the precondition exhibits client verifiable properties or details internal to the implementation. For example, this._index |/=| -2 and this._index |/=| -1 from the public method Stack/StackEnumerator::get_Current indicate that the nested class StackEnumerator of class Stack (see table 5.9) uses two negative values assigned to the private field _index to store some state of the enumerator. An examination of the CIL code reveals that a value of -2 encodes “not started” and a value of -1 encodes “finished”. The client however cannot access the field _index. The precondition is therefore not relevant to the client, it is only relevant to the implementation of StackEnumerator itself.

Another problem with preconditions from StackEnumerator is the fact that the class itself is private and clients of the library are likely to request an object of type IEnumerable from an instance of ArrayList without ever knowing about the existence of StackEnumerator. In such cases the extracted preconditions would have to be associated with client accessible superclasses or superinterfaces, if they are verifiable by clients of the corresponding interface or class.
Chapter 6

Assessment

The current approach and implementation for contract extraction is only a first step towards a more general solution for automatic contract elicitation. The following sections discuss the limitations, outline possible solutions and make proposals for future developments.

6.1 Limitations

Scanner and Parser

The application of the contract extraction tool to a module currently requires the disassembly of the binary module with ildasm [16] into the textual representation of the CIL. This is inefficient and could be resolved by the development of an alternative parser that either reads a binary module directly or uses a library that provides this functionality. Some research into the existence of such a library has been conducted at the beginning of this project. However, the known developments of such libraries were in early stages.

AST

The implementation of the AST (chapter 4 on page 18) is still very incomplete. It however was sufficient for a first approach to the problem of contract extraction. An important restriction with respect to contract extraction is the current lack of support by the AST classes for protected regions of code (see section 4.5 on page 26). Protected regions are therefore unsupported by the contract extraction algorithm. A corresponding extension to the current implementation of contract extraction would potentially have to deal with explicitly thrown exceptions in a try-block that are caught and possibly rethrown by one of the handlers following the try-block. Such usage of the exception handling mechanism should however be rare.

Expressions

Expressions (section 5.3 on page 74) are currently not typed, it is therefore not possible to query an expression about its result type. This imposes restrictions on the simplification of expressions and computation of textual representations.
of expressions in languages like Eiffel (e.g., the current representation of zero, false and null is 0, see also section 5.9 on page 96).

A possible solution to this is to add a query type to CIL_EXPRESSION and its operands. A particular expression class could then impose restrictions on its operand types, logical inversion of a relational expression could be simplified to an appropriate relational expression and the computation of textual representations in other languages would become more flexible. Note that the type of a terminal expression will have to be derived from the terminal source's type (e.g., a field of type int16 will have the type int32 on the evaluation stack, see [8, section 12.3.2.1]).

The computation of a string representation from an expression is in an early state. The availability of types for expressions will assist in the translation from CIL expressions to Eiffel expressions. The aim is an algorithm that is able to produce compilable Eiffel contracts from CIL expressions.

Classification Scheme

While the classification of basic blocks (see section 5.5 on page 88) itself does not have immediate limitations it reveals some of the limitations of the other parts of the implementation. Currently code paths with instructions storing to parameters, locals or fields or calling other methods are not considered by this first approach. There are also other less often occurring instructions that render a code path intractable for the current approach (see CIL_BASIC_BLOCK_CLASSIFICATION_VISITOR).

Symbolic Execution

The current implementation of the symbolic execution (see section 5.7 on page 91) of branch conditions is unfinished. The class CIL_SYMBOLIC_EXECUTION_VISITOR would have to redefine most visit commands and the expression cluster needs to be extended with expression classes corresponding to the additionally executed instructions.

Symbolic execution is currently limited to a single basic block. A branch condition could however be computed by several basic blocks. For example, two basic blocks compute a subexpression of the branch condition and their common successor computes the complete branch condition. In this case there would effectively be (at least) two branch conditions, depending on which code path is considered.

Current symbolic execution is restricted to extract branch conditions. A possible extension would be to extract expressions stored in parameters, locals and fields, which would be a first step towards supporting code paths containing “store” instructions. Other extensions would be to extract return values for postconditions or parameter expressions of method calls. The support for handling calls to methods is however expected to require a lot more infrastructural work and should be considered very carefully. A possible first approach might be to assume a closed system and first determine all implementations of a called method (there can be several implementations in the case of virtual calls) and if there is only one implementation determine if that implementation can be considered to be a pure function (i.e., it does not change the systems state). Only in this case would a first approach attempt to extract the method call’s
return expression if any and use that expression for further computation—all other cases would still be considered to be intractable.

**Precondition Extraction**

The current implementation of the algorithm for precondition extraction infers preconditions of a routine by assuming that an explicit exception implies a precondition. In order to extract an expression for such a precondition the code paths finishing in an exception are elicited and the branch conditions on each path are combined to form a precondition clause. This approach could be extended to associate intermediate preconditions not only to basic blocks but also to individual instructions. For example, loading a field from an object requires the corresponding object reference to be non-null, in this case an expression for the object reference would have to be extracted by the symbolic execution.

**6.2 Future Work**

**Preconditions**

The current precondition extraction algorithm proceeds in associating preconditions to basic blocks where an exception basic block is associated with a precondition of false and other basic blocks are assigned preconditions derived from their successor basic blocks.

An extension to this would be to associate preconditions to instructions directly. For example, loading of a field or a call to a method requires that the object reference on which this is done is not null. The symbolic execution algorithm would have to be extended for extracting an expression for such object references.

**Postconditions**

When considering postconditions and possible extensions of the current implementation for contract extraction two kinds of postconditions are of immediate interest: postconditions involving the return value of a method if any and postconditions involving fields.

Extracting postconditions for the result of a method can be achieved by reusing the algorithm for symbolic execution. The algorithm currently extracts branch conditions by computing an expression for the corresponding branch instruction, which derives from the one or usually two items on top of the evaluation stack before the branch instruction. The result of a method can be extracted by applying the same algorithm for computing an expression for the item on the evaluation stack before any return instruction. If there is more than one return instruction (or different code paths leading to different expressions on the evaluation stack before one return instruction) the branch conditions can be used to form postcondition clauses similarly to the currently extracted preconditions.

As the current implementation ignores code paths with instructions that store in locals, arguments and attributes, an extension for extracting postconditions on attributes has to lift that restriction at least partially. The symbolic execution would have to extract an expression for the value stored in a field at any
store instruction. A postcondition extraction algorithm would then—similarly to the current precondition extraction—form postcondition clauses from branch conditions and expressions stored in fields.

Class Invariants
As class invariants, unlike routine preconditions and postconditions, assert certain conditions for a whole class, the current focus of the contract extraction algorithm on a single method is no longer sufficient. A corresponding algorithm has to take most or all features of a class into account in order to elicit a class invariant.

An algorithm for eliciting invariants from a .NET class could start by extracting postconditions from methods. If, for example, a method without arguments ensures a certain postcondition for its return value, this postcondition can be moved to the class invariant and serve as part of the method’s definition.

A common type of invariants asserts that some field is not null. Such invariants could be extracted by observing that all constructors ensure that the field is not null and all methods implicitly or explicitly do the same. This approach could then be extended to more elaborate assertions. For example, if a constructor ensures some property of a field, an extended algorithm can check if other constructors and methods do the same. However, note that fields that are exported to clients—of which the class invariants would be interesting to the user—can also be set by the clients.
Chapter 7

Conclusion

The developed tool for contract extraction from .NET methods involves a scanner and a parser for creating an abstract syntax tree from the CIL source code of a .NET module and several algorithms for analyzing a method’s implementation and extracting preconditions from explicit exception cases.

The chosen approach is limited to code paths whose instructions effect only the evaluation stack of the virtual execution system. However, an analysis of ArrayList, Stack and Queue revealed that, in these classes, half or more of all exception code paths can be addressed despite this limitation.

The results of the current approach are very encouraging, various proposals towards the extension of the present approach and implementation have been made.
Appendix A

Glossary

**AST:** An Abstract Syntax Tree is a syntactical representation of a corresponding source code of a particular language. In a simple explanation inner nodes represent operators and leaf nodes represent operands. In the context of the CIL, an AST associates arguments with instructions and groups instructions to methods, methods and other members to classes and classes, together with other declarations, to assemblies.

**CLI:** The Common Language Infrastructure, as standardized in [12], defines an environment in which applications can be executed. Neither the applications nor the language they are written in have to consider the unique characteristics of the underlying platform, it is sufficient for them to target the platform specified by the CLI.

**CIL:** The Common Intermediate Language is part of the CLI and specifies the instructions available in the VES in terms of the machine state the VES defines.

**ECMA:** ECMA International is an industry association dedicated to the standardization of information and communication systems. It standardized the CLI.

**VES:** The Virtual Execution System is part of the CLI and defines a hypothetical machine with a corresponding machine model and state. Its main purpose is to support the execution of the CIL instruction set.
Bibliography


