AbsInt Advanced Analyzer for TriCore

User Documentation

AbsInt Angewandte Informatik GmbH
info@AbsInt.com
www.AbsInt.com

a^3 Version:	10.08i
a^3 Build:	146992
Manual Release:	00089
Date:	September 1, 2010
Reference:	a3-TriCore-manual-00089-absint

Copyright notice:
© AbsInt Angewandte Informatik GmbH

All rights reserved. This document, or parts of it, or modified versions of it, may not be copied, reproduced or transmitted in any form, or by any means, or stored in a retrieval system, or used for any purpose, without the prior written permission of AbsInt Angewandte Informatik GmbH.

The information contained in this document is subject to change without notice.

LIMITATION OF LIABILITY:

Every effort has been taken in manufacturing the product supplied and drafting the accompanying documentation.

AbsInt Angewandte Informatik GmbH makes no warranty or representation, either expressed or implied, with respect to the software, including its quality, performance, merchantability, or fitness for a particular purpose. The entire risk as to the quality and performance of the software lies with the licensee.

Because software is inherently complex and may not be completely free of errors, the licensee is advised to verify his work where appropriate. In no event will AbsInt Angewandte Informatik GmbH be liable for any damages whatsoever including – but not restricted to – lost revenue or profits or other direct, indirect, special, incidental, cover, or consequential damages arising out of the use of or inability to use the software, even if advised of the possibility of such damages, except to the extent invariable law, if any, provides otherwise.

AbsInt Angewandte Informatik GmbH also does not recognize any warranty or update claims unless explicitly provided for otherwise in a special agreement.

Contents

1 Introduction and Overview **11**

1.1 Purpose . . . 11
1.2 Availability . . . 12
1.3 License Key . . . 12
1.4 Legal Issues . . . 12
1.5 Location of Temporary Files . . . 12
1.6 Overview of Documentation . . . 13

2 Invocation of a^3 **14**

2.1 Operation Modes . . . 14
2.2 Configuration Files . . . 15

3 The User Interface of a^3 **17**

3.1 The `Welcome` View . . . 18
3.2 `Help` Menu . . . 19
3.3 `Project` Menu and Project Tool Bar . . . 19
3.4 The **`Configuration`** Views . . . 21
3.4.1 The `Files` View . . . 22
3.4.2 The `TriCore` View . . . 23
3.4.2.1 `TriCore` Analyzer Options . . . 23
3.4.2.2 Exporting, Importing, and Loading Machine Settings . . . 25
3.4.2.3 Configuration via Hardware Registers . . . 25
3.4.2.4 The `General` Tab of the `TriCore` View . . . 26
3.4.2.5 The `Flash` Tab *(Timing Analysis)* . . . 26
3.4.2.6 The `EBU` Tab *(Timing Analysis)* . . . 27
3.4.3 The `Analyses` View . . . 28
3.4.3.1 Reporting (General Tab) . . . 29

3.4.3.2 XML Reporting (General Tab) . . . 29
3.4.3.3 Source Code (General Tab) . . . 29
3.4.3.4 All Analyses (General Tab) . . . 30
3.4.3.5 Cache Analysis (General Tab) . . . 30
3.4.3.6 Pipeline Analysis (General Tab) . . . 31
3.4.3.7 Path Analysis (General Tab) . . . 32
3.4.3.8 The Advanced Tab . . . 33
3.4.3.9 The Support Tab . . . 35
3.4.4 The `Registers` View . . . 35
3.4.5 The `Source files` View . . . 35
3.4.5.1 The `Includes` Panel . . . 35
3.4.5.2 The `Replacements` Panel . . . 36
3.4.5.3 Proposals for Includes and Replacements . . . 36
3.4.6 The `Visualization` View . . . 37
3.5 The **`Analyses`** Views . . . 39
3.5.1 The `Create` View . . . 39
3.5.2 The Analysis Views . . . 40
3.5.3 Commands for Analyses . . . 41
3.5.4 The `Overview` View . . . 42
3.5.5 Start-Point Selection . . . 43
3.5.6 The Disassembly View . . . 45
3.6 The **`Information`** Views . . . 47
3.6.1 The `Messages` View . . . 47
3.6.2 The `Symbols` View . . . 47
3.6.3 The `Source files` View . . . 48
3.6.4 `Variables` and `Functions` Views . . . 48
3.6.5 The `Sections` View . . . 49

4 Background Information **50**
4.1 Loop Transformation . . . 50
4.1.1 Simple Loops . . . 50
4.1.2 Numbering of Loop Routines . . . 52
4.1.3 Loops with Several Exits . . . 52
4.1.4 Loops with Several Entries . . . 54

4.2 Routines . . . 54
4.3 Calling Contexts . . . 55
4.3.1 Contexts of Routines (without Loops and Recursion) . . . 56
4.3.2 Contexts of Routines (with Loops and Recursion) . . . 58
4.4 Analysis Phases . . . 60
4.4.1 Loop Bound Analysis . . . 61
4.4.2 The Actual Value Analysis . . . 62
4.4.3 Cache and Pipeline Analysis . . . 62
4.4.4 Path Analysis . . . 63
4.5 Locating Source Files . . . 64

5 AIS Annotations **66**
5.1 Overview . . . 67
5.2 Clock Rate *(Timing Analysis)* . . . 68
5.3 Naming the Compiler . . . 69
5.4 Context Specification *(Timing Analysis, Value Analysis)* . . . 69
5.5 Additional Start Points for Timing Analysis . . . 71
5.6 Declaring Routine Entries . . . 73
5.7 Stop Decoding . . . 73
5.8 Control-Flow Specifications . . . 75
5.8.1 Instruction Classes . . . 75
5.8.2 Targets of Computed Calls and Branches . . . 76
5.8.3 Indirect Calls or Branches via Arrays of Function Pointers or Switch Tables 78
5.8.4 Target Check . . . 82
5.8.5 Properties of Call Instructions . . . 83
5.8.6 Properties of Routines . . . 83
5.9 Addresses of Memory Accesses *(Timing Analysis, Value Analysis)* . . . 84
5.9.1 Local Address Specifications . . . 84
5.9.2 Local Address Specifications for a Base Register . . . 86
5.9.3 Global Address Specifications . . . 87
5.9.4 The Evaluation of Address Specifications . . . 89
5.9.5 Alignment Specifications . . . 90
5.10 Specification of Register Values and Definition of User Registers . . . 91
5.11 Assertions of Conditions . . . 95

5.12 Properties of Memory Areas . . . 96
5.12.1 Area Definitions . . . 97
5.12.2 Access Properties . . . 98
5.12.3 Volatile Memory Areas . . . 98
5.12.4 Content Properties . . . 98
5.12.5 Data Values in Memory . . . 100
5.12.6 Copied Areas . . . 103
5.13 Additional Execution Time *(Timing Analysis)* . . . 104
5.13.1 Note on Backward Compatibility . . . 106
5.14 WCET Contribution of Code Snippets *(Timing Analysis)* . . . 107
5.14.1 Note on Backward Compatibility . . . 108
5.15 Not-Analyzed and External Code Snippets . . . 108
5.15.1 Declaration of Not-Analyzed and External Routines . . . 109
5.15.2 Not-Analyzed Code Snippets That Are Not Routines . . . 109
5.15.3 Execution Time of Excluded Snippets *(Timing Analysis)* . . . 111
5.15.4 Stack Usage of Excluded Snippets *(Stack Analysis)* . . . 116
5.15.5 Effects of Excluded Snippets *(Both Analyses)* . . . 117
5.15.6 Cache and Pipeline Flushes *(Timing Analysis)* . . . 118
5.15.7 Summary . . . 119
5.16 Calling Conventions . . . 119
5.17 Infeasible Code . . . 120
5.17.1 Declaration of Unconditional Infeasibility . . . 120
5.17.2 Declaration of Conditional Infeasibility . . . 120
5.17.3 Assertions about Infeasibility *(Value Analysis)* . . . 121
5.18 Values of Conditions . . . 121
5.18.1 Unconditional Values of Conditions . . . 122
5.18.2 Conditional Values of Conditions . . . 122
5.19 Recursion . . . 123
5.19.1 Maximum Call Depth *(Stack Analysis)* . . . 123
5.19.2 Maximum Number of Calls *(Timing Analysis, Value Analysis)* . . . 125
5.19.2.1 Direct Recursion . . . 125
5.19.2.2 Simple Direct Recursion . . . 126
5.19.2.3 General Recursion . . . 127
5.19.2.4 Implementation Restriction . . . 127

5.20 Loop Bounds . . . 128
5.20.1 Local Loop Bound Specifications . . . 129
5.20.2 Examples with Parametric Loop Bounds . . . 132
5.20.3 Global Loop Bound Specifications . . . 134
5.21 Timing Specifications for Loops *(Timing Analysis)* . . . 135
5.22 Flow Constraints *(Timing Analysis)* . . . 136
5.22.1 Linear Flow Constraints . . . 137
5.22.2 Ratio Flow Constraints . . . 138
5.22.3 Constraints and Contexts . . . 139
5.23 Output from Value Analysis . . . 140
5.23.1 Routine Callers . . . 140
5.23.2 Memory Accessors . . . 141
5.23.3 Control-Flow Successors . . . 141
5.23.4 Output of Register and Memory Contents . . . 141
5.24 Output from Timing Analysis . . . 141
5.24.1 Counting Accesses to Some Memory Area . . . 141
5.25 Symbolic Names for Program Points and Areas . . . 142
5.26 Avoiding Problems with Unreachable Program Points . . . 143
5.27 Include Directives . . . 144
5.28 Expressions and their Evaluation . . . 144
5.28.1 Priorities and Grouping . . . 145
5.28.2 Results of Expressions . . . 146
5.28.3 Atomic Expressions . . . 147
5.28.4 Addresses of Program Points and Memory Areas . . . 147
5.28.5 Contents of Memory Cells . . . 148
5.28.6 Minimum, Maximum, and Interval Operations . . . 149
5.28.7 Rational Arithmetic . . . 152
5.28.8 Irrational Operations . . . 153
5.28.9 Rounding Operations . . . 155
5.28.10 Modulo Operations . . . 156
5.28.11 Relational Operators . . . 158
5.28.12 Logical Operators . . . 161
5.28.13 Switch Expressions . . . 163
5.28.14 The Result of Value Analysis . . . 164

5.28.15 The `before` Operator . . . 165
5.28.16 Types and Units . . . 166
5.28.17 The Interpretation of the Values of Overall Expressions . . . 167
5.29 Program Points and Targets . . . 168
5.30 Atomic Parts of AIS Annotations . . . 172
5.30.1 Keywords . . . 172
5.30.2 Names . . . 172
5.30.3 Integers . . . 173
5.30.4 Units of Addressing . . . 173
5.31 Source Code Annotations . . . 174
5.31.1 Translation of Source Code Lines into Code Addresses . . . 174

6 Requirements and Restrictions **177**
6.1 Hardware Design and System Configuration . . . 177
6.2 Sequentiality of the Analyzed Tasks *(Timing Analysis)* . . . 177
6.3 Restrictions on the Analyzed Code . . . 178
6.4 User Annotations . . . 178
6.5 Requirements Specific to Stack Analysis . . . 179

7 The Analyses of a^3 **181**
7.1 Combined Call Graphs and Control-Flow Graphs . . . 182
7.1.1 Call Graphs . . . 182
7.1.1.1 Routines . . . 183
7.1.1.2 Info Fields at Routines . . . 183
7.1.1.3 Calling Relationships . . . 183
7.1.2 Basic Block Graphs . . . 184
7.1.2.1 Basic Blocks . . . 185
7.1.2.2 Info Fields at Basic Blocks . . . 185
7.1.2.3 Control-Flow Edges . . . 186
7.1.3 Instruction Sequences . . . 187
7.1.3.1 Instructions . . . 187
7.1.3.2 Info Fields at Instructions . . . 187
7.1.3.3 Edges Between Instructions . . . 188
7.1.4 Tail Calls . . . 188
7.1.5 Displaying Source Code in Basic Block Graphs . . . 188

7.2 Value Analysis . . . 190
7.2.1 The Tree of Contexts . . . 191
7.2.2 Registers and Memory Cells . . . 191
7.2.3 Creating AIS Annotations . . . 193
7.3 Stack Analysis . . . 195
7.3.1 Stack Analysis Results at Routines . . . 196
7.3.2 The Report File . . . 198
7.3.3 Stack Analysis Results at Basic Blocks and Instructions . . . 199
7.3.4 External and Not-Analyzed Routines . . . 200
7.3.5 Recursion . . . 204
7.4 Timing Analysis . . . 206
7.4.1 Overall Worst-Case Execution Time . . . 206
7.4.2 WCET Contributions and Contexts of Routines . . . 206
7.4.3 WCET Information for Basic Blocks . . . 211
7.4.4 Loop Count Information . . . 215
7.4.5 The Worst-Case Execution Path . . . 216
7.5 Graphs with Cache and Pipeline States . . . 217

8 The aiSee Graph Viewer 221
8.1 **aiSee** Basics . . . 221
8.1.1 Command Invocation . . . 221
8.1.2 Operation Modes . . . 222
8.1.3 Scrolling and Scaling . . . 222
8.1.3.1 Scrolling the Graph . . . 222
8.1.3.2 Scaling the Graph . . . 222
8.1.4 Selection of Graph Elements . . . 223
8.1.5 Show Panner . . . 223
8.1.6 Resetting **aiSee** . . . 223
8.1.7 Accessing Info Fields . . . 223
8.2 Exposing and Hiding Subgraphs . . . 224
8.2.1 The Hierarchy Levels of Control-flow Graphs . . . 224
8.2.2 Visualizing Subgraphs . . . 224
8.2.3 Switching between the Visual Representations of Subgraphs . . . 226
8.3 Find Nodes . . . 227

8.4 Show Neighbors 229

9 Typical Working Patterns **232**

9.1 Finding a Routine in an **aiSee** Picture 232

9.2 Finding an Instruction in an **aiSee** Picture 233

9.3 Finding the Place of Maximum Stack Usage 233

9.4 Avoiding State Explosion 234

Appendix: Document History **236**

Index **241**

Chapter 1

Introduction and Overview

1.1 Purpose

a^3 (pronounced *a-cube*) is an acronym for *AbsInt Advanced Analyzer*. It is a framework combining various analyzers and mechanisms for reporting their results textually and graphically. The analyzers typically operate on code snippets in fully linked executables. This manual describes the following features of a^3:

1. a^3 can compute the call graph of an application and the control-flow graphs of its routines. These graphs are visualized by a^3 and can be interactively explored. The visualization provides three levels of details: routines and their calling relationships, basic blocks and the control flow in-between, and individual assembly instructions (see section 7.1).

2. a^3 can calculate information about the values in registers and memory cells (*value analysis*). This information can be interactively explored (see section 7.2), and can be used to prove the absence of illegal memory accesses: the user specifies which memory accesses are admissible for a particular instruction, and a^3 tries to decide, based on its analysis results, whether this specification is met.

3. a^3 can determine upper bounds for the size(s) of the stack(s) in single routines and in the whole program (*stack analysis*, see section 7.3). These upper bounds can be used to verify the absence of stack overflow for a given setting of stack parameters, or to derive a parameter setting that avoids overflow and does not reserve too much memory for the stacks.

4. a^3 can determine upper bounds for the Worst-Case Execution Time (WCET) of code snippets in the application (*timing analysis*, see section 7.4). These upper bounds can be used to verify that real-time tasks meet their deadlines, and as inputs for an overall schedulability analysis.

All these analyses require separate licenses. Depending on your licenses, it is possible that some analysis is available and some other analysis is not, or all analyses are available.

The results of the analyses are written into a report file and added to the visualization as annotations to the routines, basic blocks, and instructions.

1.2 Availability

This documentation describes a^3 for TriCore, a member of the a^3 family. a^3 for TriCore can analyze and visualize fully statically linked executables for TriCore 1.3 processors that have been produced from C source code by the Tasking C Compiler or the GNU C compiler (GCC). (All examples in this manual refer to Tasking executables.) Your license does not necessarily support all these possibilities. While stack analysis is possible for all TriCore 1.3 executables, timing analysis is restricted to TriCore 1766, 1796, and 1797 chips.

There are a^3 versions for x86/Windows (Windows 2000 and later) and x86/Linux platforms. This documentation covers both operating systems.

This manual only describes a^3 for TriCore. Nevertheless, a^3 for TriCore will usually be abbreviated to a^3 in this document.

1.3 License Key

a^3 needs a license key to run properly. A license key enables you to use certain a^3 features on certain computers for a limited period of time (evaluation license) or for ever (commercial license). The a^3 features that are not supported by your license key are present in the menus of the graphical user interface, but cannot be activated. This includes such basic functionality as stack and timing analysis. The validity of the license key is also checked by certain components of a^3 so that it may happen that a^3 starts running without complaining, but refuses to complete certain operations.

The license key for a^3 is stored in a *license file*. License files are created and distributed by **AbsInt**. In order to obtain a license file, please contact support@absint.com. Information about the current license and support for handling license files can be found in the a^3 GUI under the menu entry `Help` → `License`.

1.4 Legal Issues

Since a^3 runs a disassembler, you must be equipped with a disassembling license for the executables that you wish to be analyzed and visualized by a^3.

1.5 Location of Temporary Files

When a^3 is running, it puts *temporary files* into a directory. If the checkbox `Keep temporary files in project directory` in the `Files` view is on (see section 3.4.1), the temporary files are put into the *project directory*, which resides in the same directory as the project file and is named *P*`.apd` if the project file is *P*`.apx`. If the checkbox is off, the files are put into a standard directory of which the name is determined as follows:

1. If the environment variable `TEMP` is defined and its value is the name of a directory, this directory is used for storing temporary files.

2. If not, a^3 considers the environment variable `TMP` in the same way.
3. If this is not successful either, variable `Temp` is considered.
4. If none of these variables yields a directory name, a^3 puts its temporary files into directory `/tmp` (Linux) or `C:\` (MS Windows).

If the directory determined in this way does not exist or does not permit the creation of files, a^3 will not work.

Remark for MS Windows users: Environment variables like `TEMP` can be examined and modified via the `Advanced` tab of the `System Properties` dialog box, which is accessed by right-clicking on the `My Computer` icon and selecting `Properties` from the resulting menu (Windows 2000).

1.6 Overview of Documentation

Chapter 2 describes the invocation of a^3.

Chapter 3 provides a detailed description of the user interface of a^3 in combination with a systematic explanation of all possible commands.

Chapter 4 presents some background information about the internal representation of loops (4.1), the origin and naming of routines (4.2), calling contexts of routines (4.3), phases of timing and stack analysis (4.4), and locating source files (4.5).

Chapter 5 introduces a kind of specification files, called *AIS files*. They may contain *declarations* that help the analyzer to understand the executable, thus providing information necessary to get analysis results at all, or to improve their precision. Besides declarations, the AIS language also comprises so-called *assertions* for value analysis. They assert requirements, for example, that a certain instruction must not access a particular memory area; based on the results of value analysis, these requirements might be possibly verified or disproven.

Chapter 6 lists some restrictions on the target configuration and the code to be analyzed.

Chapter 7 explains the combined call graphs and control-flow graphs produced by a^3, including the annotations produced by the various analyses. This comprises an introduction to the meaning of the features of the visual representations of graphs (meaning of boxes and arrows of various colors). Furthermore, section 7.2 illustrates how the results obtained by value analysis can be interactively retrieved for single instructions. Section 7.5 explains the combined cache and pipeline states displayed by a^3 upon request.

Chapter 8 contains a basic introduction to the graph viewer **aiSee**: how to navigate in graphs, how to access all the information contained in the combined call graph and control-flow graph, and how to switch between various hierarchy levels and their visual representations. For more details, see the separate **aiSee** manual.

Chapter 9 describes some patterns for working with a^3 to obtain useful information or to react to reported problems.

Chapter 2

Invocation of a^3

Section 2.1 shows how a^3 can be invoked leading to different operation modes. Configuration files are introduced in section 2.2.

2.1 Operation Modes

a^3 can be started in several ways, leading to different operation modes.

1. There is a generic a^3 launcher called `a3`. When started, it shows a list of available (i.e. installed) a^3 versions, from which a^3 for TriCore can be invoked. The launcher also shows a list of recently opened projects. Since the target architecture is encoded in the projects, a^3 for TriCore can also be started by clicking on a project set up for TriCore.

2. The launcher can also be restricted to a specific target. The command `a3 -l tricore` shows a list of available (i.e. installed) a^3 for TriCore versions, from which one can be selected.

3. A specific a^3 for TriCore version can be started with the command `a3 tricore` *`build`* where *`build`* is the build number of the desired version.

4. The most recent a^3 for TriCore version can also be started directly by writing `a3 tricore` on a command line or by other standard means like clicking on the appropriate icon. Since no project file is given, the options in the user interface of a^3 for TriCore have empty or default values. In this situation, you may either set the options manually, or load a project file that contains option values saved in an earlier session.

5. Alternatively, you may supply a^3 for TriCore with a project file right from the start by calling one of

   ```
   a3  project
   a3  build  project
   ```

   from a command line, or by double-clicking the project file, or by dragging the project file with the mouse to the icon of a^3 for TriCore on the desktop. The target `tricore` need not

be specified since it is encoded in the project file, but can be added for clarity. After the call, the graphical user interface of a^3 for TriCore is started for interactive working. The contents of the given project file become the values of the options of a^3 for TriCore. Provided that the project file contains a complete and consistent set of option values, you may immediately start an analysis or visualization for this project.

6. a^3 runs in *batch mode* if it is called with parameter **`-b`**

   ```
   a3  project  -b
   a3  build  project  -b
   ```

   from a command line. Then a^3 performs all analyses specified in the project file without starting or showing a graphical user interface. The results of the analyses can later be found in the project files generated by a^3.

   A batch mode call may be given a timeout by adding the command line option `--timeout` *`seconds`*.

7. a^3 runs in *interactive batch mode* if it is called with parameter **`-B`**

   ```
   a3  project  -B
   a3  build  project  -B
   ```

   from a command line. Then a^3 performs all analyses specified in the project file like in batch mode. The graphical user interface is started with its window minimized. This means there is no open window on the desk top, only an entry in the control bar. The window can be opened by clicking on this entry.

   If the analysis started automatically terminates successfully, a^3 exits automatically. In this case, it runs without any user interaction. If the automatic analysis must be aborted because of errors, a^3 does not terminate, but displays a window to allow for human inspection.

   An interactive batch mode call may be given a timeout by adding the command line option `--timeout` *`seconds`*.

8. The above calls in batch mode (`-b` or `-B`) cause all analyses specified in the project file to be executed. Specific analyses can be executed by adding **`-i`** *ID* where *ID* is the ID of the analysis to be executed. Analysis IDs can be specified in the analysis views (section 3.5.2) and are shown in the `Overview` view (section 3.5.4).

   Several different `-i` options are possible to cause the execution of several different analyses. Note that no `-i` option does not mean no analysis, but *all* analyses.

2.2 Configuration Files

In any mode, a^3 can be instructed to read a *configuration file* by writing an additional option

```
-c FileName
```

on a command line. Calling a^3 with an explicit configuration file is particularly useful if a^3 is run in batch mode. The following is an example of a minimal configuration file containing only path names to locations in the file system (with Linux file names):

```
[General]
BinDir=/scratch/a3tests/linux64/a3_tricore/libexec
ShareDir=/scratch/a3tests/linux64/a3_tricore/share
DocDir=/scratch/a3tests/linux64/a3_tricore/doc
LicenseFile=/scratch/a3tests/share/license.dat
```

- **`BinDir`** is the directory containing the ancillary executables of a^3 (decoder, various analyzers, viewer etc.).
- **`ShareDir`** is a directory containing data files controlling the operation of a^3.
- **`DocDir`** is a directory containing online manuals.
- **`LicenseFile`** is the full path name of the license file to be used by a^3 (see also section 1.3).

If the configuration file is writable, it is written when a^3 terminates its execution. It will then contain information on the shape, size, and position of the window of the graphical user interface, and various “preferences” set in the user interface.

Chapter 3

The User Interface of a^3

The user interface of a^3 is centered around various *views* with different purposes. Figure 3.1 shows an example. The white box on the left lists the available views (`Welcome`, `Files`, etc.), structured into sections such as **a^3 TriCore**, **Configuration**, etc. Here you can select a view, which is then displayed in the main part of the GUI window. In Figure 3.1, the `Files` view in the **Configuration** section is selected (indicated by highlighting in the list of views), and consequently displayed in the main part of the GUI (indicated by the heading **Configuration – Files** above the main part).

Navigation through the list of views is possible with the arrow keys in connection with the `Alt` key. `Alt`+↑ moves the selection one up in the list and `Alt`+↓ moves the selection one down. There is also a history of visited views like in typical web browsers. `Alt`+← moves backward in history, i.e. switches to the previously selected view, and `Alt`+→ moves forward in history.

The list of available views is not static, but extensible. It can be extended by the user by specifying analyses, and it is extended by a^3 itself: when analyses are run, new views are created for displaying analysis results. These additional views can be removed again by the user. Figure 3.1 shows a minimal list of views, which appears when a^3 is started without a project file.

The top of the a^3 GUI shows a list of menus as usual. Below this list, there are some tool bars with buttons that correspond to the most important menu items, offering direct access to them. Figure 3.1 shows two menus, `Project` and `Help`, and the `Project` tool bar exhibiting some entries from the `Project` menu. This is the minimal number of menus. Some views come with additional menus and tool bars to trigger operations special to those views. In any view, some of the available tool bars can be hidden by changing the settings in a context menu invoked by a right mouse click into the areas containing the menus and the tool bars. The tool bars can also be moved around by mouse dragging.

The title bar of the a^3 window displays the target architecture and the full path name of the project file that is currently being processed. The project file name is missing if no project file has been specified. When some option setting is modified, an asterisk $\star$ is appended to the text in the title bar. The asterisk disappears when the current option settings are saved in a project file.

In the sequel, we present the `Welcome` view (section 3.1), the two pervasive menus `Help` (section 3.2) and `Project` (section 3.3), and then the remaining views with their associated menus and tool bars – see section 3.4 for views under the heading **Configuration**, section 3.5 for

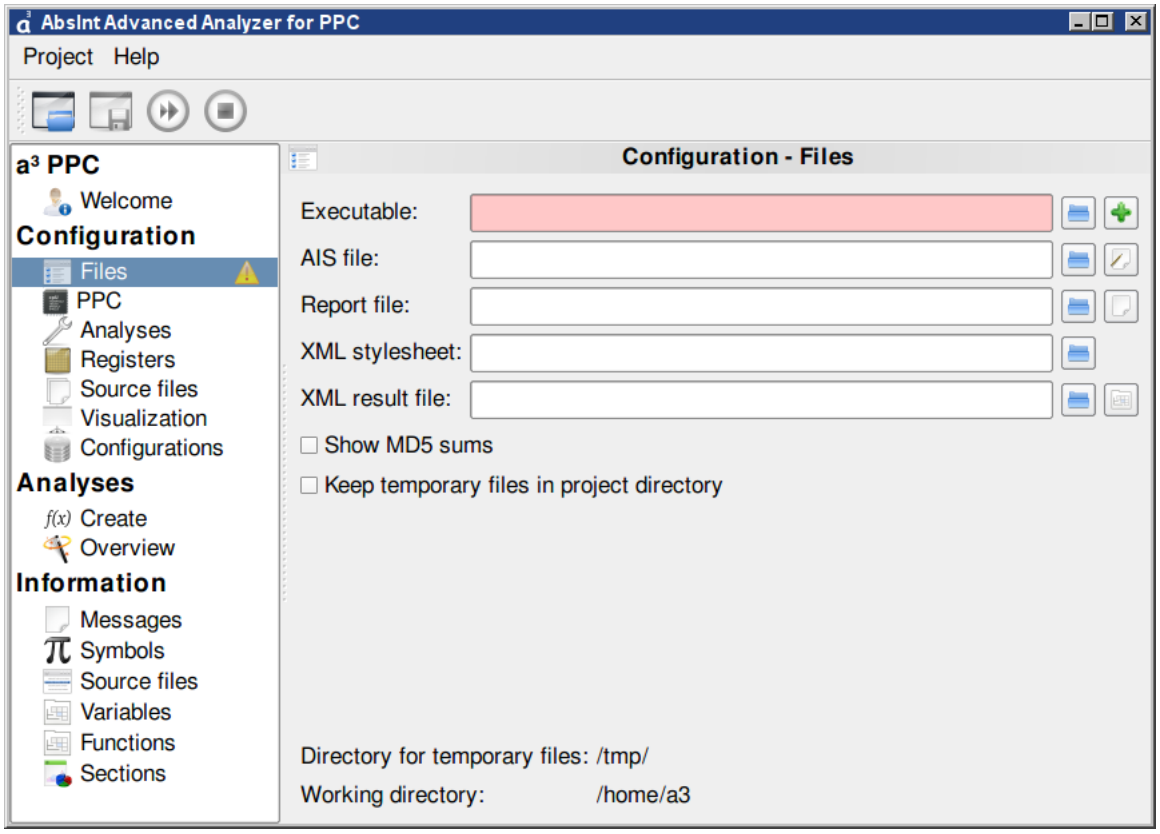


Figure 3.1: The user interface of a^3

views under the heading **`Analyses`**, and section 3.6 for views under the heading **`Information`**.

3.1 The `Welcome` View

The `Welcome` view offers quick access to various functionalities:

- Open a^3 projects;
- View documentation on a^3;
- View examples;
- Contact **AbsInt**.

The `Welcome` view can be closed by clicking on the button in its upper right corner. This operation removes the `Welcome` view from the list of available views. It can be reopened by the `Help` → `Welcome` command (see section 3.2).

3.2 `Help` Menu

The **`Help`** menu gives access to information about a^3.

- **`Welcome`**
  adds the `Welcome` view to the list of available views and lets a^3 switch to this view (see section 3.1).
- **`Manual`**
  opens the online version of the manual describing a^3 for TriCore.
- **`AIS Quick Reference`**
  opens an online version of the AIS Quick Reference. AIS is the annotation language of a^3. A detailed description of AIS is given in chapter 5 of the a^3 manual (the one you are reading now).
- **`aiSee Manual`**
  opens the online version of the manual describing **AbsInt**'s graph viewer **aiSee**, which is used to display the graph pictures generated by a^3. A short **aiSee** description specialized to the graph pictures of a^3 is provided in chapter 8 of the a^3 manual.
- **`License...`**
  opens a dialog box with information about the current license and support for handling license files.
- **`Version...`**
  opens a window containing version information about a^3 and all its ancillary tools.
- **`About...`**
  opens a window containing version information about a^3 itself, and contact information about **AbsInt**.

3.3 `Project` Menu and Project Tool Bar

The `Project` menu offers the following commands:

- **`New project`**
  puts a^3 into the state without current project file. This is the same state as the initial state of a^3 when it has been started without project file. All options have their default values; for the text fields containing file names this means they are empty.
- **`Open project...`** or **`Ctrl+O`** or in the project tool bar
  opens a file selection box where you can select a project file that is loaded when the **`Open`** button of the selection box is clicked. The contents of this project file replace the current option settings. The file becomes the new current project file whose name is displayed in the title bar.

- **`Open file`**
  lets you select between three kinds of files: AIS files, GDL files, and Text files.

  AIS files are parameter files with additional information for a^3 (see chapter 5). Opening an AIS file with this command means opening a new view with an editor for this AIS file. It does not mean that the AIS file becomes part of the project. For that purpose, AIS file names must be specified in the `Files` view or in an analysis view. Editing these project AIS files can be initiated by clicking buttons in those views. Thus, the main purpose of the `Open file` command is to open AIS files from other projects to compare them with the project AIS files, or to copy their contents into the project AIS files.

  GDL files contain graph pictures like those generated by a^3. Opening a GDL file with this command means opening a new view with a graph browser. It does not mean that the GDL file becomes part of the project. For that purpose, GDL file names must be specified in an analysis view. Whenever a^3 generates a graph during its work, it automatically opens a graph-browser view for it. Thus, the main purpose of the `Open file` command is to open GDL files from other projects to compare their contents with the graphs generated from the project.

- **`Recent Projects`**
  provides access to a list of recently opened project files. When you select a file from this list, its contents replace the current option settings. The file becomes the new current project file whose name is displayed in the title bar.

- **`Save project`** or in the project tool bar
  saves the current option settings in the current project file whose name is displayed in the title bar. This operation is disabled if the option settings in the GUI have not been changed since the last project file loading or `New project` operation.

- **`Save project as...`**
  opens a file selection box where you can select the name of an existing project file, or type in a new project file name. When the **`Save`** button of the selection box is clicked, the current option settings are stored in a file with the chosen name. This file becomes the new current project file whose name is displayed in the title bar.

- **`Export`**
  causes a^3 to export its input files (project file, executables, and AIS files) to a specific place. `Export` leads to a submenu with two choices, `To directory` and `To archive`. In the first case, a^3 asks for a directory where it places the files. In the second case, it asks for the name of a `.zip` file into which the files are put. Within the `.zip` archive, the files are placed in a directory with the same name as the `.zip` file (minus the suffix `.zip`).

- **`XTC`**
  provides access to XTC (XML Timing Cookies), an interface to tools of partner companies of **AbsInt** (see www.absint.com/xtc/). Currently, only the interface to the T1 tool of Gliwa GmbH is built in the a^3 GUI. The T1 tool can determine loop bounds by measurements. After running a timing analysis that complains about missing loop bounds, the submenu entry `Export T1 request` becomes active and can be used to write an `.xtc` file requesting

the missing loop bounds. This `.xtc` request file causes the T1 tool to measure the missing loop bounds and to produce an `.xtc` response file, which can be imported into a^3 by the submeny entry `Import T1 response`.

`Export T1 request` is not active before a timing analysis is performed, or after a timing analysis with known loop bounds, or after a timing analysis whose unbound loops all have multiple entries or exists; such loops cannot be handled by the T1 connection.

- **`Start all analyses`** or **`F6`** or in the project tool bar
  starts all analyses in the current project. The same happens without a user command when a^3 is started in batch mode (with option `-b` or `-B`, see section 2.1). Section 3.5 describes the definition and management of analyses.

- **`Stop analysis`** or **`Ctrl+Esc`** or in the project tool bar
  stops the currently running analyses. This applies to the bunch of analyses started by the `Start all` command described above, and to single analyses started from analysis views (see section 3.5.3).

- **`Preferences...`** or **`Ctrl+,`**
  opens the `Preferences` dialog box. This box offers several views for setting preferences:

  - `Messages View` defines preferences for the messages produced by a^3, including colors to distinguish between errors, warnings, etc.
  - `Text Editor` influences the text editor of a^3, which is used for editing AIS files (chapter 5) and displaying source code (font, auto completion).
  - `Disassembly` influences the presentation of disassembly code (font, colors, relative addresses – see section 3.5.6).
  - `Rendering` refers to the rendering of graphs in the graph browsers of a^3 (anti-aliasing).
  - `Interface` refers to the user interface in general (tool bars, tool tips).
  - `Shortcuts` can be used to define and modify key-code shortcuts for operations in the graph browsers of a^3 (chapter 8).

  The `Preferences` dialog box is also accessible from the context menus of the views it refers to (messages views, AIS editors, source-code views, disassembly-code views, graph-browser views).

- **`Quit`** or **`Ctrl+Q`**
  causes a^3 to terminate. Another way to terminate a^3 is to click on the close button of its window frame.

3.4 The `Configuration` Views

This section describes the various views under the heading **`Configuration`**.

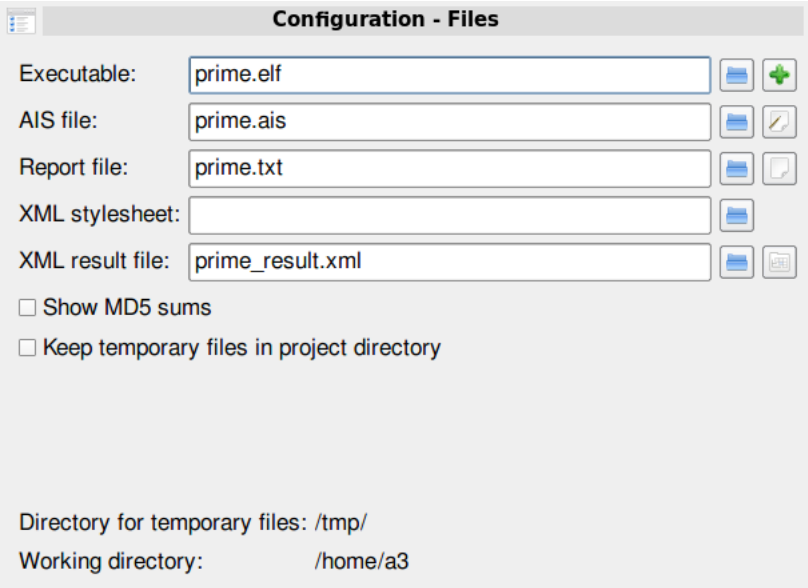


Figure 3.2: The `Files` view of a^3

3.4.1 The `Files` View

Global files to be used in all analyses can be specified in the `Files` view of the `Configuration` section. (Local files specific to an analysis can be specified in the analysis views, see section 3.5.2.) The `Files` view is depicted in Figure 3.2.

- The **`Executable`** text fields show the full path names of the executables that contain the code to be analyzed. Initially, there is only one `Executable` text field. Additional `Executable` text fields can be created by clicking on the button. An executable is selected by typing its name in the text field, or by using the button to the right of the text field, which opens a file selection dialog box.

- **`AIS file:`**
  Here you may provide the full path name of an AIS file that is used by all analyses. AIS files may contain various kinds of specifications that help a^3 in its work (see chapter 5). The button opens a view with a text editor on the AIS file.

- **`Report file:`**
  Here you may enter the full path name of a report file. If the panel specifies a file, then a^3 writes a report with messages and analysis results into this file whenever an analysis is performed. The report file is also written during the batch mode analyses performed when a^3 is started with the command line option `-b` or `-B` (see chapter 2.1).

  The contents of the report file are deleted before a new report is written.

  The button opens a view with a text browser on the report file.

- **`XML stylesheet:`**
  In this panel, you can specify an XML stylesheet to be used when displaying the local XML

report files produced by specific analyses (see section 3.5.2). (The stylesheet is not used for the display of the XML result file.)

- **`XML result file:`**
  If this panel specifies a file, then a^3 writes the main analysis results in XML format into that file whenever an analysis is performed. The remarks about the textual report file apply to the XML result file as well. The button opens a view with a text browser on the XML result file.

  The XML result file is much shorter than the local XML report files produced by specific analyses (see section 3.5.2). It only contains the main analysis results, which refer to the entire analyzed code. The local XML report files contain in addition partial results, warnings and error messages, and user settings (but only for a single analysis).

- **`Show MD5 sums:`**
  If this check box is on, a^3 displays the MD5 sums of the executable and the AIS file below the corresponding text fields. The MD5 sums can be selected with the mouse for copying.

- **`Keep temporary files in project directory:`**
  If this check box is off, temporary files are stored in some standard place as described in section 1.5. If the check box is on and there is a current project file (displayed in the title bar), a^3 puts the temporary files in a *project directory* with suffix `.apd`, which it creates as a subdirectory of the directory in which the project file resides. In any case, the current directory for temporary files is displayed in the lower part of the `Files` view.

3.4.2 The `TriCore` View

The `TriCore` view contains target-specific options. It consists of two main parts entitled `Analyzer` and `Machine Settings`. Most options are relevant for timing analysis only. We start with the description of the `Analyzer` part.

3.4.2.1 `TriCore` Analyzer Options

The `Analyzer` part features the following options:

- **`Stack address:`**
  This field is for specifying the value of the stack pointer (register `A10`) at the beginning of the analyzed piece of code. The value can be provided as a decimal or hexadecimal number. Hexadecimal numbers must be prefixed by `0x`.

  The knowledge of the initial stack pointer value improves the precision of the value analysis. For, value analysis can follow the relative changes of the stack pointer when values are pushed or popped. Thus, it can calculate the addresses of memory accesses via the stack. Knowledge of these addresses improves the precision of timing analysis if there are several memory areas with different timing behavior.

  Knowledge of the initial stack pointer value can also improve the precision of stack analysis. It is important for tracking the stack pointer when its value is saved on the stack and restored

later. It is particularly important if the code contains a software test for stack overflow that compares the current stack pointer value with a fixed threshold. Nevertheless, a successful stack analysis is often possible even if no stack pointer value is supplied.

If no stack pointer is written in this field, a^3 tries to guess an appropriate stack pointer value. The guessing is based on symbol and section names. If no stack-related symbols or sections exist, a^3 looks for a memory area that is appropriate for a stack.

- **SDA base:**
  Here you can specify the decimal or hexadecimal value of register `a0`, a global pointer to the Small Data Area SDA.

- **SDA2 base:**
  Here you can specify the decimal or hexadecimal value of register `a1`, a global pointer to the Small `const` Area.

- **Auto:**
  If you activate the `Auto` check boxes to the right of `SDA base` and `SDA2 base`, a^3 extracts the values of SDA base and SDA2 base from the executable. The value of `SDA base` is given by the symbol `_SMALL_DATA_` and the value of `SDA2 base` by the symbol `_SMALL_DATA2_`.

Further `SDA` bases (`SDA3` = register `a8` given by symbol `_SMALL_DATA3_` or `_A8_DATA_` and `SDA4` = register `a9` given by symbol `_SMALL_DATA4_` or `_A9_DATA_`) and other known register values can be specified in the `Registers` view (see section 3.4.4).

- **Number of CSAs:**
  This field is deactivated since the number of Context Save Areas is currently not used by a^3.

- **First free CSA:**
  Here two fields are provided, the left one for specifying the segment of the first free CSA, and the right one for its offset within the segment (in units of CSA size). Segment and offset must be provided as hexadecimal numbers starting with `0x`. The admissible values are `0x0`–`0xd` for the segment, and `0x0`–`0xffff` for the offset. The segment corresponds to bits 19–16 and the offset to bits 15–0 of the `FCX` register.

  Like the stack pointer, the initial value of the `FCX` register is guessed by a^3 if no explicit information is given. Again, the guess is based on symbol names, section names, and properties of memory areas.

  Knowledge of the `FCX` register is of minor importance for stack analysis. It is only important for tracking the stack pointer when its value is saved as part of the upper context and restored later. In many cases, a successful stack analysis is possible even if no `FCX` value is supplied.

- **Disable writes via LMI:**
  This flag tells a^3 that the analyzed part of the executable does not contain any writes via the Local Memory Interface LMI. Therefore, the possibility of such a write is not considered if the target address of a write access is not known. If the flag is on and a^3 finds an access that definitely writes via LMI, it issues an error message that no memory area for that access was found, and that the user should consider to change the setting of the flag.

- **Disable data accesses to PMI:**
  This flag tells a^3 that the analyzed part of the executable does not contain any data accesses to the Program Memory Interface PMI. Therefore, the possibility of such a data access is not considered if the target address of a data access is not known. If the flag is on and a^3 finds a data access that definitely accesses the PMI, it issues an error message that no memory area for that access was found, and that the user should consider to change the setting of the flag.

3.4.2.2 Exporting, Importing, and Loading Machine Settings

The options in the `Machine Settings` part of the `TriCore` view allow for specifying the setting of various hardware properties (see below). As usual, these settings are stored in the project file, but because of the huge amount of possible settings, mechanisms for reading the settings from files are provided.

The current values of the machine settings may be exported to a *machine settings file* (`.msf` file) using the **`Export`** command in the **`Target`** menu or the corresponding tool-bar button. Such an `.msf` file may be imported into another project using the **`Import`** command or button. The imported settings appear in the fields of the `TriCore` view and may be modified as if they had been entered manually.

Alternatively, an `.msf` file may be loaded into the project by selecting **`Load from machine settings file`** in the `TriCore` view. The text field to the right of this radio button is then activated for entering the name of the `.msf` file. Alternatively, you may click the button to the right of the text field to obtain a file-selection dialog box. When `Load from machine settings file` is selected, the lower part of the `TriCore` view is deactivated, i.e. the loaded settings cannot be modified manually (this is the main difference between `Load` and `Import`). Indeed, they are not even written into the fields of the view; the settings are taken directly from the `.msf` file when an analysis is performed.

3.4.2.3 Configuration via Hardware Registers

Apart from importing or loading a machine settings file, a^3 offers two ways to configure the machine settings explicitly: either you can specify the contents of the hardware control registers as a single number per register, or you can define the various parts of the machine configuration in the a^3 GUI. The latter is described in the subsequent sections; in this section we concentrate on the former.

Clicking on the button opens a dialog box for entering the contents of the hardware control registers. When the box is closed by clicking **`OK`**, the machine configuration is derived from the contents of the control registers and put into the machine configuration fields of the a^3 GUI (described in the subsequent sections).

The dialog box for the hardware control registers consists of several tabs.

- The **`Flash`** tab contains the registers `FCON` and `BFCON`.
- The **`EBU`** tab contains `EBU_ADDRSEL`, `EBU_BUSRAP`, `EBU_BUSWAP`, and `EBU_BUSRCON`.

The following sections describe the tabs of the `TriCore` view of the a^3 GUI (not the tabs of the dialog box for hardware registers).

3.4.2.4 The `General` Tab of the `TriCore` View

This tab allows for specifying the exact machine type, the sizes of the caches and of the local memory accessible via LMI.

- **`Machine Type - Target:`**
  Here you should specify the target architecture. There are three choices: **`Infineon TriCore 1766`**, **`Infineon TriCore 1796`**, and **`Infineon TriCore 1797`**.
- **`Instruction Cache - Cache size:`**
  This option indicates the size of the instruction cache. The instruction cache size is fixed to 8 KB for TriCore 1766 and to 16 KB for TriCore 1796. The TriCore 1797 can be configured to have no instruction cache, or to have an instruction cache of size 2, 4, 8, or 16 KB.
- **`Data Cache - Cache size:`**
  This option indicates the size of the data cache (if any). TriCore 1766 and 1796 do not have a data cache. The TriCore 1797 can be configured to have no data cache, or to have a data cache of size 2 or 4 KB.
- **`Instruction Cache - Memory size:`**
  This option indicates the size of the local program memory accessible via LMI. The size is fixed to 24 KB for TriCore 1766 and to 64 KB for TriCore 1796. The TriCore 1797 can be configured to have a local program memory of size 0, 4, 8, ..., 36, or 40 KB.
- **`Data Cache - Memory size:`**
  This option indicates the size of the local data memory accessible via LMI. The size is fixed to 56 KB for TriCore 1766 and to 64 KB for TriCore 1796. The TriCore 1797 can be configured to have a local data memory of size 0, 4, 8, ..., 124, or 128 KB.

3.4.2.5 The `Flash` Tab *(Timing Analysis)*

This tab consists of two sections entitled `BFCON` and `FCON` corresponding to various parts of the control registers of the same names. Most of the `BFCON` section is currently deactivated since only the default settings (visible in the gray text fields) are implemented. The following options are activated:

- **`Frequency of external clock at pin BFCLK0`**
  refers to the burst-mode flash clock output and has four possible values, `1:1`, `1:2`, `1:3`, and `1:4`.
- **`Wait states for read accesses to PFlash`**
  defines the number of wait states that are used for an initial read access to the Program Flash memory area. Possible values are `1–7`.

- **Wait states for error correction of PFlash**
  defines the number of additional wait states for error correction during read access to the Program Flash (`0` or `1`).

- **Wait states for PFlash accesses with wordline hit**
  defines the number of wait states that are used for an initial read access to the same wordline (512 bytes) of Program Flash memory as the last PFlash access. Possible values are `1`–`6`.

- **Wait states for read accesses to DFlash**
  defines the number of wait states that are used for an initial read access to the Data Flash memory area. Possible values are `1`–`7`.

- **Wait states for error correction of DFlash**
  defines the number of additional wait states for error correction during read access to the Data Flash (`0` or `1`).

- **Disable any data fetch from flash**
  disables generally any data read access to internal Flash memory (Program Flash and Data Flash). Possible values are `Off` and `On`.

3.4.2.6 The EBU Tab *(Timing Analysis)*

This tab defines the attributes of the four External Bus Unit banks. It consists of 4 sheets, one for each of `Memory bank 0` through `3`. To enter the attributes for a specific bank, select the name of the bank at the top. The attribute settings for each bank are remembered separately. Therefore, the attribute values displayed in the window change accordingly when you change the selection at the top.

Each bank can be enabled separately using the `Enabled` flag. Once enabled, several attributes can be defined:

- **Start address** and **End address** define the address range of the bank. The `End address` should be the last address within the bank (not the first address after the bank).

  When a bank is enabled for the first time in a project, the address range is set to `0x0`–`0x0`, which is invalid. Therefore, you should always supply a valid range first.

  Valid address ranges are either in the area `0x80800000..0x8fdfffff` or in the area `0xa0800000..0xafdfffff`.

Now some attributes follow that correspond to parts of the Bus Access Parameter Registers `BUSAP0`–`BUSAP3`, which are – under some conditions – multiplied by the `CMULT` field of `BUSCON0`–`BUSCON3`. The values to be specified in the a^3 view are the results of this conditional multiplication.

- **Recovery cycles between different regions**
  specifies the minimum number of cycles between consecutive accesses if the accesses are to different regions. Since the values `0`–`15` may appear in the corresponding field of the

BUSAP registers, and the multiplier CMULT is between 1 and 32, this a^3 field admits values from 0 till 480 (which is $15*32$).

- **Recovery cycles between read and write accesses**
  specifies the minimum number of cycles between a read and write access, and vice versa. Possible values are again 0–480.
- **Recovery cycles after write accesses**
  admits the values 0–224 (which is $7*32$) since the factor from BUSAP is at most 7.
- **Recovery cycles after read accesses** also has possible values 0–224.
- **Data hold cycles for write accesses** has values 0–96 (which is $3*32$).
- **Data cycles during burst accesses** with values 0–224.
- **Programmed wait states for write accesses** with values 1–224.
- **Programmed wait states for read accesses** (1–224).
- **Programmed command delay cycles** (0–224).
- **Address cycles** (1–96).

The remaining attributes correspond to parts of the Bus Configuration Registers BUSCON0–BUSCON3. The multiplier map and the cycle multiplier control CMULT need not be specified because their effects are already incorporated in the values given to the a^3 attributes listed above.

- **Port width:** Possible values are 16bit and 32bit.
- **Address generation control** has three possible values: Demultiplexed, Burst Flash Type 0, and Burst Flash Type 1.
- **Weak prefetch** specifies whether a code prefetch can be aborted by an interrupting data access.
- **Address alignment** means that the EBU aligns the address according to the specified port width (16 bit or 32 bit). If Address alignment is off, the EBU always issues a byte address when performing an external bus access via this chip select.
- **Code prefetch** means that code access always uses the prefetch buffer mechanism. If Code prefetch is off, the prefetch buffer mechanism is never used.

3.4.3 The Analyses View

This section describes the Analyses view in the Configuration section. The options in this view configure the behavior of the analyses triggered by a^3. They are relatively independent from the target architecture.

The Analyses view is structured into tabs. Within each tab, the options are grouped into sections.

3.4.3.1 Reporting (General Tab)

The `Reporting` section in the `General` tab concerns the textual output produced by a^3.

- **`Report file verbosity level`**
  is a slide ruler defining verbosity levels for messages written to the *textual* report files. A tooltip at the ruler indicates which messages are written at the selected verbosity level.

  ⚠ This verbosity selector does not affect the *XML* report files.

- **`Project messages verbosity level`**
  is a slide ruler defining verbosity levels for messages written to the `Messages` view in the `Information` section and the messages subviews in the `Overview` view (see section 3.5.4). A tooltip at the ruler indicates which messages are displayed at the selected verbosity level.

- **`Dump AIS files to report`**
  If this check box is enabled, then the contents of all AIS files used during an analysis are dumped into the report files. This holds for the global textual report file specified in the `Files` view (see section 3.4.1) and for the local textual and XML report files produced by specific analyses (see section 3.5.2).

3.4.3.2 XML Reporting (General Tab)

The `XML Reporting` section in the `General` tab concerns the XML files written by a^3.

- **`Show per context info in XML report`**
  This option influences the contents of the XML report files. If enabled, context-specific information is included, if not, only a summary over all contexts is shown. *This distinction does not matter for stack analysis whose results are not context-specific.*

- **`Output worst-case execution path in XML report`** *(Timing analysis)*
  When this flag is set, a timing analysis will write the worst-case execution path into the XML report.

  ⚠ Since the worst-case execution path is context-specific, this only works if `Show per context info in XML report` is also set.

3.4.3.3 Source Code (General Tab)

The `Source Code` section in the `General` tab concerns issues in relation with source files.

- **`Extract annotations from source files`**
  If this check box is activated, a^3 scans all source files mentioned in the executable and all files included by such files for AIS source-code annotations as described in section 5.31 (provided that this feature is included in your license).

- **`Strip compilation path mentioned in the executable`**
  This flag causes a^3 to ignore the compile directory mentioned in the executable (if any). The exact meaning of the flag is described in section 4.5.

3.4.3.4 All Analyses (General Tab)

The `All Analyses` section in the `General` tab contains options with effects on several different ancillary analyses.

- **`Use only safe patterns`**
  This flag restricts the automatic resolution of computed calls and branches and the automatic loop-bound analysis to code patterns that can be trusted in all situations (see section 4.4.1 for more details).

- **`Assume aligned data accesses`**
  This flag tells a^3 that all (unknown) data accesses are properly aligned. A memory access of width 2^k is *aligned* if the lowest k bits of the accessed memory address are 0. Such aligned accesses need less time than unaligned ones. If the flag is off, a^3 must be pessimistic about accesses to unknown addresses, i.e. it has to assume that they are not aligned to get a correct worst-case execution time. If you are sure that (almost) all data accesses in the analyzed part of the application are aligned, you may switch this flag on to get a more precise execution time bound. More exactly, the flag provides a global default that can be overwritten by AIS annotations referring to specific instructions (see section 5.9.5).

  The exact procedure of a^3 is as follows: if an access is assumed to be aligned (by the global default or a local AIS annotation), the set of possible addresses of the access as computed by the value analysis of a^3 is diminished by removing all non-aligned addresses from it. If the resulting set is non-empty, it is taken as the set of possible addresses. Yet if the resulting set is empty, which means that the access is definitely not aligned according to the knowledge of a^3, a warning is issued and the original set is taken, i.e. the alignment assumption is ignored. Thus, a^3's own knowledge has highest priority, followed by the AIS annotations described in section 5.9.5, followed finally by the flag described here, which has least priority.

  An incorrect setting of the alignment flag may lead to incorrect WCET bounds.

- **`Exceptions never occur`**
  This option has no effect on a^3 for TriCore.

- **`Reduce number of identical messages`**
  This option reduces the number of repeated warnings in the messages views and the textual report files. Repeated warnings are for instance warnings about accesses to illegal memory areas (even if the address of the access varies).

3.4.3.5 Cache Analysis (General Tab)

This part is only relevant for timing analysis.

Usually, a^3 performs a cache analysis that classifies some accesses as cache hits and some as cache misses, while others remain unclassified and thus have an unknown status w.r.t. the cache. Such unclassified accesses cause pipeline analysis to duplicate the current pipeline state: one copy proceeds as if there was a cache hit and the other one proceeds assuming a cache miss.

The `Cache Analysis` section in the `General` tab specifies whether a cache analysis is performed at all, and if so how pipeline analysis interprets its results. There are two options, one for the instruction cache and one for the data cache. They have the same set of possible option values:

- **`Normal`** means that a^3 performs a cache analysis, and pipeline analysis interprets its results in the standard way (hit as hit, miss as miss, and unclassified as unclassified).
- **`Hit if unknown`**
  means that pipeline analysis considers unclassified accesses as cache hits.
- **`Miss if unknown`**
  means that pipeline analysis considers unclassified accesses as cache misses.
- **`Always hit`** means that all cached memory accesses are considered as cache hits without performing a cache analysis.
- **`Always miss`** means that all cached memory accesses are considered as cache misses without performing a cache analysis.
- **`Always unknown`** means that all cached memory accesses are considered as unclassified without performing a cache analysis.

These cache modes apply analogously to those variants of cache analysis that are not able to predict cache misses and thus produce only the two classifications “hit” and “unclassified”.

The purpose of these cache modes is to allow you to get an impression about the influence of the caches on the calculated WCET results.

Except for **`Normal`**, the resulting calculated WCETs should not be considered as correct upper bounds for the real WCETs.

The cache modes also affect the efficiency of a^3. The selections **`Hit if unknown`** and **`Miss if unknown`** avoid the costs of pipeline-state duplication due to unclassified accesses. The choices **`Always hit`** and **`Always miss`** additionally save the costs of cache analysis. The selection **`Always unknown`** also disables cache analysis, but the resulting gain in efficiency is by far outweighed by the need to split the current pipeline state for *every* cached memory access.

`Always unknown` often leads to an explosion in the number of pipeline states, resulting in huge analysis times and memory requirements.

3.4.3.6 Pipeline Analysis (General Tab)

This part is only relevant for timing analysis.

The `Pipeline Analysis` section in the `General` tab contains an option **`WCET computation mode`**, which tells a^3 what to do when the abstract cache and pipeline state is about to split into two or more successor states because of imprecise information. This happens for instance when a memory access cannot be classified as cache hit or cache miss.

The option has three possible values: **`Global worst-case`**, **`Local worst-case`**, and **`Local best-case`**. If `Global worst-case` is selected, a^3 creates all successor states and follows their further evolution. This guarantees that the state sequence leading to the worst-case execution time is found by the analysis. In contrast, `Local worst-case` means that a^3 immediately decides which successor is likely to be the worst (leading to the greatest execution time), and only follows the evolution of this single state. Splits may still occur; they are triggered by situations in which it is not clear which is the locally worst successor state. `Local best-case` is similar, but a^3 immediately decides which successor is likely to be the best (leading to the smallest execution time). These distinctions mainly affect the determination of the run times of the basic blocks. In all cases (including `Local best-case`), the results for routines and the entire task are found by maximizing over all paths. So even with `Local best-case`, a kind of highest execution time is computed, but under the assumption that locally everything works in the best possible way.

Usually, the suppression of splits leads to a massive reduction of the runtime of the analyzer, but there is a risk that the successor state that seems to be worst from a local point of view does not really lead to the worst-case execution time.

If `Local worst-case` is selected, the WCET prediction of a^3 may be *wrong*, i.e. it may be smaller than the actual worst-case execution time.

Therefore, `Local worst-case` should only be used to get quick and rough WCET estimates. For a really reliable WCET prediction, `Global worst-case` must be selected.

3.4.3.7 Path Analysis (General Tab)

This part is only relevant for timing analysis.

The options in this section specify how a overall WCET is derived from the results of cache and pipeline analysis.

- **`Path analysis variant`**
  This selector combines choices along two different dimensions: usage of a global state graph for the entire analyzed code vs. usage of local state graphs for basic blocks, and finding the worst-case path by a graph algorithm vs. by setting up and solving an integer linear problem (ILP).

  State graphs encode the cyclewise evolution of the combined cache and pipeline states. If local state graphs for basic blocks are used, then the information is lost how the end states of one block are connected with the start states of the next block. This leads to a loss of precision, i.e. an overestimation of the WCET, compared with a global state graph.

  The graph algorithm is more efficient than the ILP method, but cannot take into account flow constraints (section 5.22) and only works if the graph is acyclic, which is the case if each loop

iteration and each incarnation of a recursive routine has a separate calling context, i.e. if the parameters `max-length` and `max-unroll` in the context specification (see section 5.4) are infinite (or at least sufficiently big).

There are three path analysis variants:

1. **`ILP based`** means that local state graphs for basic blocks are used, resulting in higher overestimation of the WCET, and an ILP is solved for finding the worst-case path. The advantage of this method is that it also works if different loop iterations share the same context.
2. **`Prediction file based`** means that a global state graph is computed, and a graph algorithm is used for finding the worst-case path. While this is more efficient than the ILP method, it cannot take into account flow constraints as described in section 5.22.
3. **`Prediction file based (ILP)`** means a global state graph is computed and the ILP method is used for finding the worst-case path. This method also works if different loop iterations share the same context.

The three methods can be compared according to their precision and their efficiency (time and space needed for the analysis). Without flow constraints, methods 2 and 3 are equally precise, while method 1 is less precise. Flow constraints may improve the precision, but can only be evaluated by the ILP methods 1 and 3 so that method 3 has greatest precision. If the same context specification method with maximum unrolling is used for all three, method 2 is the most efficient since the graph algorithm is more efficient than the ILP method, followed by method 1, and method 3 is the least efficient since its ILP is larger than the ILP of method 1. The efficiency of methods 1 and 3 can be improved by reducing the number of contexts, which is not possible for method 2.

- **`ILP solver`**
  During timing analysis, a^3 sets up an integer linear problem whose "optimal" solution yields the worst-case execution counts of the basic blocks and the overall WCET (unless path analysis is `Prediction file based`, see above). The solution is computed by invoking an external ILP solver. a^3 can interface with several such solvers, including `CPLEX`. While `CPLEX` is generally faster than the other solvers, it requires a separate license. This license has to be distinguished from the license for the interface between a^3 and `CPLEX`. If the interface license is missing, there is no selection `CPLEX` in this combo box. If an interface license exists, but not a `CPLEX` license, then the selection exists, but `CPLEX` cannot be executed.

3.4.3.8 The Advanced Tab

The `Advanced` tab of the `Analyses` view contains some options of more advanced nature.

- **`Generate pipeline basic block statistics`**
  Setting this check box causes the pipeline analysis of a^3 to write statistics on the number of pipeline states, their copying and deletion etc.

- **`Maximum memory usage`**
  Possible values are `1 %` – `100 %`. The selection `100 %` means that a^3 may claim as much memory as is available on the machine, which is problematic if more than one instance of a^3 is running on a single machine. The other choices restrict the initial memory usage to the given percentage of available memory. Yet note that the memory allocated initially is extended when required by the analysis.

- **`Enable widening for cache states`**
  Cache and pipeline analysis together compute a set of combined cache/time-pipeline-states for every program point and execution context. Each cache/time-pipeline-state is a pair consisting of an abstract cache/time state and a pipeline state. The abstract cache/time state comprises information on the cache contents, the cycle count, and the current jitter. To save space and analysis time, widening can be enabled, which means that compatible cache/time-pipeline-states are joined into one when an instruction retires from the completion queue. Here, two cache/time-pipeline-states are *compatible* if their two pipeline states are equal, i.e. their difference lies in the cache/time states. Compatible states are joined by taking the maximum cycle count, abstracting the jitter, and joining their cache states. Abstract cache states are joined by retaining only the memory blocks that are in all of the joined states, and giving them the maximum of the ages they have in the various states.

  Disabling widening reduces the overestimation caused by the joining of abstract cache/time states, at the expense of vastly increased memory consumption and analysis time.

  The increased memory consumption caused by disabling widening often makes timing analysis practically infeasible. Therefore widening should always be enabled except in special circumstances.

- **`Disable extended feasibility analysis`**
  A program part is infeasible if it can never be executed. Basic infeasibility information is obtained from value analysis when it finds out that a condition always evaluates to true or always to false, and from never-executed annotations as described in section 5.17. Basic infeasibility is propagated forward: if a program point is infeasible, then any point only reachable through it is infeasible, too. *Extended* feasibility analysis in addition propagates infeasibility backward: if a program point is necessarily followed by an infeasible program point, then it is infeasible, too.

  Example: Consider the following code:

```
void R() { R1();  R2();  R3();  R4(); }
```

  and assume that `R1` and `R4` are only called here, but `R3` is also called elsewhere. If `R2` is declared as infeasible, then basic feasibility analysis (active if the flag is on) finds that `R4` is infeasible. In contrast, extended feasibility analysis (active if the flag is off) concludes that `R4`, `R1`, and `R` itself are infeasible. (Routine `R3` is not infeasible since it is also called elsewhere.)

  Since infeasible routines need not be considered in timing and stack analysis, extended feasibility analysis usually leads to faster timing and stack analysis requiring less memory. Disabling extended feasibility analysis is useful if you want to find the origin of the infeasibility.

In the example above, the origin was the assumed declaration that `R2` is infeasible, which stands out more clearly with basic feasibility analysis (only `R2` and `R4` infeasible) than with extended feasibility analysis (all of `R`, `R1`, `R2`, and `R4` infeasible).

- **`Do not analyze values of memory cells`** *(Stack Analysis)*
  Value analysis can analyze the values of registers and memory cells. The latter are not very important for stack analysis, which mainly needs the values of the stack pointer register and of the registers where the stack pointer is saved. Switching on this flag can greatly reduce the time needed for stack analysis, but may lead to less precise analysis results.

3.4.3.9 The Support Tab

The `Support` tab of the `Analyses` view contains some options intended to be used by specialists from the **AbsInt** support team. The field for additional GUI AIS annotations at the bottom should be left empty by ordinary users.

The `Command Lines` section at the top offers text fields to specify additional command line options for some ancillary tools called by a^3. Normally, a^3 calls these tools with appropriate command line options so that no additional options are required.

3.4.4 The `Registers` View

The `Registers` view in the `Configuration` section allows for specifying register values valid at the start point of the analyzed piece of code. This is rarely needed for stack analysis, which is usually successful without knowing any initial register values.

The view contains fields for the various processor registers, which can be filled with the desired register values. The value of the free CSA list head pointer `FCX` can be specified in the `TriCore` view (see section 3.4.2). The value of the stack pointer `a10` should also be specified there.

Register values specified in this view refer to the situation at the start point. Register values at arbitrary positions (including the start point) can be specified in the AIS file (see section 5.10) or in source code annotations (section 5.31).

3.4.5 The `Source files` View

The `Source files` view in the `Configuration` section allows for specifying information that helps a^3 in locating source files. Details about the process of locating source files can be found in section 4.5. The `Source files` provides a panel for naming include paths (section 3.4.5.1), a panel for specifying path replacements (3.4.5.2), and a mechanism to let a^3 compute proposals for include paths and replacements (3.4.5.3).

3.4.5.1 The `Includes` Panel

The `Includes` panel in the `Source files` view offers the possibility to set up a list of include paths, i.e. full path names of directories that are searched for source files.

The panel contains a list of include paths. A click on the button adds a new path with a dummy name to the list. A path name in the list can be selected by clicking on it. The selection can be moved up or down with the corresponding arrow keys. The selected path name can be edited textually, or replaced by a path name from a selection box that appears when the folder icon is clicked. Clicking or moves the selected path up or down, respectively. The order of the path names matters since the include directories are searched in the order in which they are listed. When you click on the button, the path currently selected is removed.

3.4.5.2 The `Replacements` Panel

The `Replacements` panel in the `Source files` view allows for specifying replacements for paths that are part of source file names. Details about the purpose and the exact meaning of the specified replacements can be found in section 4.5. Here, we concentrate on the handling of the graphical user interface.

The panel contains a list of replacement pairs consisting of a `Path` field containing the original path and a `Replacement` field containing its replacement. A click on the button adds a replacement pair with dummy contents to the list. A field of a pair in the list can be selected by clicking on it. The selection can be moved around with the arrow keys. The selected field can be edited textually, or replaced by a path name from a selection box that appears when the folder icon is clicked. Clicking or moves the selected pair up or down, respectively. The order of the pairs matters since the replacements are tried in the order they appear in the list until the first match (see section 4.5). When you click on the button, the currently selected pair is removed.

3.4.5.3 Proposals for Includes and Replacements

Clicking on the button **`Compute Proposals`** near the bottom of the `Source files` view causes a^3 to compute a list of proposals for include paths and replacements. This only works if a^3 already has performed an analysis whereby some source files could not be located.

The proposals are obtained by searching in the directory containing the current project file and all its subdirectories, and searching in all directories already listed in the `Includes` panel and all their subdirectories. Therefore, it can be useful to enter paths in the `Includes` panel before clicking `Compute Proposals`. These paths should be directories that presumably contain one or several source files in some subdirectory.

On the other hand, searching through large directories with many subdirectories may take quite some time. Therefore there is a button **`Stop Computation`** for stopping a running computation of proposals. When a computation has been stopped by clicking on this button, all proposals found so far are listed.

The proposal computation can yield proposals for `Includes` and for `Replacements`. A double click on a proposal or a single click on the green plus symbol to the right of a proposal instructs a^3 to move this proposal to the `Includes` or `Replacements` panel where it takes effect in the next analysis.

3.4.6 The `Visualization` View

The `Visualization` view consists of two halves. The left half offers several options influencing the GDL graph descriptions generated by a^3 to visualize the combined call graph and control-flow graph of the executable and the analysis results. The GDL graph descriptions in turn determine the information in the graphs and their initial visualization. The graph descriptions in chapter 7 and the graph browser description in chapter 8 refer to the situation in which all check boxes except the one for `Manhattan edges` are disabled, and the `Color scheme` is default.

The combined call graphs and control-flow graphs generated by a^3 represent the input program down to the level of individual assembly instructions. The graphs are structured into three hierarchical levels: routines and their calling relationships, basic blocks and the control flow in-between, and individual assembly instructions. Many visualization options control this hierarchical structure and its visualization.

The right half of the view serves as a preview. It shows a picture of a typical output graph illustrating the effect of the option settings. Whenever an option is changed, the preview is automatically redrawn to reflect this change. The preview picture is displayed via the graph browser **aiSee** and thus accepts the **aiSee** commands described in chapter 8 (except for menu and tool-bar commands).

- **`Normal layout`**
  This flag enables you to choose between two different algorithms for computing the layout of the graph images. In "normal layout", basic blocks are displayed in the same order as in the executable. Without "normal layout", the positions of the nodes are derived from their sizes and their connectivities.

  Note that the normal layout property is only guaranteed for basic block graphs in the boxed representation. The clustered and unfolded representations may destroy the normal layout property. (See section 8.2.2 for a description of the various representations of subgraphs.)

- **`Manhattan edges`**
  Usually – meaning when this flag is disabled – call-graph edges are built from line segments of any orientation. When this flag is enabled, only horizontal and vertical line segments are used to build edges. (Basic block edges are always displayed in Manhattan style if the basic block graph is in the boxed representation.) Manhattan layout can also be enabled and disabled during graph viewing (see the **aiSee** manual).

- **`Comma as decimal sign`**
  When it outputs the results of timing analysis, a^3 normally uses a dot as separator between integral and fractional part, e.g., `2.875 us`. If the `Comma` flag is activated, a comma is used instead, e.g., `2,875 us`. This applies to times occurring in the graph visualization as well as times listed in the report file.

  Activation of the `Comma` flag also has two side effects: all times are output in microseconds (`us`), and the fractional part is extended to at least 3 digits by appending 0 digits. For instance, `2.8 us` becomes `2,800 us`, and `0.458 ms` becomes `457,200 us`. (Here, `0.458 ms` has been rounded from `0.4572 ms`.)

- **`Boxed routines`**

When this flag is enabled, graph browsing starts with a picture of the call graph in which all routine nodes are shown in boxed representation so that basic block graphs are visible. This also means that the `Edit` → `Reload` command of the graph browser (cf. section 8.1.6) puts the graph into this state.

- **`Boxed basic blocks`**
  When this flag is enabled, graph browsing starts with a picture in which basic blocks are in boxed representation exposing the instruction sequences contained. This is not visible in the initial image if `Boxed routines` is disabled, since all basic blocks are hidden, however the instruction sequences become visible when basic block graphs are exposed. If `Boxed routines` and `Boxed basic blocks` are enabled, all instructions are visible in the initial picture.

- **`Inline loop routines`**
  If this flag is activated, loops in loop-transformed graphs are not displayed as separate routines, but as part of the routines from which they have been extracted (see section 4.1 for the loop transformation and section 7.1 for pictures).

- **`Trace-like pipeline state visualization`**
  If this flag is disabled, the visualization of the cache and pipeline states as described in section 7.5 is structured according to the instruction about to leave the pipeline. For each such instruction, the visualization shows the evolution of the cache and pipeline states while it is at the end of the pipeline. The final states for one instruction are not visually connected to the initial states of the next instruction. If the flag is enabled, the visualization is structured according to the current basic block. Thus, the final states for one instruction are connected to the initial states of the next, and there are no borders separating the instructions.

- **`Font`**
  Here you can select the font for the labels of the nodes in the graphs generated by a^3. The selected font is shown in the preview, but note that the scaling within the preview may vary so that it may seem that font size changes are not handled correctly. To set the scaling in the preview to 100%, click into the preview window and type `0` (zero).

- **`Display WCET profiles`**
  If this flag is activated, graphs with timing analysis results contain an additional WCET profile in every routine box.

- **`Display source code`**
  This flag influences the labels in the nodes representing basic blocks. When the flag is disabled, basic block nodes are labeled by their hexadecimal start address, e.g., `0x10a`. If `Display source code` is enabled, basic block nodes are labeled by C source code (provided that this feature is included in your license). See section 7.1.5 for a more detailed description of this feature.

- **`Color scheme`**
  There are three possible values: **`Default`**, **`Black & white`**, and **`User defined`**. If `Default` is selected, a^3 uses the standard set of colors described in section 7.1. With `Black & white`, only black and white elements are used in graph visualization. With

`User defined`, the standard set of colors is replaced by the set defined in the dialog box that is activated by clicking on the button. This dialog box enables you to define your own colors and to associate them with the various features of graphs generated by a^3. Color sets defined in this way are not active unless `User defined` is selected.

- **`Disable visualization for batch mode`** (at the bottom of the view)
  This option only has effect on batch-mode runs of a^3, i.e. runs started with the option `-b` (see section 2.1). Runs in interactive batch mode (option `-B`) and interactive a^3 sessions are not affected.

  When visualization is disabled for batch mode, then batch runs of a^3 do not create the textual descriptions of the annotated call graphs and control-flow graphs, so that the report files are the only output of the analyses. Details are more complex, however.

  For timing analysis only: If `Disable visualization` is set, batch runs of a^3 do not write the table of routine WCET contributions into the report file. The report file thus carries less information than in presence of visualization. This remark only applies to batch mode runs started with `-b`.

- **`Set as default`**
  The Visualization options are stored in the project file. When a new project is started by opening a^3 without a project file or by the **`New`** operation from the `Project` menu, the Visualization options are initialized to some default values. These defaults are set to the current values when the `Set as default` button is clicked. This allows for propagating the current Visualization options to a new project, but does not affect any existing projects.

3.5 The `Analyses` Views

The **`Analyses`** section in the list of available views allows you to define and manage analyses. It contains at least the `Create` view (section 3.5.1) and the `Overview` view (section 3.5.4). The `Create` view offers commands for creating new analyses, and the `Overview` view lists the existing analyses, offers commands for managing them, and summarizes their results. These two views cannot be closed.

Each new analysis obtains its own view, in which the parameters of the analysis can be set. Analyses that cannot be executed because of an incomplete parameter set or invalid settings are marked in the list of available views by an exclamation point, which carries a tool tip indicating the reason why the analysis cannot be executed.

3.5.1 The `Create` View

The `Create` view allows you to create new analyses. It lists the kinds of analyses that are available for TriCore target processors and enabled in your license. To create a new analysis, you can either click on an analysis in the list or issue the corresponding command from the `New Analysis` menu or tool bar. It is possible to create more than one analysis of the same kind. The present manual describes the following analyses:

- **Control-Flow Graph**
  creates a visualization of the combined call graph and control-flow graph of the application without specific analysis results (see section 7.1).

- **ValueAnalyzer**
  creates a value analysis allowing to explore the effects of individual instructions on registers and memory cells (see section 7.2).

- **StackAnalyzer**
  creates a stack analysis computing the maximum stack usage of a task (see section 7.3).

- **aiT**
  creates a timing analysis (see 7.4) computing a safe upper bound of the worst-case execution time (WCET) of a task.

- **Difference Analysis** creates a difference analysis that when started runs two specified timing analyses *with the same entry points* and produces a combined result graph highlighting the differences between the results of the two analyses.

3.5.2 The Analysis Views

Whenever a new analysis is created, it is given a view for setting its parameters. Closing this view via its button removes the analysis from the project.

The views for the various kinds of analyses are quite similar and thus described all at once. They contain the following fields:

- **ID**
  gives an ID (identifier) to the analysis. This identifier appears in the list of analyses in the `Overview` view (section 3.5.4), and also will be the name of the analysis view. When a new analysis is created, it is given a dummy identifier that can be modified by editing the `ID` text field.

- **Comment**
  provides space for an optional comment text.

- **Analysis start**
  defines the start point of the analysis. You may type a routine name, a label name from the AIS file (see section 5.25), or a code address into this field, or select a routine name (but not a label name) from the selection box that pops up when the button at the right side of the text field is clicked. The start-point selection is described in section 3.5.5.

  A code address need not be a routine entry. If the symbol table does not contain a name for start address *A*, the decoder creates an anonymous routine `:Anon_`*A*.

- **AIS file**
  specifies an AIS file with annotations specific for this analysis. This AIS file is read after the global AIS file named in the `AIS file` field of the `Files` view (see section 3.4.1).

Clicking on the edit button to the right of the text field opens a new view with a text editor on the file whose name is displayed there. If the field is empty when the button is clicked, a file name is generated automatically.

- **Report file**
  is a panel for entering the name of a local textual report file. This panel does not exist for Control-Flow Graph "analyses". If the panel specifies a file, then a^3 writes a report with messages and analysis results into this file whenever the analysis is performed. The report file is also written during the batch mode analyses performed when a^3 is started with the command line option `-b` or `-B` (see chapter 2.1).

  The contents of the report file are deleted before a new report is written.

  The button opens a view with a text browser on the report file.

- **XML report file**
  is a panel for entering the name of a local XML report file. This panel does not exist for Control-Flow Graph "analyses". If the panel specifies a file, then a^3 writes a report in XML format into that file whenever the analysis is performed. The remarks about the textual report file apply to the XML report file as well.

  This text field is only visible if XML reporting is included in your license.

  The button opens a view with an XML browser on the XML report file.

- **GDL output**
  names a file in which the result of the analysis is written in GDL format, i.e. as a combined call graph and control-flow graph annotated with analysis results. If the button to the right of the text field is clicked while the text field is empty, a file name is generated automatically. If the text field is not empty and the analysis has been run successfully, the button opens a view with a graph browser on the specified GDL file. Note that a view with the generated graph is opened anyway automatically in case of a control-flow graph "analysis" or an interactive analysis after it has run successfully. In other cases, it can be opened via the `Overview` view (see section 3.5.4) or the button (see section 3.5.3) even if no name for the GDL output is specified.

3.5.3 Commands for Analyses

The `Project` menu and tool bar contains commands for running analyses, and each analysis view is associated with an `Analysis` menu and tool bar for issuing commands special for this analysis. We start with the description of the latter ones.

- **Analysis → Start** or **F7** or in the Analysis tool bar
  starts the analysis in the current view. The analysis can be prematurely stopped with **Project → Stop analysis** or **Ctrl+Esc** or in the project tool bar. If the analysis was successful, its result is listed in the `Overview` view (see section 3.5.4), from which a view with the graph containing the analysis results can be opened. Control-flow graph "analyses" form an exception; they immediately open a view with the resulting graph.

The features of the generated graphs are described in sections 7.1 (control-flow graphs), 7.2 (value analysis), 7.3 (stack analysis), 7.4 (timing analysis).

- **Analysis** → **Interactive** or in the Analysis tool bar *(Timing and value analysis)* starts the analysis in the current view. The analysis can be prematurely stopped with **Project** → **Stop analysis** or **Ctrl+Esc** or in the project tool bar. If the analysis was successful, a view with the graph containing the analysis results is opened. The difference to the `Start` operation lies in the fact that the generated graph provides access to the cache and pipeline states of the analyzer (see section 7.5) in case of timing analysis, and to the results of value analysis (see section 7.2) in case of timing and value analysis.

- **Analysis** → **Control-Flow Graph** or in the Analysis tool bar
  opens a view with the call graph and control-flow graph of the analyzed code *without running the analysis*. The elements of the generated graphs are described in section 7.1. The purpose of this operation is to get a quick overview of the structure of the code to be analyzed without the need to wait for the termination of the actual analysis.

- **Analysis** → **Disassembly** or in the Analysis tool bar
  opens a view with a disassembly listing of the analyzed code, i.e. the part of the executable reachable from the start point of the analysis (see section 3.5.6).

- **Analysis** → **Display analysis results** or in the Analysis tool bar
  opens a view with the graph containing the analysis results, or switches to that view if it is already open.

- **Analysis** → **Display analysis messages** or in the Analysis tool bar
  switches to the `Overview` view and shows there the messages generated while running the analysis (see section 3.5.4).

The `Project` menu and tool bar contains the following commands relevant for performing analyses:

- **Start all analyses** or **F6** or in the project tool bar
  starts all active analyses in the current project (those with a checked check box in the list of defined analyses, see section 3.5.4). The same happens without a user command when a^3 is started in batch mode (with option `-b`).

- **Stop analysis** or **Ctrl+Esc** or in the project tool bar
  stops the currently running analyses. This applies to the bunch of analyses started by the `Start all` command or button described above, and to single analyses started from analysis views with the `Start` command or button, or with the `Interactive` command or button.

3.5.4 The `Overview` View

The `Overview` view consists of two parts. The upper part shows the list of currently defined analyses. The lower part shows the messages that have been generated while running the analysis selected in the upper part.

The upper part is organized as a table with three columns. The first column contains the `ID` (identifier) of the analysis as defined in the corresponding analysis view (see section 3.5.2), the second column its `Type`, and the third its `Result`. The result is only shown for analyses that have successfully been executed since the start of the a^3 session. The following types and results exist:

- **`Value-Analysis`** with result **`Passed`** or **`Failed`**.
- **`Stack-Analysis`** with a result

  `User=`*U* `bytes, System=`*S* `bytes`

  giving an upper bound for the stack usage.
- **`aiT`** with a result giving the WCET in processor cycles and in real time units if a clock rate has been specified as described in section 5.2.

Lines corresponding to analyses whose run has generated warnings are displayed with an orange background, and those with errors with a red background. The warnings and errors can be inspected in the messages displayed in the lower part of the view when the line is selected.

The currently selected analysis can be deleted via the button. (Another way to delete an analysis is closing its view.) The analyses in the list can be reordered by means of the and buttons. A double click on an analysis with a result lets a^3 open a graph browser showing the detailed analysis results attached to the combined call graph and control-flow graph as described in chapter 7.

Each analysis in the list comes with a check box for activating or deactivating it. When `Start all analyses` is invoked from the `Project` menu or by clicking , or when a^3 is started in batch mode (with `-b`), all active analyses are performed (the inactive ones are skipped). There are buttons to activate or deactivate all analyses at once.

The `Overview` view is associated with an `Analyses` menu and tool bar. When no analysis is selected, this menu contains a single command `Clear analysis results` and the tool bar a corresponding button . This command clears all analysis results including the messages in the lower part. When an analysis is selected, the `Analyses` menu and tool bar is extended by the commands and buttons belonging to the kind of the selected analysis, e.g., `Start` . These commands and buttons are the same as those induced by the corresponding analysis view. They are described in section 3.5.3.

Even if an analysis is selected, the command `Clear analysis results` and the corresponding button delete *all* analysis results, not only the result of the selected analysis.

3.5.5 Start-Point Selection

As described in section 3.5.2, each analysis view features an `Analysis start` text field containing the start point of the analysis, which may be entered by text editing or via a selection box opened by clicking on the button.

The selection box mainly consists of a table listing all routine names and labels in the executable with their start address and end address (the latter may be unknown, in particular for labels). It

does not contain the labels defined in the AIS file (section 5.25) although such labels work when typed into the Analysis start text field.

The listed information is extracted from the symbol table and debug information in the executable. If the list is initially empty, click on the button to initiate the extraction of the necessary information from the executable. The button should also be used after changing the executable.

After selecting a list element by a mouse click, clicking on the OK button writes the selected name into the Analysis start text field.

To simplify the search for a particular list entry, the selection box features a text field at the top whose contents are used for filtering the list. If the check box Regular expression is off, the contents of the text field are handled as plain text, and the list is reduced to those lines matched by the text, i.e. containing the text as a substring (not necessarily at the beginning of the Name field, but somewhere in the interior of the Name field, the Start field, or the End field so that you can search for fragments of names and fragments of addresses). For instance, main matches routine main, but also other routines with names like main_trim, __init_main, or domain_check, and 0x matches all lines because of the 0x in the start address. The filtering of the list proceeds as you type; each character typed may lead to some shortening of the list. The button clears the text field and consequently restores the original list of all names.

If the check box Regular expression is on, the contents of the text field are considered as a regular expression, and the list is reduced to those lines matched by the expression. The simplest regular expressions are strings consisting of letters, digits, underscores, and spaces. Such a string matches all text containing it as a substring (like in the plain-text case described above). Thus most characters in a regular expression stand for themselves, yet there are some special characters implementing the following features:

.	matches any character.
^	matches the beginning of the Name, Start, or End field.
$	matches the end of the Name, Start, or End field.
a*	matches a sequence of zero or more a's.
a+	matches a sequence of one or more a's.
a?	matches an optional a.
\s	matches a white space character.
\d	matches a digit.
[abs]	matches a single character that is a, b, or s.
[^abs]	matches a single character that is *not* a, b, or s.
[a-s]	matches a single character that is in the range from a to s.
[^a-s]	matches a single character that is *not* in the range from a to s.

For instance, v[ei] matches all lines containing ve or vi, in particular even and divides. To match only the line with routine main and nothing else, write ^main$. The symbols ^ and $ are needed to prevent the regular expression from matching routines like main_protocol or run_main.

Regular expressions may be combined by '|', meaning *or*, and grouped by parentheses '(' and ')'. So if you want to retain only routines prime and swap in the filtered list, you should use the

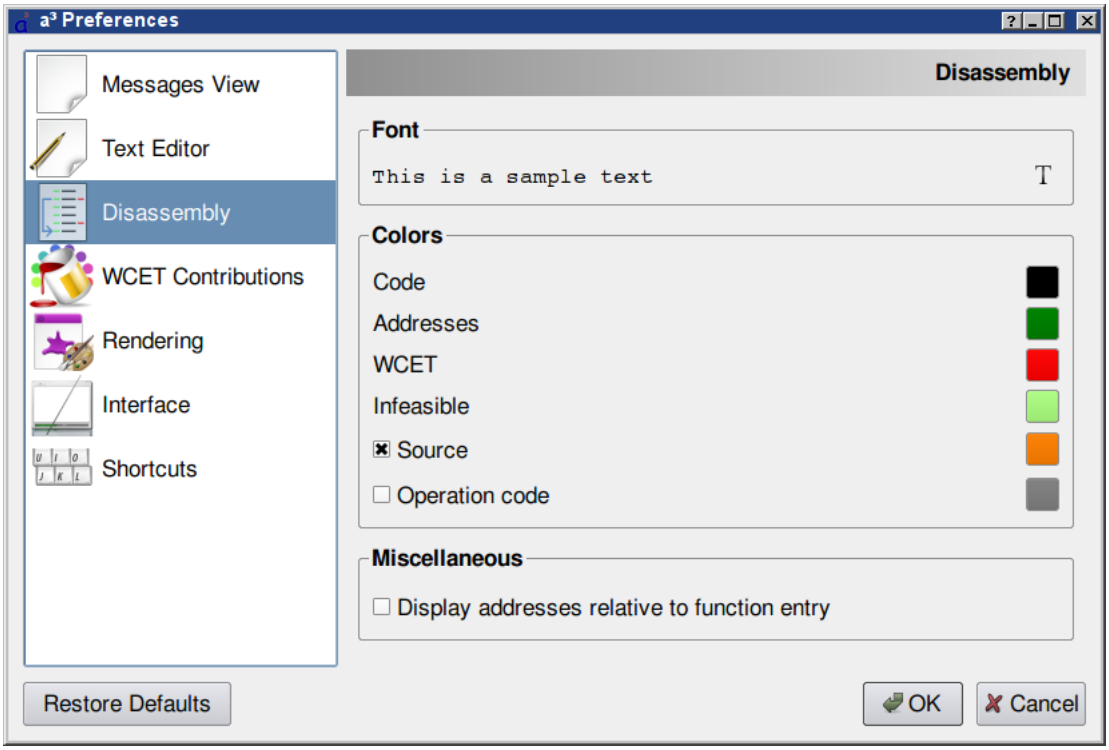


Figure 3.3: Preferences dialog box for the disassembly view

regular expression `^(prime|swap)$`.

3.5.6 The Disassembly View

Upon request, a^3 can present the disassembly of the analyzed part of the executable. This is initiated by calling **`Analysis`** → **`Disassembly`** or clicking on in the Analysis tool bar while an analysis view is active (see section 3.5.3), or an analysis is selected in the `Overview` view (section 3.5.4). The information then refers to the part of the executable reachable from the start point of the analysis.

The disassembly view is marked by the icon and named like the start point of the analyzed part of the executable, e.g., `main`.

What exactly is shown in the disassembly view depends on the settings in the `Disassembly` part of the `Preferences` dialog box (see Figure 3.3) that can be invoked by **`Edit`** → **`Preferences`**, , or from the context menu of the disassembly view.

The disassembly view always shows code (assembly instructions) and code addresses. The displaying of source code and operation codes (as hex numbers) can be enabled or suppressed by the respective check boxes in the `Preferences` dialog box. *Source* code can only be displayed if this feature is enabled in your license. It is taken from the source files whose names are extracted from the debug information in the executable. Section 4.5 describes how these files are found in the file system. a^3 uses the line information in the executable to merge the source lines into the assembly code.

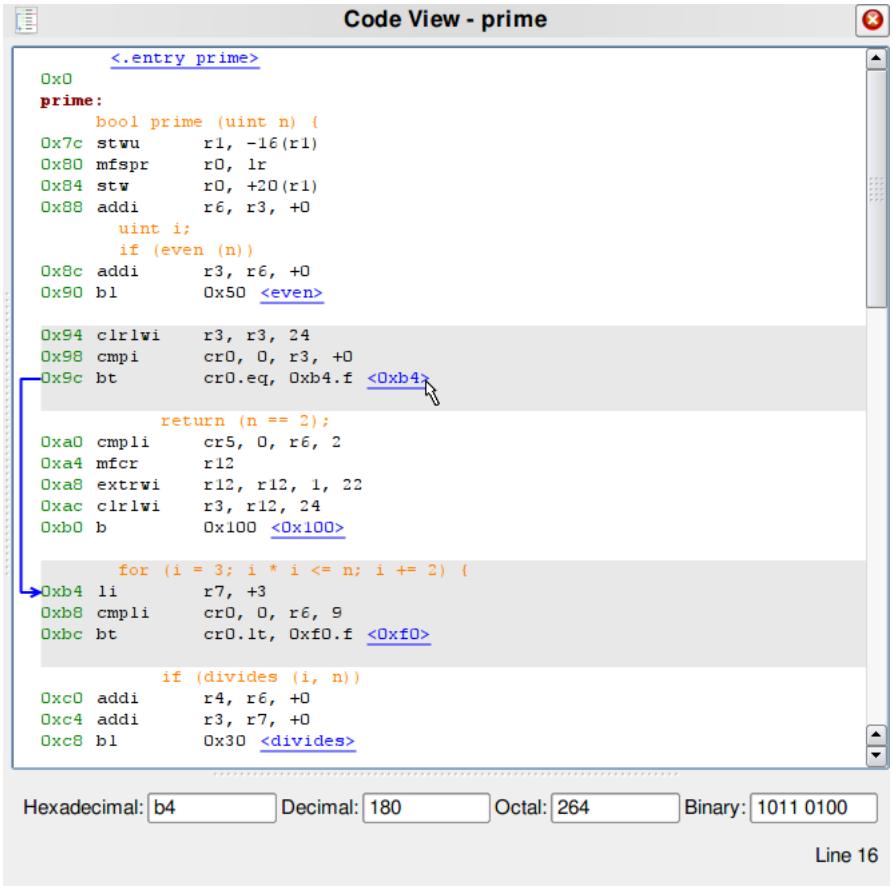


Figure 3.4: A disassembly view

The various features such as code, addresses, etc. are displayed using the colors shown on the buttons in the `Preferences` dialog box. Clicking on these color buttons activates color selection boxes where you can choose your own colors for code, addresses, etc. The term **`Addresses`** refers to the addresses of instructions (not the target addresses occurring as part of the instructions). These addresses can be shown absolutely or relative to function entries. If disassembly is produced from a WCET analysis after running the analysis, then the color **`WCET`** is used for addresses on the WCET path, and **`Infeasible`** for addresses on paths that are never executed (according to the knowledge of a^3). To activate this feature for disassembly produced before running the WCET analysis, it is sufficient to click on the reload button of the disassembly view after running the analysis. Click the **`Restore Defaults`** button to reset all colors to the system defaults.

Figure 3.4 shows a disassembly view filled with some disassembled code, using the preferences shown in Figure 3.3, i.e. interspersed with source code. *This example is generic, i.e. not specific to the TriCore architecture.*

The code in the disassembly view is not influenced by the loop transformation (see section 4.1), i.e. loops are shown as part of the routines where they occur, not as separate loop routines.

Basic blocks are separated by blank lines and indicated by varying the background color. The targets of calls and branches are active links. When the mouse points to such a target, a blue arrow

appears in the left margin pointing to the target instruction (see Figure 3.4). When you click on the target, the `Disassembly` view scrolls so that the target instruction becomes visible. The arrow remains visible after the click until the mouse pointer is moved. You can return to the original position by selecting `Return to last caller` from the context menu of the `Disassembly` window or by pressing the `Backspace` key.

The bottom of the disassembly view features four text fields entitled `Hexadecimal`, `Decimal`, `Octal`, and `Binary`. When the mouse points to a number in the disassembly view, these fields are filled with the various representations of that number. In the example of Figure 3.4, this happened with the target address `0xb4`. (This does not work for all numbers appearing in the view.)

The context menu also offers `Find` and `Go to` operations. The `Find` operation can be used for text search, which applies uniformly to all text displayed in the window, be it source code, assembly code, or code addresses. The `Go to` operation implements search for the start points of routines that can be selected from a menu and for code addresses that can be typed in.

The disassembly view shows only the part of the executable that is reachable from the current start point. In particular, the list of functions among which you may choose in the `Go to` operation usually does not contain all functions in the executable.

3.6 The `Information` Views

The views in the section **`Information`** present some information extracted from the executable without running an analysis. These views are created once an executable has been specified.

3.6.1 The `Messages` View

The `Messages` view in the `Information` section displays only the messages generated while creating the other `Information` views. The messages resulting from running an analysis are presented in the `Overview` view (see section 3.5.4).

3.6.2 The `Symbols` View

The `Symbols` view displays the symbol table of the executable. It contains the following columns for each symbol found:

- **`Routine`** indicates whether the symbol is a routine name. There are three possibilities: **`yes`**, also marked by a green flag, **`perhaps`** with a yellow flag, and **`no`** with a red flag.
- **`Symbol`** is the name of the symbol.
- **`Start address`** and **`End address`**, where the latter is not always known.
- **`Section`** (if known).

Like the start-point selection, the Symbols view contains a search function with analogous properties (see section 3.5.5 for a description of this search function). The search function is applied to the column Symbol with the symbol name and the columns Start address and End address, but not to the column Section with the section name.

3.6.3 The Source files View

Here we talk about the Source files view in the Information section. It should not be confused with the Source files view in the Configuration section (described in section 3.4.5).

The Source files view shows the list of source files, as far as it can be determined from the executable. (Section 4.5 describes how source files are located by a^3.) Source files that could be located successfully are displayed in black, the other ones in gray. A double click on a successfully located source file opens a view with a text browser showing the contents of the source file.

3.6.4 Variables and Functions Views

The views entitled Variables and Functions display the Dwarf debug information contained in the executable, which of course only works if the executable contains such information.

Variables and Functions views do not appear if the executable does not contain debug information.

The Variables view displays information on the global and local variables of the program. The Functions view lists the functions of the program with their parameters. For variables and parameters, the following information is displayed:

- The **Name** of the variable or parameter.
- Its **Type**, e.g., int, unsigned int, char[128], struct, or struct*. Structs or pointers to structs can be opened to get access to information on the struct components.
- Type **Modifiers** such as const or volatile.
- The **Size** in bytes, e.g., 4 for int. The size of an array is displayed as m/n where m is the total size of the array and n is the size of one component, e.g., the size of int[5] is shown as 20/4. Note that a struct may be larger than the sum of its component sizes because of alignment.
- The **Absolute address**, which can be a hexadecimal number, a register such as reg(7), or a register with offset such as reg(1)+8. The registers are usually not named as in the target architecture, but numbered sequentially as in the Dwarf debug information. The correspondence between the Dwarf register numbers and the actual register names is part of the processor API.

- The **Relative address**, which only appears for components of structs and means the address of the component relative to the address of the struct.
- Some optional **Remarks**.

3.6.5 The Sections View

The Sections view displays the section table of the executable. It contains the following columns for each section found:

- **Address** is the start address of the section.
- **Size** is the size of the section.
- **Name** is the name of the section.
- **Type** is code or data.
- **Byte order** is lbf (low byte first) or hbf (high byte first).
- **Relocate**, **Allocated**, **Readable**, **Writable**, and **Executable** are possible properties of a section. They can be either satisfied (yes and green flag) or not (no and red flag).

Like the start-point selection, the Sections view contains a search function with analogous properties (see section 3.5.5 for a description of this search function). The search function is applied to the columns Address, Size, Name, Type, and Byte order.

Chapter 4

Background Information

This section presents some background information about internal representation of loops (4.1), origin and naming of routines (4.2), calling contexts of routines (4.3), analysis phases (4.4), and locating source files (4.5).

4.1 Loop Transformation

a^3 works on an internal representation of the input program as a combined call graph and control-flow graph. When reading an executable, it first builds a graph whose structure closely reflects the control structure of the executable. This original graph is then modified by the so-called *loop transformation*, which turns loops into separate routines that call themselves recursively. The purpose of this loop transformation is to enhance the precision of the analyses in presence of loops. All graphs shown by a^3 are loop-transformed, and the output of a^3 refers to the loop-transformed graphs. Thus it is important to explain the effect of loop transformation before describing the parameter files of a^3 and the visualization of graphs.

Note that loop transformation only concerns the graphs used as internal representation; the executable itself is not affected. Consequently, the loop transformation is not visible in the `Disassembly` view.

4.1.1 Simple Loops

Figure 4.1 shows the effect of loop transformation on a simple loop. The picture on the left shows the control-flow graph of routine `F` that contains such a simple loop. In a kind of pseudo source code, this routine could be specified as

```
routine F {
  b0;
  while (b1)  b2;
  b3;
}
```

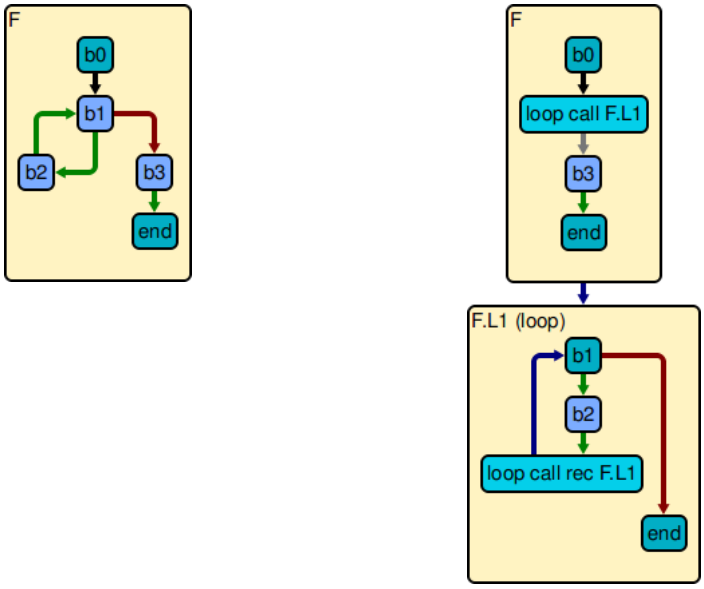


Figure 4.1: Loop transformation: simple loop

The black arrows represent normal control flow, green arrows lead from a condition to its "true" successor, and red arrows to its "false" successor. The routine is entered at `b0` and left at the special *exit node* `end`. A possible trace of routine `F` is `b0`→`b1`→`b2`→`b1`→`b3`. This corresponds to a possible path through the graph of Figure 4.1, left.

The picture on the right hand side of Figure 4.1 shows the result of loop transformation. The loop has been turned into a separate *loop routine* named `F.L1` that is called from `F` at `loop call F.L1`; the calling relationship is indicated by the dark blue arrow from routine `F` to routine `F.L1`.

In pseudo source code, the transformed program could be specified as

```
routine F {
  b0;
  loop_call_F.L1:  call F.L1;
  b3;
}
routine F.L1 {
  if (b1) {
    b2;
    loop_call_rec_F.L1:  call F.L1;
  }
}
```

Thus, iteration has been replaced by recursion: the loop routine calls itself at `loop call rec F.L1`, indicated by the dark blue arrow from `loop call rec F.L1` to `b1`. The loop routine is left if `b1` is false, which is depicted by the red arrow from `b1` to `end`.

A trace such as `b0`→`b1`→`b2`→`b1`→`b3` still corresponds to a path through the graph, but this path is not that obvious. In constructing this path, two rules must be observed:

1. Whenever a loop call block is reached (`loop call F.L1` or `loop call rec F.L1` in this example), control continues at the entry block of the corresponding loop routine (`b1` in this example; entry blocks are green in contrast to non-entry blocks, which are blue).

2. Whenever control reaches the exit node `end` of the loop, it is passed back to the loop call node in the routine the loop was extracted from (`loop call F.L1` in this example).

Example: The trace `b0→b1→b2→b1→b3` corresponds to the path

```
b0 → loop call F.L1 →
     b1 → b2 → loop call rec F.L1 →
               b1 → end of F.L1 →
loop call F.L1 →b3.
```

4.1.2 Numbering of Loop Routines

If routine `F` contains several loops, the extracted loop routines are counted from 1: `F.L1`, `F.L2`, etc. This includes nested loops: Two nested loops are extracted as `F.L1` and `F.L2`; there are no loop routines such as `F.L1.L1`.

Loop routines are numbered according to the address of their entry point. If two nested loops are entered by fall-through from the code before the loop, the entry of the outer loop is before the entry of the inner loop, and so the number of the outer loop is smaller than the number of the inner loop. Yet if two nested loops are entered by jumps to their last basic blocks, then the entry of the outer loop lies behind the entry of the inner loop, and so the inner loop has the smaller number.

4.1.3 Loops with Several Exits

Loops with additional exits in the manner of C's `break` statement lead to similar results; the `break` statements cause additional arrows pointing to the loop exit node. Yet loops with exits in the manner of C's `return` statement are more difficult because they usually cause the loop routine to have several exit points with different return targets. Consider for instance the left picture in Figure 4.2. The loop consisting of `b1` and `b2` has two exits: `b1` → `b4` and `b2` → `b3`. In pseudo source code, routine `F` could be specified as

```
routine F {
  b0;
  while (b1) {
    if (b2)  return b3;
  }
  return b4;
}
```

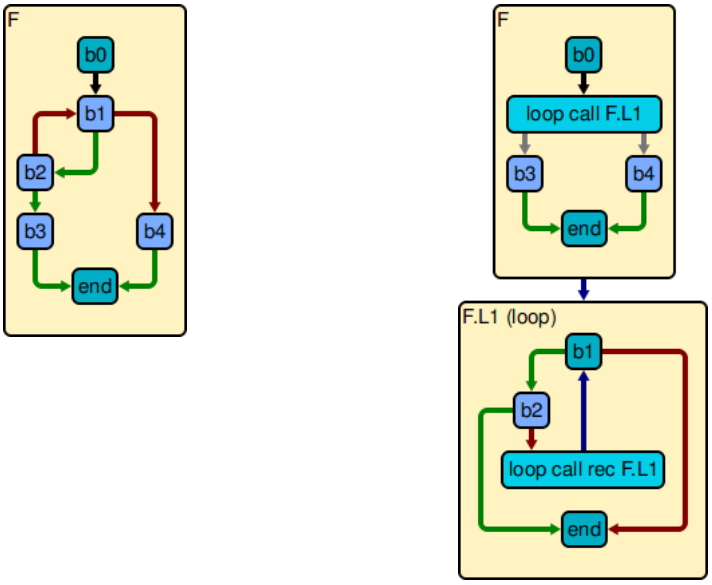


Figure 4.2: Loop transformation: loop with two exits

Although expression `b3` syntactically occurs within the braces of the loop, its evaluation does not belong to the loop. It is evaluated when the loop is being left as shown in Figure 4.2, left. The loop consists of `b1` and `b2` only. Expression `b4` more obviously does not belong to the loop. There are two ways to leave the loop: If `b1` is false, the loop is left normally, and the code after the loop is executed, i.e. `b4` is evaluated. If `b2` is true, the loop is left via `return b3`, i.e. `b3` is evaluated. Thus the two loop exits lead to two different successors: `b4` and `b3`.

The picture on the right hand side of Figure 4.2 shows the result of loop transformation. Again, the loop has been turned into a separate routine named `F.L1` that is called from `F` at `loop call F.L1`, and the loop routine calls itself at `loop call rec F.L1`, indicated by the dark blue arrow from `loop call rec F.L1` to `b1`. The two loop exits are represented by the paths

`b1` → `end` → `loop call F.L1` → `b4` and

`b2` → `end` → `loop call F.L1` → `b3`.

Since both loop exits are channeled through the same loop exit node, the loop-transformed graph also admits the mixed paths

`b1` → `end` → `loop call F.L1` → `b3` and

`b2` → `end` → `loop call F.L1` → `b4`,

which do not correspond to any valid control flow; the original graph has no direct link from `b1` to `b3`, nor from `b2` to `b4`.

The additional paths are introduced by the loop transformation. They are taken into account by the last step of WCET analysis when the worst-case path is searched. Hence, it may happen that the calculated worst-case path includes one of these invalid paths. This still results in a correct approximation of the real WCET, i.e. the calculated WCET is larger than or equal to the real WCET. It only may make the calculated WCET less precise, i.e. higher above the real WCET.

In stack analysis, the additional paths do not necessarily cause imprecision. Even if they do, they cannot induce wrong analysis results.

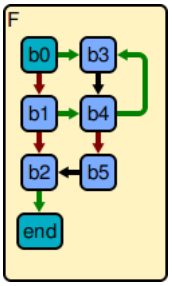


Figure 4.3: Loop with two entries

4.1.4 Loops with Several Entries

Executables may also contain loops with several different entry points (*multi-entry loops* or *irreducible loops*). Such loops are unlikely to occur in object code resulting from compiling C source code, yet may be found in library routines.

Figure 4.3 shows an example for a loop with two entries. The loop consists of the blocks `b3` and `b4`, and the two entries are `b0` → `b3` and `b1` → `b4`.

⚠ Loop transformation is not able to transform loops with several entry points into loop routines because routines have to have a single entry point. Thus, loops with several entry points are hard to find; you need to look closely at the control-flow graph or watch out for info messages of a^3 about irreducible loops.

Loops with several entry points are not endowed with special loop contexts, are not considered by automatic loop bound analysis as described in section 4.4.1, and cannot be bounded by user annotations as described in section 5.20.

4.2 Routines

The code in an executable is partitioned into *routines*, i.e., assembly procedures, which have several different origins:

- A C source function is translated into a routine unless the compiler decides to inline its code. The assembly name of such a routine is usually the name of the original C function. For instance, the `main` function of the C source is translated into routine `main`.

  ⚠ Of course, the routine names in the executable are the compiler's choice. The remarks above describe a convention that seems to be obeyed by the compiler, but there is no guarantee that the compiler always does so.

- In general, the executable also contains routines resulting from the translation of functions in the C library. Usually, the names of these routines are derived from the corresponding source functions using the conventions outlined above. Often, the names of internal library functions start with an underscore, so that the corresponding routines can be recognized by a leading underscore.

- An executable may also contain hand-written assembly routines implementing special functions, with usually quite special names.
- In addition, there are virtual loop routines that are created by loop transformation (see section 4.1). The loops in routine *R* lead to loop routines named *R*`.L1`, *R*`.L2`, etc.
- There may also be *anonymous routines* `:Anon_`*A* where *A* is the start address of the routine. A routine with start address *A* is anonymous if the symbol table does not contain a routine name associated with *A*. Thus all routines are anonymous if the executable is stripped.

  Even if the executable is not stripped, there may be anonymous routines. There are two reasons: First, a^3 creates a routine with start address *A* when it starts the analysis at *A*, or when it finds a call to address *A* or an `ENTRY` *A* specification in the AIS file (see section 5.6). Second, a^3 resolves code sharing between routines by turning a shared code snippet into a routine of its own. Consider the following example: Routine R_1 extends from address `0x1000` to `0x2FFF`. Another routine R_2 starts at `0x4000` and contains a branch to `0x2000`, which is in the code area of R_1. Such a branch may be the result of some compiler optimization. a^3 then resolves the question whether `0x2000` belongs to R_1 or R_2 by creating a new routine `:Anon_2000` with start address `0x2000`. The branch from R_2 to `0x2000` is then considered as a call of `:Anon_2000`. Furthermore, R_1 now ends just in front of `0x2000`. If control can flow from the instruction before `0x2000` to `0x2000`, a^3 introduces a *dummy call* before `0x2000` that calls `:Anon_2000`.

In the call graphs of a^3, routines are visualized as yellow or orange boxes labeled by the routine name. Loop routines are further marked by the string `(loop)` put after the routine name. In general, a yellow routine contains source code, while an orange routine does not; see section 7.1.1 for details.

4.3 Calling Contexts

In an over-simplified view, a static program analysis computes some abstract information for every program point *p*. The abstract information for *p* has to be a correct approximation of the concrete program state at *p* whenever control reaches *p* (no matter what happened before). Thus, the abstract information for a program point *p* in a routine *R* must approximate all program states at *p* in all calls of *R*.

To be more concrete, consider a *value analysis* that computes an interval of possible values for every register *r*. The interval for *r* is a correct approximation of a concrete program state if it contains the value of *r* in this program state. Suppose now a routine *R* is called twice, once with parameter 0 and once with parameter 3. Then the best abstract information that can be obtained for the parameter register is the interval $[0,3]$, which indicates that the value of the register might be 0, or 1, or 2, or 3. The precision of the analysis can be improved considerably if the analysis does not compute a single abstract value for each program point in *R*, but two different ones, one for each call of *R*. In the example considered above, these are the intervals $[0,0]$ for the call with parameter 0 and $[3,3]$ for the call with parameter 3. The values 1 and 2 are thus excluded successfully.

To be more general again, most of **AbsInt**'s analyzers compute an abstract information for every pair of a program point *p* and a possible *calling context* of *p*. All program points in a given routine

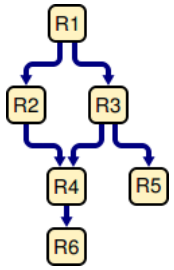


Figure 4.4: Call graph of an example program

R have the same set of calling contexts. Each calling context indicates a particular way of calling *R*.

Contexts are of particular importance for timing and value analysis. In contrast, stack analysis is initially performed with a single context for every routine. Only if this lack of context distinction prevents the calculation of the stack usage, a second round of stack analysis is started in which each loop iteration gets its own context. These stack analysis contexts are fixed and cannot be influenced by user annotations. The following description of contexts therefore applies – strictly speaking – to timing and value analysis only.

The following two subsections describe the exact structure of contexts, for programs without loops (4.3.1) and then for programs with loops (4.3.2).

4.3.1 Contexts of Routines (without Loops and Recursion)

This section introduces contexts in a simple program without loops. As an example, consider a program consisting of routines `R1`–`R6` whose call graph is depicted in Figure 4.4. One may expect that `R4` has two contexts since it apparently has two calls, one in `R2` and one in `R3`. But is this distinction inherited by `R6`, which is called by `R4`? The answer is yes, unless there is an artificial upper bound on the size of contexts. And what about `R5` if it is called *twice* in `R3`? (The fact that `R5` is called twice in `R3` cannot be seen from the call graph because there is at most one call edge between any two routines.) The answer is that `R5` has two contexts, too, because not the calling routine, but the address of the calling instruction is noted in the contexts. Thus, these addresses must be included in the example before we can present the contexts.

Example with call addresses:

```
R1 { 0x120: call R2; ...
     0x130: call R3; ...  }
R2 { 0x240: call R4; ...  }
R3 { 0x340: call R4; ...
     0x350: call R5; ...
     0x358: call R5; ...  }
R4 { 0x460: call R6; ...  }
R5 {  ...  }
R6 {  ...  }
```

Then the routines of the example have the following contexts:

```
R1:  no-history
R2:  0x120 → "R2"
R3:  0x130 → "R3"
R4:  0x120 → "R2",  0x240 → "R4"
     0x130 → "R3",  0x340 → "R4"
R5:  0x130 → "R3",  0x350 → "R5"
     0x130 → "R3",  0x358 → "R5"
R6:  0x120 → "R2",  0x240 → "R4",  0x460 → "R6"
     0x130 → "R3",  0x340 → "R4",  0x460 → "R6"
```

Here, each line shows one context, so `R3` has one context, but `R4` has two. In all routines but the entry routine (`R1`), the contexts are *call strings* indicating a sequence of calls starting from a call in the entry routine. Call strings consist of components of the form $A \rightarrow$ "R", where A is the address of a call to routine R.

Actually, A is not the address of the call instruction, but of the first instruction of the basic block ending with the call. The examples in this section are however based on the addresses of the call instructions.

If a routine R is "called" by a jump instruction (dummy call – see section 7.1.4), then the address A in $A \rightarrow$ "R" is not the address of the basic block containing the jump instruction, but the address of the "called" routine.

In this scheme, the entry routine would have an empty call string. This empty call string is denoted by the word `no-history` because the entry routine is not called by another routine and thus has no call history. (Actually, this context may also appear as `any-history`, and there may be three dots `...` in front of the other contexts.)

In programs with a complicated calling structure, the number of contexts may be too large for an efficient analysis. Therefore, a^3 offers a way to restrict the number of contexts by restricting the lengths of the call strings. An upper bound for these lengths is provided by the `interproc` parameter `max-length` in the AIS file (see section 5.4). If a call string grows longer than this parameter, it is shortened by omitting its first element. This may lead to an identification of contexts and thus to a loss of precision in the analysis. The example above was obtained with `max-length = inf`, which stands for an infinite bound, i.e. no restriction on the lengths of contexts.

A prefix `...` is used to indicate that some calls may be omitted at the beginning.

This prefix even appears if there is actually no omission since the call strings are too short. Thus it always appears if `max-length` is different from `inf`.

By these conventions, the context of the entry routine would appear as `..., no-history`. This string is replaced by `any-history`.

Examples:

The above example with `max-length = 2` leads to the following restricted contexts:

```
R1:  any-history
R2:  ...,  0x120 → "R2"
R3:  ...,  0x130 → "R3"
R4:  ...,  0x120 → "R2",  0x240 → "R4"
     ...,  0x130 → "R3",  0x340 → "R4"
R5:  ...,  0x130 → "R3",  0x350 → "R5"
     ...,  0x130 → "R3",  0x358 → "R5"
R6:  ...,  0x240 → "R4",  0x460 → "R6"
     ...,  0x340 → "R4",  0x460 → "R6"
```

The restriction `max-length = 1` leads to the following list of contexts:

```
R1:  any-history
R2:  ...,  0x120 → "R2"
R3:  ...,  0x130 → "R3"
R4:  ...,  0x240 → "R4"
     ...,  0x340 → "R4"
R5:  ...,  0x350 → "R5"
     ...,  0x358 → "R5"
R6:  ...,  0x460 → "R6"
```

Here, `R6` has merely one context; the two contexts that exist if the call string length is not restricted have been identified.

4.3.2 Contexts of Routines (with Loops and Recursion)

This section introduces the contexts in a program with loops (including direct recursion). Consider the following example:

```
R1 { 0x120: call R2; ...
     0x128: ...      ...
     0x130: call R3; ...
     0x138: goto 0x128;  }
R2 { 0x230: call R3; ... }
R3 { 0x340: call R4; ... }
R4 {  ...  }
```

It consists of routines `R1`–`R4`, with a loop in `R1`, which is extracted as loop routine `R1.L1` by the loop transformation (see section 4.1). The resulting call graph is depicted in Figure 4.5.

In principle, the contexts as described in section 4.3.1 could be used for loop routines, too. But then the context representing, say, the third iteration of `R1.L1` would be

```
0x128 → "R1.L1",  0x138 → "R1.L1",  0x138 → "R1.L1"
```

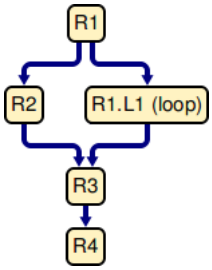


Figure 4.5: Call graph of an example program with a loop

which would cause rather long context strings that would be cut down soon under the `max-length` regime. Therefore, the initial call of the loop routine and the subsequent recursive calls are summarized into one call endowed with an iteration counter, namely `[1]` for the first iteration, `[2]` for the second, etc. The sample context from above is thus replaced by the shorter context `0x128 → "R1.L1"[3]`. (Actually, the total number of contexts is also shown, see below.)

The value of the iteration counter must be bounded to avoid an infinite number of contexts. By default, the number of iteration contexts equals the number of iterations of the loop as determined by automatic loop bound analysis or given by AIS annotations (section 5.20). If this number is for instance 5, the resulting 5 contexts are denoted as `[1/5]`, `[2/5]`, `[3/5]`, `[4/5]`, and `[5/5]`.

The number of iteration contexts can be reduced by the `interproc` parameter `max-unroll` in the AIS file (see section 5.4). For the following, we assume a bound of 3 for the number of loop contexts. Then the contexts `[1/5]` and `[2/5]` are renamed to `[1/3..]` and `[2/3..]`, and the contexts `[3/5]`, `[4/5]`, and `[5/5]` are mapped into a single context `[3/3..]`. Hence the routines of the example have the following contexts:

```
R1:     no-history
R2:     0x120 → "R2"
R1.L1:  0x128 → "R1.L1"[1/3..]
        0x128 → "R1.L1"[2/3..]
        0x128 → "R1.L1"[3/3..]
R3:     0x120 → "R2",             0x230 → "R3"
        0x128 → "R1.L1"[1/3..],   0x130 → "R3"
        0x128 → "R1.L1"[2/3..],   0x130 → "R3"
        0x128 → "R1.L1"[3/3..],   0x130 → "R3"
R4:     0x120 → "R2",             0x230 → "R3",  0x340 → "R4"
        0x128 → "R1.L1"[1/3..],   0x130 → "R3",  0x340 → "R4"
        0x128 → "R1.L1"[2/3..],   0x130 → "R3",  0x340 → "R4"
        0x128 → "R1.L1"[3/3..],   0x130 → "R3",  0x340 → "R4"
```

If the number of loop iterations equals 3, the labels `[`n`/3..]` appear as `[`n`/3]` since there is no 4th, 5th, etc. iteration. Otherwise, i.e. if the number of iterations is higher than 3, but the number of contexts is restricted to 3 by setting `max-unroll`, the two dots show up to indicate that the last context `[3/3..]` corresponds not only to the third, but also to all subsequent iterations.

A directly recursive routine, i.e. a routine calling itself, is handled in the same way as a loop. If `RA` calls `RR` at address A_1 and `RR` calls itself at address A_2, then the contexts of `RR` end in $A_1 \rightarrow$ `"RR"[1/`m`]`, $A_1 \rightarrow$ `"RR"[2/`m`]`, etc. Note that the address A_2 of the recursive call does not occur in these contexts.

Independently from the restriction of the loop contexts, the lengths of the call strings can be restricted by the `interproc` parameter `max-length` in the AIS file (see section 5.4) as explained in section 4.3.1. For instance, the above example with `max-length = 2` leads to the following contexts:

```
R1:     any-history
R2:     ..., 0x120 → "R2"
R1.L1:  ..., 0x128 → "R1.L1"[1/3..]
        ..., 0x128 → "R1.L1"[2/3..]
        ..., 0x128 → "R1.L1"[3/3..]
R3:     ..., 0x120 → "R2",            0x230 → "R3"
        ..., 0x128 → "R1.L1"[1/3..], 0x130 → "R3"
        ..., 0x128 → "R1.L1"[2/3..], 0x130 → "R3"
        ..., 0x128 → "R1.L1"[3/3..], 0x130 → "R3"
R4:     ..., 0x230 → "R3",            0x340 → "R4"
        ..., 0x130 → "R3",            0x340 → "R4"
```

Here, `R4` has only two contexts because the different loop iterations are no longer distinguished in this routine.

With `max-length = 1`, the amount of contexts is reduced further:

```
R1:     any-history
R2:     ..., 0x120 → "R2"
R1.L1:  ..., 0x128 → "R1.L1"[1/3..]
        ..., 0x128 → "R1.L1"[2/3..]
        ..., 0x128 → "R1.L1"[3/3..]
R3:     ..., 0x230 → "R3"
        ..., 0x130 → "R3"
R4:     ..., 0x340 → "R4"
```

4.4 Analysis Phases

a^3 supports several "top-level" analyses: stack analysis, value analysis, and timing analysis. Each of these analyses consists of several phases.

In timing analysis, **`exec2crl`** reads the executable and constructs its call graph and control-flow graph, *loop bound analysis* tries to determine bounds on the number of loop iterations (section 4.4.1), then the *actual value analysis* tries to statically determine the values stored in registers and memory cells (section 4.4.2), a combined *cache and pipeline analysis* computes the WCETs

of the basic blocks (4.4.3), and *path analysis* derives the overall WCET from the block WCETs (4.4.4).

Value analysis uses similar steps as timing analysis: it starts with **`exec2crl`**, runs loop bound analysis, and finally carries out the actual value analysis to compute the address ranges for memory accesses; cache and pipeline analysis as well as path analysis are not executed. The results of the actual value analysis can be interactively explored.

Calculation of the stack usage also starts with **`exec2crl`** reading the executable and constructing its call graph and control-flow graph. Then a simple stack analysis is tried in which all routines (including loop routines) have a single context. Only if this simple analysis fails because there are loops that increase or decrease the stack in every iteration, loop bound analysis as described in section 4.4.1 tries to determine bounds on the number of loop iterations. This knowledge together with user specified loop bounds is then used in a more sophisticated second run of stack analysis to obtain stack usage results even in presence of stack-modifying loops. More details can be found in section 7.3.

4.4.1 Loop Bound Analysis

Loop Bound Analysis is an optional feature of a^3; it may or may not be activated in the license file.

Timing analysis requires that upper bounds for the iteration numbers of all loops be known. Stack analysis requires upper bounds for loops that increase or decrease the stack in every iteration. Knowing upper bounds for the iteration numbers of loops can improve the precision of value analysis, e.g., if the loop iterates through an array. a^3 tries to determine the number of loop iterations by *loop bound analysis*, but succeeds in doing so for simple loops only. Bounds for the iteration numbers of the remaining loops must be provided by the user in the AIS file (see sections 5.20 and 5.22) or as source code annotations (section 5.31).

The results of loop bound analysis can be found in the report file whose name is provided in the `Files` view (see section 3.4.1), and in the first info fields of the loop call nodes (see section 7.4.4). User annotations in AIS format can be absolute or by default; the former supersede any results of the automatic loop bound detection. This means the automatic method comes into effect only for loops without absolute AIS annotations. User annotations marked by `default` are different; they have less priority than the automatic results.

Loop bound analysis itself is run with restricted contexts that do not distinguish between the various iterations of a loop. It consists of some pattern matching with the goal to identify the loop variable, and an analysis to predict its values.

If the executable has been compiled with `-O0`, then the code for even the simplest loops is so diverse that loop bounds are unlikely to be found automatically. Optimization leads to more uniform and more concise loop code, which simplifies the task of automated loop analysis.

The patterns used for loop bound analysis are partitioned into safe and unsafe loop patterns. The latter are used only if the flag `Use only safe patterns` in the `General` tab of the `Analyses` view is switched off (see section 3.4.3.4). Examples for unsafe loop patterns include linear search in an array for a particular element. The bounds derived from this pattern are only valid under the

assumption that the element searched is really part of the array. The usage of unsafe loop patterns is marked by a message in the message view and noted in the report file. Loop bounds obtained using unsafe loop patterns should be verified and overwritten by annotations if necessary (recall that there are annotations taking precedence over automatic loop bound analysis).

After loop bound analysis, the contexts are expanded to distinguish between loop iterations according to the parameters in the AIS file (see section 5.4). In doing so, the results of loop bound analysis and the user-specified loop bounds are taken into account, where the user specifications take priority over the analysis results in case of conflict except for those marked as `default`.

4.4.2 The Actual Value Analysis

The actual value analysis tries to statically determine the contents of the registers and memory cells at each program point for each context. Value analysis of memory cells is of minor importance for stack analysis. It can be suppressed by the flag `Do not analyze values of memory cells` in the `Advanced` tab of the `Analyses` view (see section 3.4.3.8).

The results of value analysis are used to predict the addresses of data accesses and to find infeasible paths caused by conditions that always evaluate to true or always evaluate to false (in a specific context). The knowledge of infeasible paths improves the efficiency and the precision of WCET analysis since the execution times along infeasible paths do not contribute to the true WCET and thus need not be considered in the calculation of an estimated WCET by a^3.

The distinction of contexts greatly improves the precision of value analysis. Consider for instance a routine `F` with an integer parameter `x` and two calls: `F(0)` and `F(y)` where `y` cannot be statically determined. Without contexts, the parameter `x` would be completely unknown. With contexts, `x` is known to be `0` in the context of the first call and unknown only in the context of the second call. Suppose further `F` contains an `if` statement with condition `x!=0` whose body needs 100 cycles to execute. Then the body of the `if` statement is infeasible in the context of the first call, which reduces the estimated WCET by 100 cycles compared to an analysis without context distinction or to a WCET analysis without value analysis.

4.4.3 Cache and Pipeline Analysis

The next phase of WCET analysis is a combined cache and pipeline analysis, which computes a WCET for every basic block. More exactly, it computes an estimated WCET for each basic block graph *edge* and each context. The estimated WCET for the edge from block B to block B' provides an upper bound for the execution time of block B provided that it is left via the edge to B'. Here, the execution time of B means the time needed to execute the instructions contained in B; it does not include the time needed to execute routines called from B. Consequently, loop call blocks – that do not contain code by themselves – have 0 execution time; thus edges starting from loop call blocks are not endowed with WCET annotations in the graph pictures generated by a^3.

To be more concrete, consider the graph in Figure 4.6 and assume that `F` has a single context, while the loop routine `F.L1` has 2 contexts: `[1/2..]` for the first iteration, and `[2/2..]` for all other iterations. Then the analysis computes one estimated WCET for the edge from `b0` to `loop call F.L1`, but two estimated WCETs for the edge from `b1` to `b2`, one for each context,

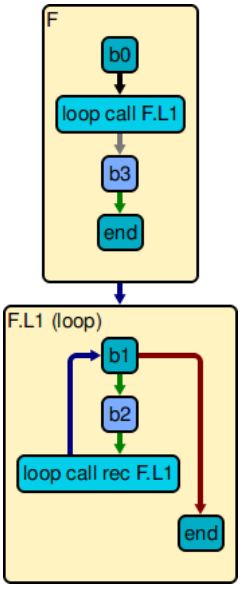


Figure 4.6: Examples for basic block graphs

and two estimated WCETs for the edge from `b1` to `end`. The estimated WCETs at the two edges leaving `b1` may be different because the execution time of the branch instruction at the end of `b1` may depend on whether the branch is taken ("true" edge to `b2`) or not taken ("false" edge to `end`).

Letting the estimated WCETs depend on the context may improve the precision of the analysis. Consider for instance the edge e from `b1` to `b2`. The estimated WCET for e in context `[1/2..]` is an upper bound of the execution time of `b1` in the first iteration, while the estimated WCET in context `[2/2..]` is an upper bound of the execution time in all other iterations. If there are 10 iterations and `b1` takes 20 cycles in the first iteration and 5 cycles in all other ones because of cache effects, then the total estimated WCET for `b1` is $1 \cdot 20 + 9 \cdot 5 = 65$ cycles thanks to the distinction between context `[1/2..]` and context `[2/2..]`. If there were no context distinction, the estimated WCET would be 20 for all iterations, giving a total estimated WCET of $10 \cdot 20 = 200$, which is quite a big overestimation.

4.4.4 Path Analysis

Path analysis uses the results of pipeline analysis, i.e. the estimated WCETs at the control-flow edges for all contexts, to compute a WCET estimation for the entire code that is currently analyzed.

Let $T(e,c)$ be the estimated WCET for edge e and context c as determined by the combined cache and pipeline analysis (see section 4.4.3 above). Furthermore, let $C(e,c)$ be the *execution count*, which indicates how often control passes along edge e in context c. If one knows for a specific run of the code the execution counts $C(e,c)$ for each edge e in each context c, then one can get an upper bound for the time of this run by taking the sum of $C(e,c) \cdot T(e,c)$ over all edge-context pairs (e,c). Thus, the task of obtaining a global WCET estimation can be solved by finding a feasible assignment of execution counts $C(e,c)$ to edge-context pairs that maximizes $\sum C(e,c) \cdot T(e,c)$. The value of this sum is then the desired global WCET estimate.

Therefore, path analysis is implemented by *integer linear programming*: The path analyzer

`pathan2` sets up a system of linear constraints over the integer variables $C(e,c)$. Then an LP solver looks for a solution of this constraint system that maximizes $\sum C(e,c) \cdot T(e,c)$. The constraints of the constraint system are derived from the control structure and from the loop bounds and flow constraints in the AIS file or in source code annotations (sections 5.20 and 5.22). Constraints derived from the control structure are for instance those that assert that the sum of the execution counts of the incoming edges of a block equals the sum of the execution counts of the outgoing edges.

The method sketched above explains some possible error messages of a^3:

- `This problem is unbounded.`
  This message indicates that there are not enough constraints so that for some edge and context, an arbitrarily high execution count can be chosen. This is usually caused by missing loop bounds, i.e. the analyzed code contains loops for which no maximum iteration number has been found by a^3 or has been specified by the user (see sections 5.20 and 5.22). In this case, watch out for messages of the form "Loop bound missing at start node *A*" to find the addresses of the loops without loop bound. (Yet notice that there is no "loop bound missing" message for loops with several entries that are not extracted by the loop transformation as described in section 4.1.4).

- `This problem is infeasible.`
  This message indicates that there are too many constraints, in particular constraints that contradict each other so that there is no choice of execution counts that would satisfy all constraints. This may happen if value analysis is able to find the true iteration count of a loop, and some user specification is inconsistent with the true iteration count. In this case, check your specifications and extend the range of possible iteration numbers. In particular, lower bounds do matter here although they are not important for the WCET.

  "This problem is infeasible" may also be caused by a non-terminating loop at the end of the application. Such a loop may for instance consist of an *unconditional* branch instruction branching to itself. The WCET problem is then infeasible because there is no path leaving the analyzed code snippet and thus path analysis cannot find a WCET path. Specifying a loop bound does not help in this case. A possible solution is to exclude the loop from the analyzed code by means of an end specification in the AIS file (see section 5.7).

- Some LP solvers do not always distinguish between these two cases. In this case, a^3 issues the following combined message:
  `This problem is infeasible or unbounded.`

4.5 Locating Source Files

Although the timing and stack analysis of a^3 are based on an executable, a^3 also reads the source files of the executable for various reasons. Source code may be displayed in special source-code views, as annotations in disassembly views (3.5.6), and as basic-block labels in the graph pictures of a^3 (7.1.5). If the flag `Extract annotations from source files` is activated in the

`General` tab of the `Analyses` view (see section 3.4.3.3), a^3 is scanning the source files for source code annotations, which are described in section 5.31. The algorithm for finding the source files is described in the following. It depends on various parameters:

- The setting of the flag `Strip compilation path mentioned in the executable` in the `General` tab of the `Analyses` view (see section 3.4.3.3);
- The include directories listed in the `Includes` panel of the `Source files` view (see section 3.4.5.1);
- The path replacement pairs listed in the `Replacements` panel of the `Source files` view (see section 3.4.5.2).

The algorithm for locating the source files is applied to all source file names listed in the debug information of the executable. For each source file name, the following steps are performed:

1. Some executable formats offer a field for specifying the compilation path of the application, but compilers do not always use this field. If a compilation path is specified in the executable and `Strip compilation path` is off and the file name is not already an absolute path name, the compilation path is added to the file name. Otherwise, i.e. if no compilation path is available or if the `Strip` option is on or if the file name is already an absolute path name, it is not modified in this step.

2. Now the path replacements specified by the replacement pairs are performed. The purpose of these pairs is to specify path prefixes and their replacements. The replacement pairs are tried from top to bottom. The first one that matches is executed; the remaining ones are not tried after a successful match. If no replacement pair matches, the file name is left unmodified.

   Normally, a path prefix is considered as complete directory name, i.e. there is an implicit separator / (Linux) or \ (MS Windows) at the end. In contrast, a path prefix followed by a wild card symbol $\star$ matches arbitrary prefixes of directory names.

   Example (Linux notation):
   If there is a replacement pair `/compile` $\rightarrow$ `/test`, then `/compile/source.c` is replaced by `/test/source.c`, but `/compile_dir/source.c` is not replaced by `/test_dir/source.c`. Yet if the replacement pair is `/compile`$\star$ $\rightarrow$ `/test`, then both replacements are performed.

3. If the file name resulting from Step 2 is absolute, it is not modified further. If it is a relative name R, the corresponding file is searched first in the directory where the executable resides, then in the specified include directories. The include directories are searched from top to bottom. The search in directory D is successful if the file R exists in D and is readable. The name of the first directory where the search is successful is prefixed to R, and the search stops. If the search is not successful in any of the considered directories, the resulting file name remains R, and a^3 will complain that this source file could not be found.

Chapter 5

AIS Annotations

a^3 may be supplied with various kinds of specifications and annotations that control its work and help it to do its job and to improve the precision of the results. The format of these specifications and annotations is called *AIS format*.

AIS information can be specified in the global *AIS file* named in the `Files` view (see section 3.4.1), or in the local AIS files named in the analysis views (see section 3.5.2). These AIS files may include further AIS files (see section 5.27). If enabled in the license file, AIS information may also be provided as source code comments (see section 5.31).

Declarations versus Assertions. AIS specifications come in two different flavors. Most specifications declare facts, e.g., this instruction accesses that memory area. a^3 can verify such *declarations* to some extent using its analysis results, but in general it cannot and does not perform this sort of verification; the idea of specifying declarations is to compensate for missing or imprecise analysis results, and the specified declarations always replace the results of automatic analysis. Some specifications declare *assertions*, e.g., this instruction should not access that memory area. Assertions are only used if value analysis is invoked as a separate analysis (see section 3.5.1). If a value analysis is started, a^3 tries to verify these assertions using its analysis results and complains if they are not satisfied; the idea of specifying assertions is to cause a^3 to verify that the software really meets its specifications. Declarations and assertions should be carefully distinguished; in particular, it is useless to specify declarations that logically imply assertions because a^3 then assumes the declarations as true and the assertions will thus be satisfied from the point of view of a^3 even if the software violates them.

Assertions are not used for timing and stack analysis.

If the information based on AIS declarations is not correct, the results of a^3 may become incorrect. Most declarations given by AIS specifications cannot be checked by a^3 for correctness.

General Structure. Possible contents of AIS files and AIS source annotations are specifications and directives. All specifications and directives are terminated by a semicolon '`;`'. AIS information has a free format: specifications and directives may extend over several lines, and several

specifications may be written into a single line (although this rarely is a good idea). The words and symbols of a specification or directive may be separated by arbitrary space, including empty lines.

There is one exception to the rule that line breaks do not matter: *Comments* are introduced by the character `#` and extend till the end of the line. If however `#` occurs in a quoted string (`"..."` or `'...'`), it is considered as part of that string, not as comment marker.

Specifications often refer to *program points*. In the simplest cases, program points are described by an address or a routine name. More complicated descriptions are also possible, e.g., the third computed call in a particular routine, or the loop beginning in a specific source code line. Such descriptions consist of more atomic elements like numbers, addresses, or names for routines or files. The following description is *top down*: It starts by describing complete specifications and directives in the following sections, using in examples only the simplest forms of program points. Then AIS expressions and their evaluation are described in section 5.28. Afterwards program points are described in full detail (section 5.29). Then the exact description of the atomic elements follows (section 5.30). Finally, AIS source-code annotations are introduced in section 5.31.

A remark about routine names in advance: Routine names must be written as in the executable, not as in the source code, i.e. with the leading underscore added by some compilers.

5.1 Overview

Below, we enumerate the subsections that describe the various specifications and directives possible in AIS files and as source code annotations.

- Subsections marked by **`S`** are relevant for stack analysis. Those marked by **`s`** describe annotations of minor importance for stack analysis; they are rarely needed, but maybe useful in special situations.
- Subsections marked by **`V`** are relevant for value analysis.
- Subsections marked by **`T`** are relevant for timing analysis.
- Subsections marked by **`***`** are relevant for all analyses in a uniform way.

5.2	**`T`**	the clock rate of the processor;
5.3	**`***`**	naming the compiler;
5.4	**`VT`**	specification of contexts;
5.5	**`T`**	declaring additional start points for timing analysis;
5.6	**`***`**	declaring that addresses are routine entries;
5.7	**`***`**	end specifications telling **`exec2crl`** to stop decoding;
5.8	**`***`**	control-flow: targets of computed branches, never-returning routines, etc.;
5.9	**`VT`**	addresses of memory accesses;
5.10	**`sVT`**	specification of register values and definition of user registers;
5.11	**`***`**	assertions, e.g., about register values;
5.12	**`***`**	properties of memory areas;
5.13	**`T`**	additional execution time;
5.14	**`T`**	requesting the WCET contribution of non-routine snippets;

5.15 *** code snippets that should not be analyzed and their properties;
5.16 *** violation of the calling conventions;
5.17 *** infeasible code, i.e. code that is never executed;
5.18 *** values of conditions;
5.19 SVT recursion depth and number of calls of recursive routines;
5.20 sVT iteration counts of ordinary loops;
5.21 T timing specification for loops;
5.22 T iteration counts of multi-entry loops – and more;
5.23 V commands that cause value analysis to produce certain output;
5.24 T commands that cause timing analysis to produce certain output;
5.25 *** introduction of symbolic names for program points and areas;
5.26 *** avoiding problems with unreachable program points;
5.27 *** include mechanism for AIS files.

Then AIS expressions and their evaluation are described in section 5.28. Thereafter, program points are described in full detail (section 5.29), followed by the exact description of the atomic elements of AIS specifications (section 5.30). Finally, AIS source-code annotations are introduced in section 5.31.

5.2 Clock Rate *(Timing Analysis)*

The clock rate is irrelevant for stack and value analysis.

a^3 is informed about the clock rate of the microprocessor by a clock specification, which has one of the following forms:

```
CLOCK  EXACTLY  n unit;
CLOCK  MIN  n1 unit1  MAX  n2 unit2;
```

where *unit* is `Hz`, `kHz`, or `MHz`. If $unit_1$ equals $unit_2$ in the second form, it may be omitted. The numbers n, n_1, and n_2 may be integers, floats, or fractions.

Examples:

```
clock  exactly  40 MHz;
clock  min 39.950  max 40.050 MHz;
clock  exactly 120/3 MHz;
```

(Keywords can be written in lower case or upper case.)

Remarks:

- There must be at most one clock specification. Multiple clock specifications are forbidden even if they specify the same clock rate.
- A clock specification enables you to talk about real time in later specifications. If there is no clock specification, you can only talk about cycles.

- If there is a clock specification, WCET results are displayed in cycles and in real time. Without a clock specification, only cycles are computed.

The AIS file from the `Files` view is read before any source-code files. Thus it is recommended to place a single clock specification at the beginning of the AIS file, which then also applies to all source-code files.

5.3 Naming the Compiler

The executable reader **`exec2crl`** works better if it knows the compiler that has produced the executable. Thus one of the following declarations should always be present:

```
COMPILER "tricore-tasking-v1";
COMPILER "tricore-tasking-v2";
COMPILER "tricore-gcc";
```

Here, the identifiers `"tricore-tasking-v1"` and `"tricore-tasking-v2"` refer to the Tasking C compiler for TriCore 1.3 processors, depending on which version of the compiler is used, and `"tricore-gcc"` to the GNU C compiler `gcc`. Depending on your license, some of these possibilities may not be supported.

5.4 Context Specification *(Timing Analysis, Value Analysis)*

Context specifications take effect in timing and value analysis only. They are irrelevant for stack analysis, which uses its own fixed context system.

Calling contexts have been introduced in section 4.3. Recall that a routine is analyzed for each of its contexts separately. Therefore, the time complexity of timing and value analysis increases with the number of contexts. On the other hand, the precision may increase if more contexts are distinguished. Therefore the AIS format admits specifications of parameters that influence the number of contexts.

There are global context specifications for defining global parameter settings and local context specifications for overwriting the global settings at individual loops.

The general form of a *global* context specification is as follows:

```
interproc flexible,
     max-length = n, max-unroll = m, default-unroll = l;
```

All three parameters are optional. They carry the following meanings:

- **`max-length =`** n with $n \geq 1$ restricts the lengths of call strings to n. Examples for the effect of this restriction have been presented in section 4.3. To specify that the call string length

is unbounded (potentially infinite), write **`max-length = inf`** or omit the **`max-length`** parameter (**`inf`** is default).

The chosen value of `max-length` holds for the entire analysis; local modifications are not possible.

- **`max-unroll =`** m with $m \geq 1$ provides an upper bound for the number of loop contexts. To specify that the number of loop contexts is unbounded (potentially infinite), write **`max-unroll = inf`** or omit the **`max-unroll`** parameter (**`inf`** is default).

  If `max-unroll` is `inf`, the number of loop contexts equals the maximum number of executions of the loop header, which is derived from AIS specifications (see section 5.20) or found by the automatic loop bound detection. Yet if a user-specified loop bound is too large and this is noticed by a^3, the number of contexts is derived from the real loop bound.

  The purpose of the `max-unroll` parameter is to limit the number of loop contexts to m, i.e. if the number of contexts as derived from the loop bound is greater than m, it is replaced by m. This is useful if there are loops with big iteration numbers.

  The `max-unroll` parameter can be modified for individual loops by local context specifications (see below). Thus the `max-unroll` parameter from a global context specification applies to all loops without a local specification.

- A declaration **`default-unroll =`** l defines a number of contexts for those loops whose maximum number of executions is not known to a^3. This number is still limited by `max-unroll`, i.e. the effective number of contexts is the minimum of m and l.

  The value `inf` is not allowed here. The `default-unroll` parameter can be omitted; the default value is `2`.

  The chosen value of `default-unroll` holds for the entire analysis; local modifications for individual loops are not possible.

The default values for missing parameters given above imply that an `interproc` specification

```
interproc flexible;
```

without any parameters is equivalent to

```
interproc flexible,
   max-length = inf, default-unroll = inf, max-unroll = 2;
```

If the `interproc` specification is missing altogether (no `interproc` in the AIS files), the behavior is quite different; a^3 assumes as default the specification

```
interproc flexible,
   max-length = 2, default-unroll = 2, max-unroll = 2;
```

Note that it is allowed to supply more than one global context specification. If the same file (either a global or a local AIS file) contains several such specifications (possibly via included AIS files)

only the one that is encountered by **exec2crl** at last takes effect; all previous declarations are ignored. If both the global and the local AIS file contain a context specification the declaration from the local file has precedence. This is useful if you have defined several analyses and a general context specification in the global AIS file but want to run some analyses with a different context setting.

Using *local* context specifications the `max-unroll` parameter from the global `interproc` specification can be overwritten at individual loops by writing

```
LOOP ProgramPoint max-unroll = m;
```

This annotation can be combined with a loop-bound specification as described in section 5.20.1 by means of `AND`:

```
LOOP ProgramPoint LoopBound
AND max-unroll = m;
```

The general format of *`ProgramPoint`*s is described in section 5.29. Section 5.20.1 presents the *`ProgramPoint`*s that are most useful in referring to loops.

Neither `default-unroll` nor `max-unroll` affect the number of loop *iterations* taken into account in analyses. They only affect the number of different loop *contexts*. If there are less iterations than contexts, the loop is not executed in some contexts, which therefore do not contribute to the analysis results. If there are more iterations than contexts, e.g., 8 iterations and 3 contexts, the loop is executed once in each of the specific contexts `[1/3..]` and `[2/3..]`, and 6 times in the summary context `[3/3..]`.

The limits described above apply to single loops. Nested loops may cause an explosion of the number of contexts. With `max-unroll` = 10 for instance and loops that iterate more than 10 times, the innermost loop in a loop nest of 3 loops has at least 1000 contexts (unless the call string length is limited to 2 or 1 by `max-length`).

The usage of a graph algorithm in path analysis (`Path analysis variant` set to `Prediction file based` in the `General` tab of the `Analyses` view, see section 3.4.3.7) is only successful if `max-length` and `max-unroll` are `inf` (or at least sufficiently high) so that each loop iteration and each incarnation of a recursive routine has a separate context.

5.5 Additional Start Points for Timing Analysis

It is possible to define additional start points for a timing analysis. These additional start points must be reachable from the main start point specified in the `Analysis start` text field of the analysis view defining the timing analysis (see section 3.5.2). There are several syntactic forms:

- `ROUTINE` *R* `IS ADDITIONAL START;`
  defines routine *R* itself as an additional start point.

- `ROUTINE` *R* `DESCENDANTS TILL LEVEL 0 ARE ADDITIONAL STARTS;`
  defines the routines directly called by *R* as additional start points (not *R* itself).
- `ROUTINE` *R* `DESCENDANTS TILL LEVEL 1 ARE ADDITIONAL STARTS;`
  defines the routines directly called by *R* and the routines directly called by these as additional start points (not *R* itself).
- `ROUTINE` *R* `DESCENDANTS TILL LEVEL` *n* `ARE ADDITIONAL STARTS;`
  is the general case: the routines reachable from *R* via at most *n* intermediate routines are additional starts.
- `ROUTINE` *R* `DESCENDANTS ARE ADDITIONAL STARTS;`
  causes all routines reachable from *R* (excluding *R* itself) to be additional starts.

Here, *R* may be a routine name, an AIS label, or an address. Additional start points can only be entries of *non-recursive routines* (ordinary routines, *not loop routines*). Loop routines are not counted as intermediate routines when the descendants till a certain level are determined.

When a timing analysis with additional entry points is started, a^3 first decodes the entire code reachable from the main start point specified in the `Analysis start` text field and then performs a value analysis on this entire code (see section 4.4 for the phases of a WCET analysis). Then it performs for each start point (main start point plus additional start points) its own cache and pipeline analyses followed by path analysis and result visualization. Each of these specialized analyses uses the value information computed from the main start point. The results of all these analyses are added to the report files.

Summarizing, we have one decoding, one value analysis, but multiple cache and pipeline analyses, path analyses, and result graphs. The result graphs for the additional start points can be opened by double clicks in the `Overview` view (section 3.5.4). They appear as subentries of the entry corresponding to the main start point and the entire analysis. The graph for the main start point carries the heading

`Computed Worst-Case Execution Times`

while the graph for the additional entry point *R* has heading

`Computed Worst-Case Execution Times for entry` *R*.

The result for an additional start point *A* includes the execution times for all routines reachable from *A*. It is thus similar to the cumulative WCET contribution of *A* in the analysis for the main start point *M* (section 7.4.2). The main difference is that the result for additional start point *A* is the WCET for one invocation of *A*, while the cumulative WCET of *A* in *M* is the sum of the estimated WCETs of all invocations of *A* along the WCET path of *M*. Thus the cumulative WCET contribution is about *n* times as high as the WCET for one invocation if *A* is called *n* times on the WCET path of *M*. In particular, the cumulative WCET contribution is 0 if *A* is not on the WCET path of *M*. Even if *n* is 1, the WCET for one invocation is in general higher than the WCET contribution since cache and pipeline analysis starts anew at the beginning of the analysis for an additional entry point.

5.6 Declaring Routine Entries

These specifications influence the combined call graph and control-flow graph underlying all analyses.

Usually, routine entries are found automatically by the executable reader **exec2crl**: if it encounters a call to an address *A*, it marks *A* as the beginning of a routine. The name of this routine is taken from the symbol table. If the symbol table does not contain a routine with address *A*, the routine is called `:Anon_`*A*.

The purpose of entry specifications is to declare more addresses as routine entries. The general form of entry specifications is as follows:

```
ENTRY  ProgramPoint;
```

It declares that the *ProgramPoint* is the beginning of a routine. The general format of *ProgramPoint*s is described in section 5.29. The simplest special case of this general format is

```
ENTRY  A;
```

where *A* is an address given as a number (see section 5.30.3 for the possible formats of numbers).

5.7 Stop Decoding

These specifications influence the combined call graph and control-flow graph underlying all analyses.

Decoding a routine normally ends when the routine is left. End specifications instruct the decoder **exec2crl** to stop reading a routine prematurely at a certain program point. The general format of an end specification is

```
END  ProgramPoint;
```

The simplest special case of this general format is

```
END  A;
```

where *A* is an address given as a number (see section 5.30.3 for the possible formats of numbers). The exact effect of this specification is that **exec2crl** stops decoding when the currently considered code address is exactly *A* – *before* trying to decode the instruction at address *A*. The current basic block is ended prematurely, and a new basic block with address *A* is created that does not contain any instructions. For the analyses, this block is treated as leaving the analyzed code snippet.

Example: *This example is generic, i.e. not specific to the TriCore architecture.*

Figure 5.1 (left) shows the basic block graph of a routine called `even`. It consists of two basic blocks, the first starting at address `0x50` with 6 instructions, and the second starting at address

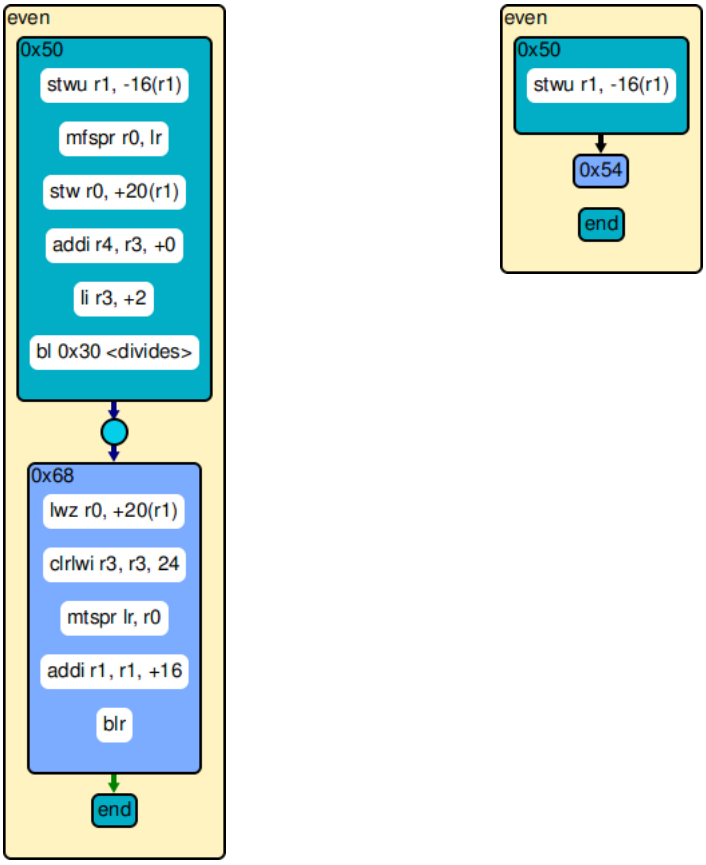


Figure 5.1: Effect of an end specification

`0x68` with 5 instructions. The box labeled by `end` signifies the routine exit. The second instruction of the first block has address `0x54`. Figure 5.1 (right) shows the basic block graph resulting from the specification

```
END   0x54;
```

Now, the first basic block ends after the first instruction and a new block with address `0x54` has been formed that does not contain any instructions (the `mfspr` instruction at address `0x54` and all further instructions of `even` are no longer decoded). There is no arrow from this block to the routine exit `end`. The parts of the executable behind the call to `even` are decoded, but considered as unreachable since a^3 assumes that `even` does not return because of the end annotation.

Therefore, an end annotation at the first instruction of a routine R yields a result very different from declaring R as not-analyzed (see section 5.15). Both annotations prevent a^3 from decoding R. The difference is that a not-analyzed routine may return as usual, while an end annotation that cuts off all paths from entry to exit of R makes a^3 believe that R does not return.

For the purposes of WCET analysis, blocks created by end annotations are considered as additional exit points of the analyzed part of the executable. This means that a^3 looks for the WCET

path among the paths leading from the start point of the analysis to either the end of the routine containing the start point or to any of the blocks created by end annotations.

In contrast to the example specification above, the specification `END 0x52` has no effect since the currently considered code address never is exactly `0x52`: it is `0x50` before decoding the `stw` instruction and `0x54` afterward.

End specifications are particularly useful if the analyzed code snippet contains a loop without exit, i.e. a loop formed by an unconditional branch.

While non-terminating loops can be handled by stack analysis without special user annotations, they require an annotation for timing analysis. Since there is no path leaving the loop, path analysis cannot find a WCET path, and thus a^3 states "This problem is infeasible". Specifying a loop bound for the non-terminating loop does not help. A possible solution is to remove the loop from the analyzed code by means of the end specification

```
END  Address;
```

where *Address* is the address of the loop.

5.8 Control-Flow Specifications

These specifications influence the combined call graph and control-flow graph underlying all analyses.

The task of the ancillary program **exec2crl** is to read the executable and to reconstruct the control flow of the application from it. **exec2crl** can find the target addresses of absolute and pc-relative calls and branches, but has difficulties with target addresses computed from register contents. It usually can recognize branches to a previously stored return address, and knows some typical compiler-generated patterns of branches via switch tables. Yet non-trivial applications usually contain some computed calls and branches that cannot be resolved by **exec2crl**; these unresolved computed calls and branches are documented by appropriate messages. In the graph pictures of a^3, routines and basic blocks containing unresolved computed calls can be recognized easily because the border of their boxes is red.

The AIS specification language offers various means to help **exec2crl** in performing its task. Section 5.8.1 explains **exec2crl**'s classification of instructions and ways to override it. Section 5.8.2 describes specifications listing the possible targets of computed calls and branches. Section 5.8.3 describes indirect target specifications telling **exec2crl** where to find the target addresses in the executable, e.g., in an array of function pointers or in a switch table. Section 5.8.5 shows how to specify additional properties of call instructions, e.g., this call never returns. Section 5.8.6 describes analogous specifications for routines, e.g., any call to this routine never returns. All these specifications are declarations, not assertions (see the discussion near the beginning of chapter 5).

5.8.1 Instruction Classes

exec2crl distinguishes two main classes of instructions: so-called *boring* instructions that can-

not alter the control flow, and control-relevant instructions that may perform control-flow alterations. The latter are classified further into *branches*, *calls*, and *returns*. A branch lets control jump to a different position, usually in the same routine. A call records a return address somewhere and then lets control jump to a different position, usually the entry point of a routine. A return instruction lets control return to a return address stored by a previous call instruction. Note that these classes are not disjoint; they provide "may" information, i.e. there are instructions classified as branch-or-call without **exec2crl** being able to decide between these two possibilities. The distinction between conditional and unconditional is orthogonal to the classification described above; for instance, a branch may be conditional or unconditional.

exec2crl's classification may be overridden by AIS specifications as follows:

```
INSTRUCTION ProgramPoint IS A CALL;
INSTRUCTION ProgramPoint IS A TAIL CALL;
INSTRUCTION ProgramPoint IS A BRANCH;
INSTRUCTION ProgramPoint IS A RETURN;
INSTRUCTION ProgramPoint IS BORING;
```

`CALL`, `TAIL CALL`, `BRANCH`, and `RETURN` specifications take effect only for instructions already classified as control-relevant. They introduce a new classification that completely replaces the original **exec2crl** classification; only boring remains boring. A `TAIL CALL` introduces the same classification as a `CALL`, but adds the information that this call does not return to the place after the tail call, but immediately to the place where the routine containing the tail call returns, i.e. `IS A TAIL CALL` is equivalent to `IS A CALL AND IMMEDIATELY RETURNS` (see section 5.8.5 for `IMMEDIATELY RETURNS`).

`BORING` specifications take always effect; they replace the original **exec2crl** classification whatever it was, and remove any control-flow alteration originally assumed by **exec2crl**.

`BORING` specifications are discouraged. They may confuse the analyzers, leading to wrong analysis results, or failure to produce any results at all.

5.8.2 Targets of Computed Calls and Branches

Targets of computed control-flow changes can be specified in the following forms:

```
INSTRUCTION ProgramPoint CALLS TargetList;
INSTRUCTION ProgramPoint BRANCHES TO TargetList;
INSTRUCTION ProgramPoint IS A RETURN;
```

The first form specifies call targets for a computed call, while the second form lists branch targets for a computed branch. The third form specifies that a computed branch is in fact a return instruction since the branch target is the return address of the present routine (see also section 5.8.1). These annotations are only necessary for those computed calls and branches that cannot be resolved automatically (these are listed in the message window and in the report file).

The application of these annotations to *non-computed* calls and branches is discouraged. They may confuse the analyzers, leading to wrong analysis results.

Here are some simple examples:

```
instruction  0xc0f8  calls  "disable";
instruction  0x9024  calls  0xa4, "go", "munch";
instruction  0x46c8  branches to  0x4710, 0x49a6;
instruction  0x37a0  is a return;
```

Several targets are separated by commas. Empty target lists are possible, yet note that they make the call or branch instruction infeasible. Targets may be addresses and routine names – and more, see below.

Routine names must be surrounded by double quotes (like all names).

Program points are not restricted to simple addresses (see section 5.29 for a full description of program points). A program point description particularly suited for `calls` and `branches` specifications is

"*R*" + *n* COMPUTED

which refers to the *n*th computed call or branch in routine *R* – counted statically in the sense of increasing addresses, not dynamically following the control flow. (This feature is subject to some restrictions – see section 5.29.) In a similar way, targets can be specified relative to a routine entry in the form

"*R*" + *n* bytes

Example: Assume routine `MC` contains 2 computed branches whose branch targets are specific instructions within `MC`. They can be specified as follows:

```
instruction "MC" + 1 computed
branches to "MC" + 0x5C bytes,
            "MC" + 0x6C bytes,
            "MC" + 0x7C bytes;

instruction "MC" + 2 computed
branches to "MC" + 0xA0 bytes,
            "MC" + 0xB0 bytes;
```

Alternatively, branch targets can be expressed relative to the address of the computed branch instruction, which is denoted by `pc`:

```
instruction "MC" + 1 computed
branches to pc + 0x04 bytes, pc + 0x14 bytes, pc + 0x24 bytes;

instruction "MC" + 2 computed
branches to pc + 0x10 bytes, pc + 0x20 bytes;
```

The advantage of such relative specifications is that they work no matter what the absolute address of `MC` is.

Such descriptions refer to the original extension of the routines *before loop transformation* (see section 4.1).

Specify targets of computed branches in-order, i.e. starting from the first. There is a high risk that **`exec2crl`** does not understand the specification for the second computed branch if the first computed branch has not been annotated before (see the gap problem at the end of section 5.29).

Program points of the kind R + n `computed` lead to an error if routine R is not reachable from the start point of the analysis. This error can be avoided by adding a `TRY` directive (see section 5.26).

5.8.3 Indirect Calls or Branches via Arrays of Function Pointers or Switch Tables

If the application contains an array `P` of function pointers, then a call `P[i](x)` may branch to any address contained in `P`. a^3 tries to obtain the list of these addresses automatically: If the array access and the computed call in the executable are part of a small code pattern as it is typically generated by the compiler, a^3 notices that the computed call is performed via this array. If furthermore the array contents are defined in a data segment so that they are statically available, and the array is situated in a ROM area so that its contents cannot be modified, then a^3 considers the addresses in the array as possible targets of the computed call (except for `NULL`, which is not considered as a call target).

If array access and computed call are too far apart or realized in an untypical way, a^3 cannot recognize that they belong together. Similar remarks apply to computed branches via switch tables. In both cases, the array or table belonging to the computed call or branch can be declared in the AIS file.

Sometimes, the situation is more complex because there are some steps of indirection: there is a master table containing not the function pointers themselves, but pointers to some other tables containing the function pointers, or pointers to yet other tables etc. The AIS declaration language can describe all these cases.

A declaration of calls or branches via tables starts like the ones presented in section 5.8.2. It has one of the two following forms:

```
INSTRUCTION ProgramPoint CALLS VIA Arrays;
INSTRUCTION ProgramPoint BRANCHES VIA Arrays;
```

A *`ProgramPoint`* may be an absolute address such as `0x8090`, a relative address such as `"main" + 8 bytes`, or a symbolic description such as `"main" + 2 computed`; more examples have been provided in section 5.8.2. A full account of all possible program-point descriptions is given in section 5.29.

The simplest case is that all function pointers are in a single table. The *Arrays* part then looks like

Start *Layout*$_1$

where *Start* describes the location of this table and *Layout*$_1$ its layout.

The next case is that there is a master table containing pointers to tables containing the function pointers. The *Arrays* part then looks like

Start *Layout*$_1$ -> *Layout*$_2$

where *Start* describes the location of the master table, *Layout*$_1$ its layout, and *Layout*$_2$ the layout of the tables containing the function pointers.

If there are two indirections, the *Arrays* part looks like

Start *Layout*$_1$ -> *Layout*$_2$ -> *Layout*$_3$

The AIS language also supports three and more indirections.

The *Start* part identifies the location of the master table by its name or start address. The start address may be an absolute address such as `0x9000` or an address relative to a routine or relative to the address of the computed call or branch, which is denoted by `pc`.

The layout descriptions *Layout*$_1$, *Layout*$_2$, ... consist of several attributes, some of which are optional. These attributes must be provided in the same order as in the following list.

- The first information describes how many and/or which elements of the array should be considered. There are several possibilities to provide this information.
  - `[` *n* `]`
    specifies the number of array elements. All these elements are considered by a^3.
  - `terminated by null`
    tells a^3 to continue looking at elements until a null pointer is found.
  - `indices` i_1`,` i_2`, ...`
    instructs a^3 to look specifically at the given indices, which may be positive or negative. Index 0 corresponds to the start address of the array.
  - This information is optional. If none of the above is specified, a^3 tries to extract the number of array elements from the symbol table in the executable and looks at all those elements.

  Examples:
  `indices 2, 3, 11` tells a^3 to look at the array elements with index 2, 3, and 11.
  `[4]` tells a^3 that there are 4 array elements and instructs it to look at all of these. It is therefore equivalent to `indices 0, 1, 2, 3`.

- `struct` *Number* `bytes`
  specifies the size of a single array element in bytes. This information is optional. If it is missing, a^3 tries to extract the information from the symbol table in the executable.

- `offset` *Number* `bytes`
  specifies the offset of the address information inside the array elements. An offset of 0 bytes may be omitted together with the keyword `offset`.

- `access` *Number* `bytes`
  specifies the number of bytes to be extracted from each array element, starting from the specified offset. This information is optional. If it is missing, a^3 assumes the standard address width of the processor.

- `little endian` or `low byte first` or `lbf`
  means the information to be extracted is stored with low byte first (lbf).

  `big endian` or `high byte first` or `hbf`
  means the information to be extracted is stored with high byte first (hbf).

  This attribute is optional and only needed if the endianness is not specified in the executable.

- `mask` *M*
  means the extracted information is filtered through the bit mask *M* by means of bitwise AND. Here, *M* is a number that can be written in binary notation as `0b...`.

  This information is optional. The default bit mask is `0b1...1`, i.e. all of the extracted information is processed further.

- `scale` *S*
  indicates that the number derived from the array element must be multiplied by the scaling factor *S* to become a target. The factor $S = 1$ is default and may be omitted together with the keyword `scale`.

- `absolute`
  means the extracted information yields an absolute address. Absolute addresses are default; so the keyword `absolute` may be omitted.

  `relative to` *ProgramPoint*
  `relative to` *ProgramPoint* `signed`
  both mean that the extracted information is an address relative to the given *ProgramPoint*. In this case, the final address results from adding the extracted number (after an optional sign extension) to the address of the *ProgramPoint*. The *ProgramPoint* may – among other things – be `pc` meaning the address of the computed call or branch instruction whose targets are being described.

- The last attribute of an *ArrayDescriptor* is the *NullFlag*, which is either `null` or `not null`. The *NullFlag* is optional; default is `not null`. Selection `null` means that `NULL` addresses found in the array are not considered as target addresses, but ignored as in the cases that can be handled automatically. Selection `not null` means that `NULL` is considered as an ordinary call target.

To obtain the same behavior as in the case where a^3 recognizes the array by itself, the *NullFlag* must have value `null`.

The `struct`, `offset`, and `access` attributes must be specified in *bytes*. Other units such as words are not allowed at these syntactic positions.

Example 1: The first computed call or branch in `main` is a call via array `A` with 10 elements. Each array element is a target address of 4 bytes. The number and size of the array elements can be read off from the symbol table. There are no `NULL` entries in this array.

```
instruction  "main" + 1 computed  calls  via "A";
```

This specification is of minimum size; all optional information has been omitted.

Example 2: The computed branch at relative address `0x120` in `R` is performed via a switch table located 4 bytes after the branch instruction. The table consists of 6 relative addresses of 1 byte size. These addresses are relative to the position just after the table, i.e. 4 + 6 bytes after the branch instruction.

```
instruction  "R" + 0x120 bytes
  branches via  pc + 4 bytes  [6]  struct 1 byte
    access 1 byte  relative to pc + 10 bytes;
```

Example 3: The call at address `0x9000` is via array `B` with 17 elements. Each array element is a record of 12 bytes. The number and size of the array elements is not specified in the symbol table. The size of the target addresses is 4 bytes. The addresses start at byte 4 of the records, i.e. occupy bytes 4–7 of the records. `NULL` entries should be ignored.

```
instruction  0x9000  calls via  "B" [17]
  struct  12 bytes
  offset   4 bytes   access  4 bytes
  null;
```

Example 4: The call at address `0x9500` is via a master table `T` with 4 elements that are pointers to null-terminated arrays of function pointers.

```
instruction  0x9500  calls via
  "T" [4] -> terminated by null;
```

Example 5:

```
instruction "handleMessage" + 1 computed calls via
  # array of pointers to structs, possibly containing NULL
    "inputMessagesArray"  struct 4 bytes  null  ->
  # struct, where the first element points to another struct
    struct 16 bytes  access 4 bytes  ->
  # the second entry of the struct contains the function pointer
    struct 50 bytes  offset 8 bytes  access 4 bytes;
```

An indirect target declaration via arrays or tables takes effect only if all involved arrays or tables are situated in ROM areas.

a^3 can read off the ROM property from the executable, but the compiler does not mark all ROM areas as such. If a ROM area is not marked in the executable, you can specify the ROM property in the AIS file (see section 5.12).

The ROM area specification must be written *before* the indirect target declaration it belongs to.

5.8.4 Target Check

Normally, a^3 performs a target check: it uses the results of value analysis to check whether all targets of computed calls and branches and of returns can be reached, and marks the edges to the unreachable targets as infeasible. This is for instance useful if a computed branch implements a switch statement in a loop with the property that the ith case of the switch is taken in the ith iteration:

```
for (i = 0; i < 3; i++) {
  ...
  switch (i) {
    case 0: ... break;
    case 1: ... break;
    case 2: ... break;
  }
  ...
}
```

Without target check, a^3 would assume that the case with the highest WCET is taken in all three iterations. With target check, a^3 realizes that case i is taken in the ith iteration, provided that each iteration has its own loop context (see section 4.3.2) and value analysis can track the values in the register corresponding to `i`.

The target check is also used to control to some extent the validity of the list of targets given by annotations as described in sections 5.8.2 and 5.8.3. If the target check indicates that all possible targets are infeasible according to the results of value analysis, a^3 issues a warning.

In some situations however, you may wish to add a target that should be analyzed although it is unreachable, or you may want to replace the list of proper targets entirely by some other list that should be analyzed. This is only possible if the target check is switched off for this call or branch, which can be done by

```
INSTRUCTION ProgramPoint SKIP TARGET CHECK;
```

where *`ProgramPoint`* is the position of the branch, call, or return for which the target check should be skipped.

The annotation can be combined with a target specification as described in sections 5.8.2 and 5.8.3 by means of AND, e.g.,

```
INSTRUCTION ProgramPoint CALLS TargetList
                         AND SKIP TARGET CHECK;
```

5.8.5 Properties of Call Instructions

Normally, a call instruction transfers control to a routine, from which it returns to the instruction after the call. This default behavior may be modified in various ways. The following specification for instance indicates a non-standard return address:

```
instruction "main" + 1 call returns to pc + 8 bytes;
```

It is possible to list several different return targets.

A "returns to" specification can only be applied to an instruction classified as a possible call (see section 5.8.1). If this is not the case, the necessary reclassification can be combined with the "returns to" specification as follows:

```
instruction "main" + 1 branch is a call
    and returns to pc + 8 bytes;
```

Maybe, the return address stored by a non-standard call is the return address of the routine containing the call. The effect is that a return of the routine called behaves as if it had triggered an immediate return of the routine containing the call. This behavior can be specified as follows:

```
INSTRUCTION ProgramPoint IMMEDIATELY RETURNS;
```

Finally, it may happen that a call never returns, e.g., a call to a routine of the operating system. This can be specified as follows:

```
INSTRUCTION ProgramPoint NEVER RETURNS;
```

The next section (5.8.6) contains related features for routines, and some further remarks about the "never returns" property.

5.8.6 Properties of Routines

The specifications in section 5.8.5 apply to single calls. There are related specifications for routines that apply to *all* calls of the routine under consideration.

```
SNIPPET "R" NEVER RETURNS;
```

informs **exec2crl** that routine R does not return when called. Thus, **exec2crl** does not read code textually following a call to R unless this code is reached from somewhere else.

A specification `SNIPPET "exit" NEVER RETURNS` is not needed; a^3 knows by itself that `exit` does not return.

The no-return property is not propagated automatically: If R ends by calling R' and R' does not return, then R does not return either, but a^3 does not infer this property. Hence it does not suffice to declare that R' never returns; R should be declared as never returning, too.

Omitting a never-returns declaration for a non-returning routine R does not necessarily lead to an error. It usually causes a loss of precision since a^3 unnecessarily considers the code after the calls of R during its analyses. An error results if a call of R is followed by data whose decoding leads to nonsense code or illegal-instruction errors.

```
SNIPPET "R" IMMEDIATELY RETURNS;
```

informs **exec2crl** that routine R causes its caller to return immediately: if some R' calls R, then R' will return immediately after R has returned. The reason for such behavior may be that R removes its own return address so that its return instruction branches to the return address of R', thus causing R' to return immediately.

5.9 Addresses of Memory Accesses *(Timing Analysis, Value Analysis)*

Addresses of memory accesses are irrelevant for stack analysis.

The actual value analysis, which is part of timing analysis and value analysis, tries to determine the addresses of memory accesses. In cases when it fails to obtain exact addresses, you may supply specifications of exact addresses or possible address ranges. These specifications are only relevant for timing and value analysis. They fall into two classes: local specifications valid for a single instruction (sections 5.9.1 and 5.9.2), and global specifications valid for entire routines (section 5.9.3).

5.9.1 Local Address Specifications

To specify addresses of memory accesses in single instructions, you may write specifications of the form

```
INSTRUCTION ProgramPoint ACCESSES RangeList;
```

The intention of these specifications is to restrict imprecise value analysis results to a smaller range of possible values, which may be an exact value in the extreme case. Let U be the set of addresses specified by the AIS specification, and let V be the set of addresses computed by value analysis.

These two address sets are combined into a result set R that is communicated to pipeline analysis. If U is a subset of V, then R is U, i.e. the annotation wins. If U is not a subset of V, then a^3 issues a warning. If in this case U and V have something in common, then R is the intersection of U and V. If U and V are disjoint from each other, then R is V, i.e. the result of value analysis wins. So these specifications are declarations, not assertions (see the discussion near the beginning of chapter 5). Section 5.12 describes related specifications of assertions that express that certain instructions should only access certain memory areas and cause a^3 to complain if they fail to do so.

Useful *ProgramPoint* formats for such specifications are simple instruction addresses and routine-relative expressions of the forms

"*R*" + *n* READS and "*R*" + *n* WRITES

meaning the *n*th instruction in routine *R* reading from memory and the *n*th instruction writing to memory, respectively.

A ***RangeList*** is a comma-separated list of ranges. A range may be one of the following:

- *Position*
  is a single position in memory, e.g., an address (see below for other possibilities);
- FROM *Position* TO *Position*
  Position .. *Position*
  are two equivalent ways to specify a memory area given by its start and end position (both belonging to the area);
- "*A*"
  denotes the memory area covered by array A.

A single position denotes the start address of the access. If you know the exact address of, say, a word access, you need (and should) not specify a range of 4 bytes, but only the start address of the access. In contrast, a range spreading over several addresses expresses some degree of uncertainty. It should be specified not as a range of start addresses, but as a range covering the full width of all possible accesses.

A *Position* may be an address in memory or an expression of the form

"*A*" + *n* bytes

denoting a specific byte in the memory area covered by array A. Such positions can be used not only for byte accesses, but also for halfword, word, or larger accesses; in these cases, a byte position greater 0 denotes the start address of the accessed piece of memory.

If the *Position* .. *Position* form is used to specify a memory area, it is quite common that both positions are expressed w.r.t. the same start point. In this case the following abbreviation can be used:

Position .. *Position* + *n* bytes

is equivalent to

```
Position .. + n bytes
```

Examples: Assume array `TAB` is mapped to memory area `0x8100–0x81FF`, and the first read instruction in routine `main` has address `0x8500`. Then the following specifications are equivalent:

```
instruction 0x8500 accesses 0x8100 .. 0x81FF;
instruction 0x8500 accesses "TAB";
instruction "main" + 1 read accesses "TAB";
```

The following specifications are also equivalent to each other, but different from the ones above:

```
instruction 0x8500 accesses 0x8100;
instruction 0x8500 accesses start "TAB";
```

While address ranges in the AIS file cover the full width of all possible accesses, address ranges in the report file indicate the possible range of start addresses followed by the width of the access. For instance, a range specified as `0x1000..0x1fff` in the AIS file appears as `[0x1000..0x1ffc]:4` in the report file if the access is a word access (4 is the access width, and `0x1ffc` is the start address of the last access that entirely fits into `0x1000..0x1fff`).

If value analysis found an exact address A for a load instruction and A points to a ROM data section, then it reads the value at A and uses it for obtaining further results. *This happens even if there is a user annotation specifying an address A' different from A.* Yet for timing analysis, the access time of A' is considered, not that of A. User annotations for memory accesses do not influence value analysis itself, but only timing analysis performed by cache and pipeline analysis. This not only holds for the annotations described above, but also for the global address specifications introduced below (section 5.9.3).

5.9.2 Local Address Specifications for a Base Register

There is an alternative form of local address specifications that looks as follows:

```
INSTRUCTION ProgramPoint ACCESSES Range FOR BASE Register;
```

Processor registers are specified by name (`A0–A15` or `D0–D15`).

These specifications are similar to those described in section 5.9.1, but the following important differences exist:

- An annotation involving `FOR BASE` *R* takes effect only for accesses using register *R* as base register.

- `FOR BASE` annotations do not admit an address range list, but only a single address range.
- In an annotation without `FOR BASE`, the specified address ranges replace the value analysis result. ⚠ If `FOR BASE` is present, the behavior is entirely different: The specified address range is intersected with the value analysis result. For instance, if the value analysis computes `0x80..0x100` and the user specifies `0x10..0xff`, the result is `0x80..0xff`.
- If the intersection is empty, a warning is issued and the user-specified range is taken.

5.9.3 Global Address Specifications

Global address specifications provide information about all memory accesses in a routine or even in the entire executable. These are declarations, not assertions (see the discussion near the beginning of chapter 5). Section 5.12 describes related specifications of assertions telling that some routines should or should not access certain memory areas, causing a^3 to complain if it finds something different.

Global address specifications have the following general form:

```
Dom1  Kind1  Qualifier  Dom2  Kind2  TO RangeList;
```

Each annotation applies to a certain part of the executable only, called the *application domain* of the annotation. The application domain is defined by *Dom1* and *Dom2* together. There are three choices:

```
         ...          Routines  ...;
GLOBAL   ...                    ...;
GLOBAL   ...  EXCEPT Routines  ...;
```

- In the first form, *Dom1* is empty and *Dom2* is *Routines*, a comma-separated list of routine names. The application domain of this form consists of the code of the listed routines (including all loop routines extracted from these routines).
- In the second form, *Dom1* is the keyword `GLOBAL` and *Dom2* is empty. The application domain of this form consists of the entire executable (as far as it is analyzed).
- In the third form, *Dom1* is the keyword `GLOBAL` and *Dom2* consists of the keyword `EXCEPT` followed by a comma-separated list of routine names. The application domain of this form is the entire executable minus the listed routines.

Kind1 and *Kind2* together determine to which accesses the annotation applies. There are four possible choices:

```
...  READS      ...                      ...;
...  WRITES     ...                      ...;
...  ACCESSES   ...                      ...;
...  ACCESSES   ...  FOR BASE Register   ...;
```

Kind2 is non-empty only in the last case. In this case, *Kind1* is restricted to ACCESSES. An annotation with ACCESSES (third case) applies to all data accesses in its application domain, while annotations with READS or WRITES apply to read accesses or write accesses only. The last form (ACCESSES ... FOR BASE *Register*) applies to all accesses that use the given register as base register. *Processor registers* are specified by name (A0–A15 or D0–D15).

A ***RangeList*** is a comma-separated list of ranges. These ranges have the same syntax and meaning as the ranges used in INSTRUCTION ... ACCESSES annotations (see section 5.9.1).

If FOR BASE is used, the *RangeList* should consist of a single range only.

The ***Qualifier*** is either restrict or default. An annotation with *Qualifier* = restrict applies to all accesses of the specified kind in the application domain, while an annotation with *Qualifier* = default only applies to the accesses for which no information at all has been found by value analysis. These are the accesses with address range [?] in the report file.

The combination of DEFAULT with FOR BASE has not been implemented.

With 3 choices for the domain (*Dom1* and *Dom2*), 4 choices for the kind (*Kind1* and *Kind2*), and *Qualifier* = RESTRICT, the following 12 syntactic forms exist:

```
READS     RESTRICT  Routines  TO RangeList;
WRITES    RESTRICT  Routines  TO RangeList;
ACCESSES  RESTRICT  Routines  TO RangeList;
ACCESSES  RESTRICT  Routines  FOR BASE Register  TO Range;
GLOBAL  READS     RESTRICT  TO RangeList;
GLOBAL  WRITES    RESTRICT  TO RangeList;
GLOBAL  ACCESSES  RESTRICT  TO RangeList;
GLOBAL  ACCESSES  RESTRICT  FOR BASE Register  TO Range;
GLOBAL  READS     RESTRICT  EXCEPT Routines  TO RangeList;
GLOBAL  WRITES    RESTRICT  EXCEPT Routines  TO RangeList;
GLOBAL  ACCESSES  RESTRICT  EXCEPT Routines  TO RangeList;
GLOBAL  ACCESSES  RESTRICT  EXCEPT Routines
                            FOR BASE Register  TO Range;
```

There are another 9 syntactic forms with RESTRICT replaced by DEFAULT. (These are 9 instead of 12 forms since DEFAULT excludes FOR BASE.)

Example 1: Assume that the address range 0x20000000 .. 0x3fffffff is mapped to slow memory, which is accessed in routines slow1 and slow2 only. Then you may specify

```
global accesses restrict except "slow1", "slow2"
       to 0x0..0x1fffffff, 0x40000000..0xffffffff;
```

Example 2: In routine mark, value analysis can predict all accesses via the stack, but cannot obtain any information about the other accesses. You want to specify that these other accesses all go into array A. Since this array is disjoint from the stack, a restrict annotation leads to contradictions, but a default annotation works:

```
accesses default "mark" to "A";
```

Example 3: Assume you know that in the entire program all accesses from base register `A0` go into the area `0x1000..0x1fff`. Then you may specify

```
global accesses restrict for base A0 to 0x1000..0x1fff;
```

5.9.4 The Evaluation of Address Specifications

In this section, "value analysis" means the actual value analysis, which is part of timing analysis and of value analysis.

The addresses considered by a^3 for a memory access A are influenced by value analysis and by user annotations. Their calculation is performed in the following steps:

1. Value analysis determines a set of possible addresses for A. This set may contain a single address (exact prediction), more than one, but not all addresses (inexact prediction), or all addresses (no information). In the last case, the set is denoted by `[?]` in the report file.
2. If an `INSTRUCTION ACCESSES` annotation with or without `FOR BASE` applies to A, the result of value analysis is *intersected* with the addresses specified in that annotation. If the result is empty, a^3 issues a warning and takes

   the *user-specified range* if `FOR BASE` is present, but the result of *value analysis* if there is no `FOR BASE`.
3. If value analysis did not obtain any information and there was no `INSTRUCTION ACCESSES` annotation for A, the global `DEFAULT` annotations applying to A are taken into account (otherwise, the result of Step 3 is the same as the result of Step 2). An annotation applies to A if A is in its application domain and has the appropriate access kind. If there is no `DEFAULT` annotation applying to A, the address set as determined by Step 2 remains unchanged, i.e. `[?]`. If there is at least one, the result set of Step 3 is the union of the address sets of the relevant `DEFAULT` annotations.
4. Now the global `RESTRICT` annotations applying to A are taken into account. If there is none, the address set as determined by Step 3 is the final result. If there is at least one, the final result is the intersection of the result set of Step 3 and the union of the address sets of the `RESTRICT` annotations applying to A.

If the intersection in Step 4 results in the empty set since the annotations are inconsistent, a^3 issues a warning and marks the memory access as infeasible. The final result set is documented in the report file whose name is specified in the `Files` view (see section 3.4.1). If a `DEFAULT` annotation has been applied to obtain the final result, the documentation contains a remark telling that this happened.

Value analysis is performed separately for each execution context, but AIS specifications uniformly apply to all contexts. So it may happen that value analysis finds an exact result for Context 1 and `[?]` for Context 2. Then default annotations take only effect in Context 2, not in Context 1.

So the final set of addresses for an access depends on the execution context in general. Internally, these sets are passed to timing analysis (cache and pipeline analysis) separately for each context to avoid a loss of precision. Yet the sets documented in the messages of a^3 and in the report file are joined over all contexts. Thus, they often appear to be less precise than they are in reality.

The concept of execution contexts is introduced in section 4.3.

5.9.5 Alignment Specifications

Apart from address ranges, it is also possible to specify that certain accesses are aligned. A memory access of width 2^k is *aligned* if the lowest k bits of the accessed memory address are 0.

Alignment specifications take one of the following forms:

```
INSTRUCTION ProgramPoint HAS ALIGNED ACCESS;
INSTRUCTION ProgramPoint HAS MISALIGNED ACCESS;
```

The `ALIGNED` form declares that the access is definitely aligned. The purpose of the `MISALIGNED` form is to overwrite a global alignment assumption given by setting the flag `Assume aligned data accesses` in the `Analyses` view (section 3.4.3.4). It means that the access may be aligned or misaligned (not that it is definitely misaligned).

Alignment specifications can be combined with local address specifications as described in section 5.9.1 by means of `AND`:

```
INSTRUCTION  ...  HAS ALIGNED ACCESS  AND  ACCESSES ...;
```

Alignment information is evaluated as follows:

- An access is assumed to be aligned in two cases:
  - There is an `ALIGNED` annotation referring to this access;
  - the flag `Assume aligned data accesses` in the `Analyses` view is set (section 3.4.3.4), and not overwritten by a `MISALIGNED` annotation referring to this access.
- If an access is assumed to be aligned, the set of possible addresses of the access as computed by the value analysis of a^3 is diminished by removing all non-aligned addresses from it.
- If the resulting set is non-empty, it is taken as the set of possible addresses. Yet if the resulting set is empty, which means that the access is definitely not aligned according to the knowledge of a^3, a warning is issued and the original set is taken, i.e. the alignment assumption is ignored.

Thus, a^3's own knowledge has highest priority, followed by the AIS annotations described here, followed finally by the global flag, which has least priority.

5.10 Specification of Register Values and Definition of User Registers

The annotations presented in this section serve two purposes:

1. They can be used to specify the values of processor registers at a program point. Analogous annotations for specifying memory values at a program point are presented in section 5.12.5. Information about the values of some registers at some program point may imply that some conditions are always false or always true, and therefore some paths are never executed. These paths then do not contribute to the results of the analyses, improving their precision. Register values are also used to determine the addresses of memory accesses. Thus, the specification of known register values can reduce the number of unknown addresses of accesses, leading to more precise timing results in presence of memories with different access times.

2. The annotations can also be used to assign values to so-called user registers. These are variables that the analyzers handle like additional registers not corresponding to any concrete processor registers. They can be given arbitrary names marked by an initial `@`. The values assigned to user registers can later be used in the definition of values of processor registers and of further user registers (this section), in the specification of additional execution time (section 5.13), the declaration of non-executed control-flow paths (section 5.17.2) and of the values of conditions (section 5.18.2), and in the definition of loop bounds (section 5.20). Sections 5.17.2 and 5.20 contain some examples for the usage of user registers.

The actual value analysis, which is part of timing analysis and value analysis, tries to determine the values of the processor registers for every program point and execution context. Thus, the declaration of the value of a processor register at a program point is meant to refine the results of value analysis.

Value analysis does not assign values to user registers by itself; such assignments are only possible with the declarations presented in this section. Of course, user registers are never modified by the assembly instructions of the analyzed program. Nevertheless, value analysis handles user registers like processor registers: it propagates their values through the program following the control flow, taking into account execution contexts (described in section 4.3). When a user register has two different values at a control-flow join, these values are combined into one by value analysis. Consider for instance the following piece of code:

```
if (C) { A; R(); }
else   { B; R(); }
D;
```

- If `@X` is set to 1 in `A` and set to 2 in `B`, then it has two different values in the body of the routine `R`: value 1 in the context of the call after `A` and value 2 in the context of the call after `B`. The two values are combined at the beginning of `D` to the set $\{1,2\}$, which is the value of `@X` within `D` and thereafter.

- If `@X` is set to 1 in `A` and this is the only setting (no setting in `B` or anywhere else), then `@X` has value 1 in `R` in the context of the call after `A`, but is undefined in `R` in the context of

the call after `B`. It is also undefined within `D` and thereafter since an undefined value and a defined value are combined to undefined.

The general form of a declaration of register values is as follows:

```
INSTRUCTION  ProgramPoint  IS ENTERED WITH
   Register1 = Expr1, ..., Registern = Exprn;
```

The *ProgramPoint* can be an absolute address or an address relative to a routine entry, or something more complicated (see section 5.29).

A *Register* is either a processor register or a user register. *Processor registers* are specified by name (`A0`–`A15` or `D0`–`D15`).

The names of *user registers* are arbitrary identifiers, marked by an initial `@` symbol, e.g., `@a`, `@userreg1`, `@bound`. The `@` symbol can only appear at the beginning; `@x@prime` is not a legal user register name.

An *Expr* is an expression that evaluates to a set of possible values (see section 5.28.2 for details). In general, an expression is built up from references to processor registers or user registers, addresses of various kinds, and constants by means of various operators, including arithmetic operators. The full syntax and semantics of expressions is detailed in section 5.28. Here we present only a few examples. An expression may be among others:

- A number with an optional sign −. Numbers in AIS may be decimal, hexadecimal, octal, or binary (see section 5.30.3). Small numbers are sign-extended to the size of the register. Numbers that are too large for the register size are cut down by omitting the most significant bytes.
- An interval written in the form `(`*A*`..`*B*`)` where *A* and *B* are numbers as described above. This is a closed interval, i.e. *A* is the smallest possible value and *B* the largest possible value of the register.
- In both cases, the number may be replaced by *R* or `reg` *R* where `reg` is an optional keyword and *R* a register name (processor register or user register). The meaning of this expression is the current value of the processor register as determined by value analysis, or the last value assigned to the user register. This value may be an exact value (a single number) or a set.
- The keyword `address` followed by a program point stands for the address of the program point. Complex program points must be written in parentheses.
- The keyword `address` followed by an area description stands for the interval of addresses in the area. An area description may be a single program point, a pair of program points `from` *Start* `to` *End*, or *Start* `..` *End*, or an array or section name. If the extent of the area is not provided explicitly and cannot be derived from the symbol table information in the executable, the interval of addresses consists of the start address only. The import of addresses into expressions is described in greater detail in section 5.28.4.
- The interval can be restricted to the start address in any case by writing `start` or `start address` instead of `address`.

- Similarly, the interval can be restricted to its last address by writing `last` or `last address`.
- The elements listed above (numbers, contents of registers, and addresses) may be combined by arithmetical operators such as +, −, and ⋆ as described in section 5.28.

Simple examples:

```
instruction 0x9110 is entered with D3 = 0, D7 = (0x10..0x1F);
instruction "prime" is entered with D2 = (-20 .. 20);
instruction "even"  is entered with D3 = D7;
```

The first specification says that the instruction at address `0x9110` is always entered with 0 in `D3` and $\texttt{0x10} \leq \texttt{D7} \leq \texttt{0x1F}$. (The values may be entirely different after executing the instruction.) The second specification declares that $-20 \leq \texttt{D2} \leq 20$ when `prime` is entered (for all calls of `prime`). The last specification declares that `D3` has the same value as `D7` when `even` is entered.

```
instruction "prime" is entered with @D7 = D7;
```

saves the value of `D7` at the beginning of `prime` in the user register `@D7` where this value remains unchanged even if the value of the processor register `D7` is modified by some assembly instructions.

The value of a register *R* at a program point *P* is always the value that was in effect before any value declaration for *R* at *P* was processed. Thus, the declaration

```
instruction "even" is entered with @X = 1, @Y = @X;
```

does not assign 1 to `@Y`, but the last value assigned to `@X` at some previous program point if any; if there was no previous assignment to `@X`, then `@Y` is undefined after the above declaration. Separating this declaration into two does not change the behavior;

```
instruction "even" is entered with @X = 1;
instruction "even" is entered with @Y = @X;
```

is equivalent to the combined declaration shown above.

Counting loop iterations: A user register can also be used to distinguish between the various iterations of a loop. Suppose routine `"prime"` contains a loop at relative address `0x100`. Then the following annotations establish a user register `@ctr` counting the loop iterations:

```
instruction "prime" is entered with @ctr = 0;
instruction "prime" + 1 loop is entered with @ctr = @ctr + 1;
```

The two annotations cause that `@ctr` has value 1 in the first iteration of the loop, value 2 in the second etc. – as long as the loop iterations have different contexts. If the number of loop contexts is restricted to, say, 3 by `max-unroll = 3` (see section 5.4), then the third, the fourth, and all following loop iterations share the same context, in which `@ctr` will be undefined since no consistent value is possible for it.

Addresses: As explained above, the specified values may also be addresses of arbitrary program points (see section 5.29). For instance,

```
D5 = start address "prime"
```

means the value of `D5` is the start address of `prime`,

```
D7 = address ("MC" + 1 computed + 8 bytes)
```

means the value of `D7` is 8 plus the address of the first computed call or branch in routine `MC`, and

```
D3 = address ("SU" + 4 bytes .. "SU" + 11 bytes)
```

means the value of `D3` is between 4 and 11 plus the address of routine `SU`.

Suppose `A` is an array whose start and end addresses are specified in the symbol table. Then `address "A"` describes the value range spanned by the start address and end address of `A`. If however the symbol table does not contain the end address of `A`, `address "A"` describes an exact value, namely the start address of `A`. The same happens in any case if you write `start address "A"`.

Now suppose `A` is an array whose extent is specified in the symbol table as `0x1000..0x1fff`. Then `address "A"` means the range `0x1000..0x1fff`. Yet

```
address ("A" + 8 bytes)
```

means the single number `0x1008`, calculated from the start address of `A`. If you wish to make this behavior explicit for documentation purposes, you may write

```
address (start "A" + 8 bytes)
```

yet the keyword `start` is redundant here and does not change the meaning.

If you wish to address relative to the end of `A`, you may write for instance

```
address (last "A" - 0xf bytes)
```

which means `0x1ff0`.

Analogously,

```
address ("A" + 0x10 bytes .. "A" + 0x1f bytes)
```

means the range `0x1010..0x101f` where the start address of `A` is taken in both occurrences of `"A" + ...`. If you wish to shift the entire extent of `A`, you may write

```
address (start "A" + 0x2000 bytes .. last "A" + 0x2000 bytes)
```

which means `0x3000..0x3fff`.

5.11 Assertions of Conditions

It is possible to assert that certain conditions, e.g., on values of registers or memory cells, hold at certain program points:

```
ASSERT  INSTRUCTION  ProgramPoint
        IS ENTERED WITH  Expr1, ..., Exprn;
```

As usual, the *ProgramPoint* can be an absolute address or an address relative to a routine entry, or something more complicated (see section 5.29). Symbolic loop expressions of the kind *R* + *n* `loops` are also possible. As suggested by the use of `ENTERED`, the expressions are evaluated when the *ProgramPoint* is reached, before the execution of the instruction at the *ProgramPoint*.

$Expr_1, \ldots, Expr_n$ are AIS expressions as described in section 5.28, in particular relational (section 5.28.11) or logical expressions (section 5.28.12). See section 5.28.5 for how to refer to memory cells in expressions.

Relational and logical expressions have three possible results: 0 meaning "false", 1 meaning "true", or the set $\{0,1\}$ (both 0 and 1) meaning "ambiguous". The assertion fails if one of the expressions $Expr_1, \ldots, Expr_n$ yields "false"; an ambiguous result (may be true or false) does not cause an assertion failure. In this regard, the `always` operator is useful, which transforms an ambiguous result into a false result (see section 5.28.12). `assert ...` *E* fails if *E* is definitely false, not if *E* is true or ambiguous, but `assert ... always (`*E*`)` fails if *E* is potentially false (false or ambiguous), not if *E* is definitely true.

Example: Consider the following assertions on register `D5`:
(1) `assert ... D5 == 5;`
(2) `assert ... always (D5 == 5);`
If `D5` indeed has value 5, both assertions hold. If `D5` has value 6, both fail. If a^3 only knows that the value of `D5` is in the set $\{5,6\}$, then the result of the comparison is ambiguous and therefore assertion (1) holds, but (2) fails.

It is important to note that the value information determined by the value analysis of a^3 includes all values possible in reality, but maybe more. In the example above, it is possible that in reality, `D5` is certainly 5, but a^3 only finds that the value is in the set $\{5,6\}$. In this case, failure of assertion (2) would be a false alarm. In contrast, assertion (1) only fails in case of definite errors, but it may falsely accept some error cases.

Since the `always` operator is part of the expression syntax, an `always` before $Expr_1$ does not affect the remaining expressions; it must be repeated if it is intended to affect all expressions.

The `always` operator has high operator precedence. Thus, `always D5 == 5` is parsed as `(always D5) == 5`, which is probably not the intended meaning. Therefore, the expression to which `always` is applied should be parenthesized as in `always (D5 == 5)`.

A failed assertion causes a message in the message window of the analysis and in the textual report file:

```
tricoredaan: Warning: In "/a3/prime.c", line 7:
        At address 0x10be in routine 'even':
        Assertion failed.
In "/a3/prime.ais", line 35:
        Annotation is specified here (index 1).
```

The first part refers to the program point to which the failed assertion belongs, while the second part refers to the text of the assertion in the AIS file, or more exactly to its beginning (the line with the keyword `assert`). The "index" is the index of the failed condition; it is 1 if $Expr_1$ failed, 2 if $Expr_2$ failed etc. In a message window, both parts are active links leading to the place they refer to when clicked.

5.12 Properties of Memory Areas

There are global properties of memory areas that constantly hold for the entire application, and local properties that only hold in a certain routine or at a certain instruction. Global properties can be specified in the form

```
AREA Area Properties;
```

where *Area* defines a memory area (see section 5.12.1) and *Properties* lists its properties (see sections 5.12.2–5.12.6).

Local properties relative to a routine are specified as

```
ROUTINE Point  AREA Area Properties;
```

where *Point* is a program point defining a routine, i.e. a routine name (quoted as usual) or the start address of a routine, and the rest is as above.

Similarly, local properties relative to an instruction are specified as

```
INSTRUCTION Point  AREA Area Properties;
```

where *Point* is an arbitrary program point resolving to an instruction address.

`ROUTINE "R"` refers to the entire extent of routine `R`, while `INSTRUCTION "R"` means the first instruction of routine `R`.

While the global properties can only be declarations, the local properties are mixed: some constitute declarations that are not scrutinized by a^3, while other ones describe assertions that are tested by a^3 and cause complaints if they are violated (cf. the discussion near the beginning of chapter 5). The explanation which is which is delivered when the various properties are introduced.

5.12.1 Area Definitions

The area definitions *Area* appearing after the keyword AREA may be one of the following:

- FROM *Position* TO *Position*
  Position .. *Position*
  are two equivalent ways to specify a memory area given by its start and end position (both belonging to the area);
- "*A*"
  denotes the memory area covered by array *A*;
- section "*S*"
  denotes the memory area covered by section "*S*" as defined in the executable. More precisely, it refers to the *first* memory area listed for section "*S*". In case of a relocated section, this usually is the area before relocation.
- relocated section "*S*"
  denotes the memory area covered by section "*S*" after relocation.

Here, a *Position* is the same as a *ProgramPoint*, which is defined in section 5.29. The most important cases are absolute addresses in memory and relative addresses of the form

"*N*" + *n* bytes

where "*N*" denotes the start address of array or routine *N*.

A common use case is that the area is specified with a start and an end position which are both expressed in terms of the same offset:

Position .. *Position* + *n* bytes

In this case the following short form is equivalent:

Position .. + *n* bytes

Note that this abbreviation cannot be used in combination with the from .. to form.

The description of an area by naming an array or a section only works if the executable contains information enabling **exec2crl** to derive the extent of the area. For an array for instance, the symbol table not only must contain the start address of the array, but also its end address or its length. The data-value specifications described in section 5.12.5 form an exception: they also work if the extent of the area is not known.

Properties fall into several classes: access, content, and copy properties. A special kind of content property is the specification of data values in memory (see section 5.12.5).

5.12.2 Access Properties

There are two possible *global* access properties: `is read-only` and `is read-write`. These global properties constitute declarations. The most useful global property is the `read-only` property. If an area is read-only then any data found there is known to be immutable and can be safely used by value analysis and for control-flow reconstruction (values of function pointers and switch tables).

Examples:
```
area 0x1000 .. 0x1fff is read-only;
area section "data" is read-write;
```

There are four possible *local* properties: `is read-only`, `is write-only`, `is read-write`, `is inaccessible`. In contrast to the global access properties, the local ones describe assertions, not declarations.

Examples:

`routine "prime" area 0x1000 .. 0x1fff is read-only;`
asserts that routine `prime` should not write to the given area. If it does, a^3 will complain.

`routine "swap" area "A" is write-only;`
asserts that routine `swap` should not read from array `A`; it may only write to it.

`instruction 0x1000 area section "data" is inaccessible;`
asserts that the instruction at address `0x1000` should not access section `"data"`.

In contrast, the access specifications described in section 5.9 define declarations, not assertions.

5.12.3 Volatile Memory Areas

The property `is volatile` declares a memory area as volatile. Such a declaration prevents a^3 from assuming any knowledge about values read from the area even if it knows the values that have been written to the area before. This annotation must be applied e.g., to memory areas to which I/O devices are mapped and to areas corresponding to C variables qualified by the keyword `volatile`. By itself the analysis cannot determine whether an area is volatile or not (not even for `volatile` C variables).

Examples:
```
area 0x1000 .. 0x1fff is volatile;
area section "input" is volatile;
```

Volatile declarations are only possible in global area declarations and cannot constitute assertions. There are no local volatile declarations.

a^3 does not know by itself that an area is volatile. Failure to declare volatile areas as volatile may lead to wrong results from timing and stack analysis.

5.12.4 Content Properties

There are four possible content properties: `contains code`, `contains data`, `might contain data`, and specifications providing information about data values in memory. Data-

value specifications are described in section 5.12.5. Here we concentrate on the `code` and `data` declarations. These properties exist only as global properties and can only constitute declarations.

Example: `area 0x1000 .. 0x1fff contains data`
Properties may be combined:
`area 0x1000 .. 0x1fff is read-only and contains data;`

The most important usage of this feature is to declare a data region within code. The executable reader **`exec2crl`** does not read unreachable code. Thus the code seen by **`exec2crl`** is not a connected region in general, but usually contains some gaps. Some of these gaps are known to contain data such as switch tables; these need no particular specification. The nature of other gaps is unknown to **`exec2crl`**. Without further annotations, **`exec2crl`** refuses to resolve program point descriptions such as `"`*R*`" + 1 computed` (meaning the first computed call or branch in routine *R*) across such gaps because the gaps might contain code, in particular computed calls or branches. However, if the gap is annotated as data area, **`exec2crl`** knows that it cannot contain code and skips over it when resolving program points.

Example: Assume routine `MC` has 2 computed branches requiring target annotations as presented in section 5.8.2:

```
instruction "MC" + 1 computed
branches to pc + 0x04 bytes, pc + 0x14 bytes, pc + 0x24 bytes;

instruction "MC" + 2 computed
branches to pc + 0x10 bytes, pc + 0x20 bytes;
```

Assume further the first branch of the second computed call ends prematurely with a return instruction whose last byte is at relative address `0x1B`. Thus, relative position `0x1C` is never reached and not decoded by **`exec2crl`**. This "gap" of 4 bytes should be declared as data to tell **`exec2crl`** that no reachable instruction is hiding there. The following annotation does the job:

```
area from "MC" + 2 computed + 0x1C bytes
     to   "MC" + 2 computed + 0x1F bytes contains data;
```

In some situations the code segment contains many such data blocks and it might be difficult to estimate their start addresses and sizes. By using the third form with `might contain data` it is possible to declare that a certain area might contain data blocks without knowing their exact locations (the given area might contain the gaps plus some code).

More precisely, when this annotation is supplied and **`exec2crl`** identifies that a certain area which lies within the given address range is unreachable (i.e. is a gap) it adds the assumption that this area contains data and jumps over it when resolving program point declarations. If in a later analysis step it can be determined that this assumption is wrong, i.e. **`exec2crl`** identifies that some instruction branches inside the area that has been assumed to contain data, a warning is issued; in this case the annotation needs to be reconsidered. Address ranges that have been declared to contain code via `contains code` are not affected by this annotation (this way, declarations of areas possibly containing data become more precise).

Example: The annotation

```
area "MC" .. + 100 bytes might contain data;
```

declares that there might be one or several data blocks within the first 100 bytes after the routine entry of `MC`.

The `might contain data` annotation should be used with greatest care and only as a last resort. If the given area does not contain data this might confuse **`exec2crl`** in resolving program points and lead to wrong results.

5.12.5 Data Values in Memory

The AIS language supports the specification of information about data values in memory. This information can be global or local to an instruction or a routine. Both kinds specify declarations, not assertions (cf. the discussion near the beginning of chapter 5). A data-value specification that is local to an instruction describes data that is valid *before* the instruction is executed; the data may be entirely different after executing the instruction. (Analogous specifications for register values are described in section 5.10.) A data-value specification that is local to a routine describes data that is valid before every instruction of the routine. In contrast, a global data-value specification describes data constant over the entire analyzed task. A global specification overwrites any values that may be contained in the executable. The values specified in a global specification are automatically marked as constant. This means that a^3 complains if they are overwritten by different values. (In contrast to areas marked as `read-only` as in section 5.12.2, writes that do not modify the specified values are not reported.) The specified values are *not* marked as data automatically; so if you want a^3 to use them as data, you should add an explicit `data` specification as described in section 5.12.4.

Global data-value specifications have two effects: First, they are used in the decoder **`exec2crl`** when it knows the address where the target address of a computed call or branch is stored, but this address is not filled with data in the executable so that the target address cannot be read from the executable. In this case, **`exec2crl`** checks whether there is a data-value specification specifying the data missing in the executable, and extracts the target address from that data. Second, the specified data values are used in value analysis: if the processor performs a load from the area for which data values have been specified, the value analyzer extracts the loaded value from the specified data values. Local data-value specifications are *only* used in value analysis, not in the decoder **`exec2crl`**. Hence, they are irrelevant for stack analysis.

Specifications about data values in memory look as follows:

```
AREA  Qualifier  Area   CONTAINS   DV1, ..., DVn;
INSTRUCTION  Point  AREA  Qualifier  Area   CONTAINS   ...;
ROUTINE   Routine   AREA  Qualifier  Area   CONTAINS   ...;
```

The first form is a global data-value specification, the second is local to the given instruction, and the third local to the given routine, which may be specified by its name or its address. In all three forms, an *`Area`* is specified as described in section 5.12.1. The additional *`Qualifier`* is optional. If it is omitted, the specification refers to the entire extent of the area, as far as it can

be derived from the symbol table of the executable. If the extent is known, a^3 checks whether the given data values exactly fit into the extent of the area. This checking is suppressed if the extent is not known, or if the *Qualifier* is `start`. In that case, the values are filled into memory starting from the start address of the area without any checking of overflow or underflow of the area. The *Qualifier* may also be `last`, meaning that the values are filled into memory beginning from the last address of the area (in the direction of increasing addresses i.e. leaving the area soon). This way, you may for instance specify a value appearing at the last address of an array. Of course, this only works if the last address of the area is known explicitly or can be derived from the symbol table information in the executable.

The list $DV_1, \ldots, DV_n$ provides information about the data values in the memory. This information may consist of exact values, or of intervals providing lower and upper bounds for the values. Note that the components $DV_1, \ldots, DV_n$ are not alternatives applying to the same memory cell, but information packages applying to n consecutive memory cells; the exact explanation follows below.

A *DV* may consist of a list of qualifiers, followed by an atomic *DV*. The following qualifiers exist:

- *Size qualifiers* are size units such as `byte` or `word`. They declare the size of the `DV`, e.g., `byte 1` describes a single byte containing the value 1 (seven 0-bits and one 1-bit), while `word 1` describes a word of 4 bytes containing the value 1 (31 0-bits and one 1-bit). If there are several size qualifiers in a row, the rightmost one wins. A complete list of all possible size qualifiers is given in section 5.30.4.
- *Byte-order qualifiers* describe how the bytes are ordered within larger memory units such as words. The following choices exist:
  - `little endian` or `low byte first` or `lbf`
    means the bytes are ordered so that the least significant byte comes first.
  - `big endian` or `high byte first` or `hbf`
    means the bytes are ordered so that the most significant byte comes first.
  - `sectionorder` is default and means that the endianness is read from the section information in the executable.
- The selectors `start` and `last` restrict an interval to an exact value, namely the first or the last value in the interval.

An atomic *DV* may be one of the following:

- A number written in hexadecimal or decimal, with an optional sign + or −. Small numbers are sign-extended to the size prescribed by the size qualifier. Numbers that are too large for the prescribed size are cut down by omitting the most significant bytes, e.g., `byte 0x1234` is the same as `byte 0x34`.
- An interval written in the form *A*`..`*B* or `from` *A* `to` *B*, where *A* and *B* are numbers as described above. This is a closed interval, i.e. the two end points *A* and *B* belong to the interval.

- A string, which stands for a sequence of bytes. With the size qualifier `byte`, the bytes of the string correspond to a contiguous sequence of memory bytes. With a larger qualifier, say `word`, each byte of the string describes its own word of memory (by zero extension).
- The keyword `address` followed by an area description stands for the interval of addresses in the area. An area description may be a single program point, a pair of program points `from` *`Start`* `to` *`End`*, or *`Start`* `..` *`End`*, or an array or section name. If the extent of the area is not provided explicitly and cannot be derived from the symbol table information in the executable, the interval of addresses consists of the start address only. (The interval can be restricted to the start address in any case by using `start`.) Size qualifiers for addresses usually can be omitted since a^3 knows about the size of addresses.

You should carefully distinguish between the string `"divides"` (a sequence of 7 bytes), and `address "divides"`, which denotes the start address of `divides` (4 bytes). You should also distinguish between the string `"12345"` and the number `12345`. With the size qualifier `word` for instance, the string describes 5 words of memory filled with the zero-extended characters of the string, while the number describes a single word of memory filled with that number.

In the `DV` list, common qualifiers including the keyword `address` may be factored out using parentheses. For instance,

```
word hbf (123, 0x422, address ("main", "even"), "bla")
```

is the same as

```
word hbf 123, word hbf 0x422, word hbf address "main",
word hbf address "even", word hbf "bla"
```

which describes 7 words of memory since the 3 characters of `"bla"` define 3 words. Since the rightmost (or innermost) size qualifier wins,

```
word hbf (123, 0x422, address ("main", "even"), byte "bla")
```

is the same as

```
word hbf 123, word hbf 0x422, word hbf address "main",
word hbf address "even", hbf byte "bla"
```

which describes 19 bytes of memory (4 words plus 3 extra bytes from `"bla"`). This sequence of data values could also be written with a different grouping as

```
hbf (word (123, 0x422, address ("main", "even")),
     byte "bla")
```

A data-value specification may describe an interval of possible data values. Suppose for instance `main` is a routine whose start address is specified in the executable, but not the end address, and `A` is an array whose start and end address are specified in the symbol table. Then

```
address ("main", "A")
```

provides information about two pieces of memory, each of the size of an address. The information about the first piece is exact: “this piece contains the start address of `main`”, while the information about the second piece is inexact: “this piece contains some address from the interval spanned by the start address and end address of `A`”. If however the symbol table does not contain the end address of `A`, the specification provides exact information about the second piece, too: “this piece contains the start address of `A`”. The same happens in any case if you write

```
start address ("main", "A")
```

5.12.6 Copied Areas

To specify the fact that some part of the executable is copied to a different memory area, you may write

```
area Target is copied from Source;
```

where *`Source`* is either an area *`SourceArea`* to be copied or the start point *`SourceStart`* of that area, and *`Target`* is either the area *`TargetArea`* where the copy is placed, or the start point *`TargetStart`* of that area.

Example: `area 0x140a is copied from section ".Cmd";`
copies the entire section `.Cmd` to the area starting at `0x140a`.

Remarks:

- The *`SourceArea`* must be part of a single section; it must not cross a section border.
- The copied area becomes a new section starting at the start address of the target area, which is either the start address of *`TargetArea`*, or *`TargetStart`* if only the start address of the target area is given.
- The length of this new section is the length of the target area if *`Target`* is given as an area *`TargetArea`* and the length of that area can be derived by a^3.
- If the length of the target area is unknown, in particular if *`Target`* is given as *`TargetStart`*, the length of the copied section is the length of the source area if *`Source`* is given as an area *`SourceArea`* and the length of that area can be derived by a^3.
- If the length of the source area is also unknown, in particular if *`Source`* is given as *`SourceStart`*, the copying proceeds from the start of the source area till the end of the section to which this start point belongs.
- A copy annotation cannot be used to overwrite a read-only section by another read-only section, or to overwrite a writable section by another writable section. However, overwriting a writable section by a read-only section is possible. Here, “read-only” and “writable” refer to the properties of the sections in the executables independent from any AIS annotations.

If *R* is the name of a routine in the copied section, then the identifier "*R*" still refers to the instance of *R* in the original section. To refer to the instance of *R* in the copied section, you may use the identifier "*R*(*T*)" where *T* is the start address of the copied section, i.e. the start address of the target area (not the start address of the copied routine!).

Example: `"SynCPU(0x140a)"`

Do not insert blanks in such an identifier. The number in parentheses must be written in hexadecimal form with header `0x` and lower case hex digits `a–f`. Leading zeroes or inserted underscores are not allowed.

5.13 Additional Execution Time *(Timing Analysis)*

The specification of additional execution times is relevant for timing analysis only.

WCET analysis determines an execution time for each basic block and execution context, taking into account possible cache and pipeline effects. This time may be artificially increased by specifying that certain instructions need additional execution time beyond the time determined by the analyzer. Such additional execution time may for instance be caused by DMA accesses.

The additional execution time has no influence on the pipeline states. In particular, it does not cause the pipeline to run empty.

Specifications of additional execution time refer to individual instructions:

```
INSTRUCTION ProgramPoint ADDITIONALLY TAKES time;
```

The *time* information can be an expression as described in section 5.28, which must be of type *cycles*. This can be achieved by using time units as described in section 5.28.16. The basic time unit is `cycles`, which means core processor cycles. If the clock rate of the processor has been specified before (section 5.2), the "real-time" units `sec` (seconds), `msec` (milliseconds), `usec` (microseconds), and `nsec` (nanoseconds) can be used, too. These units are internally converted into cycles using the specified clock rate. Thus, the expression used as *time* information ultimately yields some amount of processor cycles.

In general, expressions evaluate to a set of possible values, which are fractions (see section 5.28.2 for details). For the *time* expression in an `additionally takes` annotation, only the highest possible value (fraction) h matters. If $h \geq 0$, then $ceil(h)$ (h rounded up to the nearest integer amount of cycles) is taken as the additional execution time. If h is negative, then a^3 issues a warning and assumes an additional execution time of 0 cycles, i.e. the annotation is ignored.

If different additional-time annotations refer to the same program point, e.g.,

```
INSTRUCTION P ADDITIONALLY TAKES E1;
INSTRUCTION P ADDITIONALLY TAKES E2;
```

their additional times are added, i.e. the two annotations above are equivalent to

```
INSTRUCTION P ADDITIONALLY TAKES E1+E2;
```

Apart from the syntax described above, there are some other syntactic forms mainly to ensure backward compatibility:

- `...` `ADDITIONALLY TAKES EXACTLY` E
  is equivalent to
  `...` `ADDITIONALLY TAKES` E
- `...` `ADDITIONALLY TAKES MAX` E
  is equivalent to
  `...` `ADDITIONALLY TAKES` $(0..E)$
- `...` `ADDITIONALLY TAKES MIN` I `MAX` E
  is equivalent to
  `...` `ADDITIONALLY TAKES` $(I..E)$

where $(I..E)$ is an expression denoting the range from I to E (see section 5.28.6). The minimum additional execution time is irrelevant for WCET analysis since only the maximum is used, but the minimum may be specified for documentation purposes or future uses like in Best-Case Execution Time analysis.

Examples:

```
instruction 0x8124 additionally takes 5000 cycles;
instruction 0x3038 additionally takes
  @factor * 100 usec;
instruction prime additionally takes
  2500 + D7 * 4000 / 74 nsec;
instruction 0x4020 additionally takes
  switch (D5) {0 => 100, 1 => 300, 2 => 200} cycles;
```

The first annotation specifies a fixed additional time of 5000 cycles. The additional time in the second annotation depends on the value of the user register `@factor`. If it is 3, the time is 300 microseconds. If it is $(3..6)$, the time is 600 microseconds. If it is unknown or undefined, the time is 0, and a warning is issued. The third annotation is similar, but the time is proportional to the value of processor register `D7` and a constant amount of time is added. The last annotation uses a `switch` expression as described in section 5.28.13. Its additional time is 100 cycles if `D5` is 0, 200 cycles if it is 2, and 300 cycles if it is 1. If `D5` is 0..1, the switch expression yields a set of values containing 100 and 300, but 300 is the maximum of this set so that the additional time is still 300. The same happens if `D5` is 1..2 or 0..3, indeed if it is any set containing 1, or if it is unknown altogether. If `D5` is 4 or 7 or 6..12, the `switch` expression yields an undefined value, which causes a warning and lets a^3 assume an additional execution time of 0 cycles:

```
Unable to evaluate given expression to a concrete value
  for additional cycles.  Please review your annotations.
  Assuming 0 cycles.
```

5.13.1 Note on Backward Compatibility

Before support for general AIS expressions as described in section 5.28 was added to additional-time annotations, they already supported a limited amount of expressions with a special syntax that could not be unified with the more powerful general expressions. These special expressions now lead to syntax errors or warnings and should be replaced by equivalent modern expressions as described in the following.

- Annotations with constants still work as before.
- The only exception is something like
  `MIN 100 MAX 200 nsec`
  which causes the message
  `Annotation expects expression of type 'cycles', but found type 'number'.`
  The unit must be added to both parts:
  `MIN 100 nsec MAX 200 nsec`
- A product like
  R `* EXACTLY 1000 cycles`
  where *R* is a register now gives a syntax error:
  `syntax error, unexpected exactly`
  The word `EXACTLY` should be put before the register or deleted altogether:
  R `* 1000 cycles`
- A product such as
  R `* MAX 1000 cycles`
  where *R* is a register now works in principle, but gives the warning
  `Deprecated use of 1 argument for function 'max', you should use at least 2.`
  The reason for this warning is that `MAX` is now considered as the `MAX` operator described in section 5.28.6, which is applied to the single argument 1000. To avoid the warning, you should delete `MAX` or place it before the register:
  `MAX` *R* `* 1000 cycles`
- A selection had the form
  `(`$time_0$`, `$time_1$`, ...) [`*register*`]`
  This was a parenthesized comma-separated list of constants followed by a register name in square brackets. The idea of selection was that the register value is used as an index in the list (starting from 0), i.e. the meaning of the selection was $time_i$ if i is the value of the register. Since all parts $time_i$ had to be started by `MAX` or `EXACTLY`, one of the messages shown above will appear. In addition there may be
  `syntax error, unexpected ','`
  The solution is to use a `switch` expression as described in section 5.28.13. For instance,
  `(max 100 nsec, max 300 cycles, max 200 usec) [D5]`
  should be replaced by

```
switch D5 {0 => 100 nsec, 1 => 300 cycles, 2 => 200 usec};
```

A common unit can be put at outside the switch, e.g.,
`(max 100 cycles, max 300 cycles, max 200 cycles) [D5]`
may be replaced by
`switch D5 {0 => 100, 1 => 300, 2 => 200} cycles;`

5.14 WCET Contribution of Code Snippets *(Timing Analysis)*

After a timing analysis, a^3 reports the WCET contributions of the various routines in the displayed graphs and in the report files (see section 7.4.2). The WCET contributions of code snippets that are not routines are included in the report files upon special request. Such requests may take several forms. One of these forms is the following:

```
SNIPPET P1 ENDING P2 IS EVALUATED AS "Name";
```

Here, P_1 and P_2 are two program points indicating the beginning and the end of the considered snippet, and *Name* is a name given to the snippet for identifying it in the report files. In the textual report file, the WCET contribution of the snippet appears under the heading

```
* WCET contributions of region 'Name' (routine 'R'):
```

where *R* is the routine containing the beginning of the snippet. The names must be unique among the evaluated snippets; if there are two snippets with the same name, only one of them appears in the report file.

A snippet to be evaluated cannot contain fragments of basic blocks. Therefore, the actual beginning of the evaluated snippet is at the beginning of the basic block B_1 containing P_1, and the actual end is at the end of the basic block B_2 containing P_2. The snippet also includes all routines called within the snippet, including possible routines called at the end of B_2. The snippet must be single-entry / single-exit, i.e. B_1 must be the only entry into the snippet and B_2 the only exit. (Yet B_1 may have several predecessors and B_2 several successors.) Violation of this condition is reported in the report files. Snippet start and end should be in the same routine.

There is a variant

```
SNIPPET P1 CONTINUING P2 IS EVALUATED AS "Name";
```

which excludes the block B_2 containing P_2 from the evaluated snippet; the snippet ends immediately before the beginning of B_2. If P_2' is the address of the predecessor B_2' of B_2, then `snippet` P_1 `continuing` P_2 includes B_2' and the edge from B_2' to B_2. In contrast, `snippet` P_1 `ending` P_2' includes all edges leaving B_2' (there may be others than just the edge to B_2).

The third form

```
SNIPPET P1 IS EVALUATED AS "Name";
```

has no overt end point for the snippet. In this case, the snippet extends from the beginning of the block B_1 containing P_1 till the end of the routine containing P_1. In the special case that P_1 is a quoted routine name `"`*R*`"`, this means that the evaluated snippet is the entire routine *R* (but the WCET contribution of *R* is listed in the report files anyway).

The WCET contribution of a snippet *S* is the contribution of *S* to the overall WCET, not the WCET of one execution of *S*. The WCET contribution is 0 if *S* is not on the calculated WCET path. If *S* is in a loop that is executed 100 times in the worst case, then the WCET contribution of *S* is the WCET of all 100 executions together, not that of a single execution.

5.14.1 Note on Backward Compatibility

The syntax described above replaces an older syntax that is no longer supported. Annotations in this older syntax should be rewritten as described in the following:

- `SNIPPET` P_1 `..` P_2 `IS EVALUATED AS "`*Name*`";`
  should be replaced by
  `SNIPPET` P_1 `ENDING` P_2 `IS EVALUATED AS "`*Name*`";`
- `SNIPPET` P_1 `..` P_2 `IS EVALUATED AS "`*Name*`" EXCLUDING LAST;`
  should be replaced by
  `SNIPPET` P_1 `CONTINUING` P_2 `IS EVALUATED AS "`*Name*`";`
- `SNIPPET` P_1 `IS EVALUATED AS "`*Name*`";`
  should be replaced by
  `SNIPPET` P_1 `ENDING` P_1 `IS EVALUATED AS "`*Name*`";`

  The form
  `SNIPPET` P_1 `IS EVALUATED AS "`*Name*`";`
  is valid in both syntax schemes, but with different meanings: In the old version, it meant the snippet consisting of the block B_1 containing P_1 and all routines called by it, while in the new version, it means the snippet from the beginning of B_1 till the end of the routine containing B_1. To retain the old meaning, P_1 must be replaced by P_1 `ending` P_1 as described above.

5.15 Not-Analyzed and External Code Snippets

The examples in this section are generic, i.e. not specific to the TriCore architecture.

Snippets are pieces of code, such as routines or basic blocks, or any instructions between two given addresses A_1 and A_2. The AIS format allows for specifying that certain snippets should not be decoded and thus not analyzed for WCET and stack usage. The execution time and stack usage of such snippets must be specified explicitly. Section 5.15.1 describes how to exclude entire routines from decoding and analysis. Thereafter, section 5.15.2 describes the exclusion of code pieces that are not routines. The subsequent sections 5.15.3 and 5.15.4 explain how to specify the execution

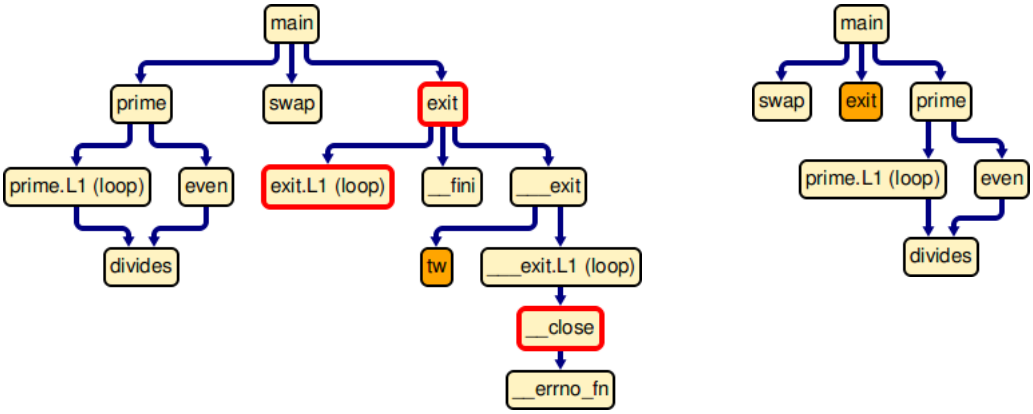


Figure 5.2: Effect of declaring `exit` not-analyzed

time and stack usage of excluded snippets. Of course, 5.15.3 is only relevant for timing analysis and 5.15.4 is only relevant for stack analysis. Section 5.15.5 explains how to declare effects of excluded snippets, which are important for both stack and timing analysis. Finally, section 5.15.7 contains a summary.

5.15.1 Declaration of Not-Analyzed and External Routines

To avoid the decoding of routines you are not interested in, like library code, you may write:

```
SNIPPET "name" IS EXTERNAL     AND ...;
SNIPPET "name" IS NOT ANALYZED AND ...;
```

Here, *name* is the name of the routine the specification applies to.

Routine names must be surrounded by double quotes (like all names).

Both forms prevent routine *name* from being analyzed and inform a^3 about its execution time, stack usage, and effects. This information is contained in the part abbreviated to "..." and will be explained in sections 5.15.3 (time), 5.15.4 (stack), and 5.15.5 (effects). Use the form with `EXTERNAL` if you want to analyze an incomplete executable that does not contain routine *name*, and the form with `NOT ANALYZED` if your executable contains *name*, but a^3 should not analyze it.

When **`exec2crl`** encounters a call to a routine declared not-analyzed, it does not follow the call, and thus does not read and decode the routine declared not-analyzed, and any routines only reachable through it. For the effect, compare the left graph in Figure 5.2 with the right graph where `exit` has been declared not-analyzed.

5.15.2 Not-Analyzed Code Snippets That Are Not Routines

In addition to exclusion of entire routines from analysis, it is also possible to exclude fragments of routines with arbitrary start and end points. There are two possible formats for such specifications:

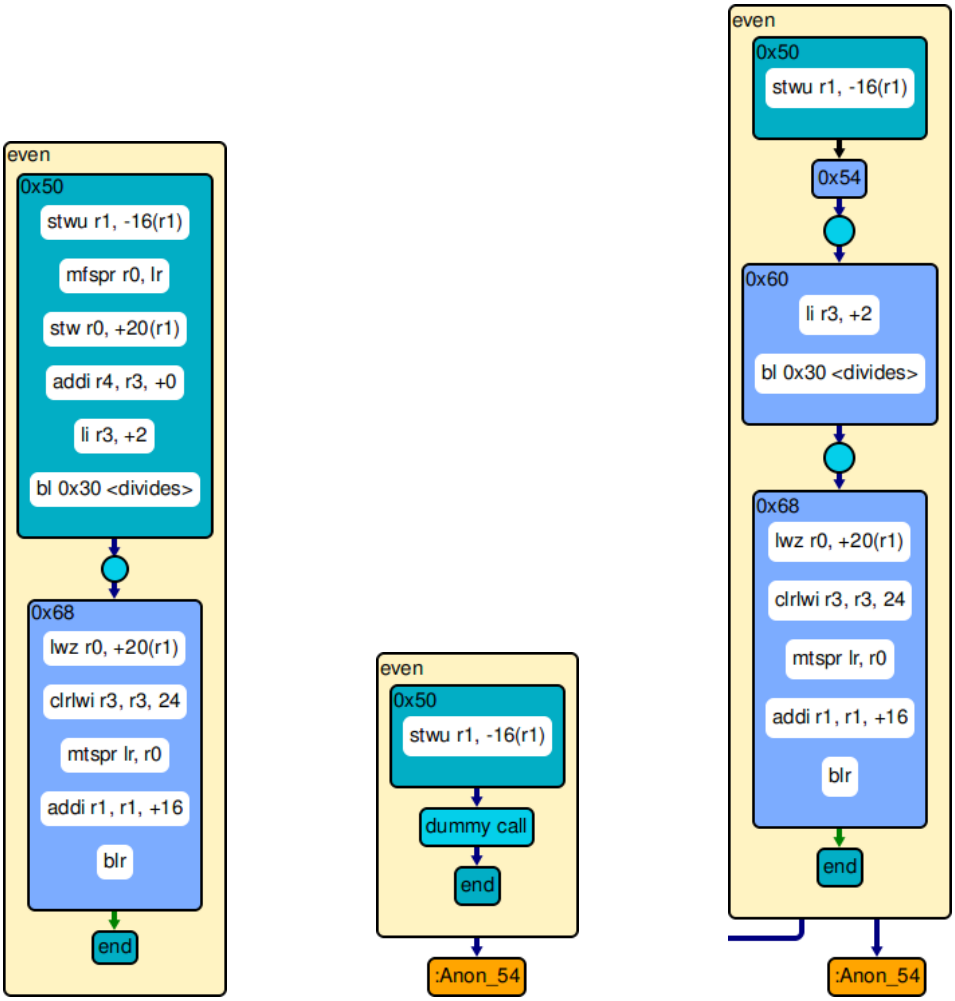


Figure 5.3: Effects of excluding a snippet from analysis

```
SNIPPET P1 .. P2 IS NOT ANALYZED AND ...;
SNIPPET P1 .. P2 CONTINUING L IS NOT ANALYZED AND ...;
```

Here, P_1 and P_2 are two program points indicating the beginning and end of the excluded snippet. The program point indicating the end should resolve to the last address within the snippet, not the first address afterward. Since the snippet is not decoded, a^3 cannot find out by itself the *continuing program points* to which control flows when the snippet is left. In the first form, this is the exit node of the routine the snippet belongs to, while in the second form, the continuing program points are specified explicitly in the form of a comma-separated list L. If the excluded snippet is a linear piece of code and its last instruction is not control-relevant, there is only one continuing program point, namely the first address after the snippet. If the last instruction is a conditional branch, the continuing program points are the successor instruction and the branch targets. The part abbreviated to "..." should contain information about execution time (5.15.3), stack usage (5.15.4), and effects (5.15.5) of the excluded snippet.

Example: Figure 5.3 (left) shows the basic block graph of a routine called `even`. It consists of two basic blocks, the first starting at address `0x50` with 6 instructions, and the second starting

at address `0x68` with 5 instructions. The box labeled by `end` signifies the routine exit. The five instructions of the first block have addresses `0x50`, `0x54`, `0x58`, `0x5c`, and `0x60`, respectively. Figure 5.3 (middle) shows the graph resulting from the specification

```
snippet 0x54..0x5f is not analyzed and ...;
```

which excludes the second, third, and fourth instruction from the analysis. The excluded instructions are represented by the `dummy call` block and a new routine, which usually is an anonymous routine named `:Anon_54` after the start address of the excluded snippet. If however a name is associated with this code address in the symbol table, that name is taken as the name of the new routine. Since no continuing program point has been specified, a^3 assumes that routine `even` is left after leaving the non-analyzed snippet. Therefore, the instructions starting from `0x60` are not reached and thus not contained in the graph.

Figure 5.3 (right) shows the graph resulting from the specification

```
snippet 0x54..0x5f continuing 0x60
        is not analyzed and ...;
```

which again excludes the second, third, and fourth instruction from the analysis. The excluded instructions are now represented by the block labeled `0x54` after the start address of the excluded snippet, which does not contain any instructions, and by a new routine called by this block, which is named as in the example above. As specified, control flows to the address `0x60` after leaving the excluded snippet. Therefore, the instructions starting from `0x60` are now contained in the graph.

As you can see from these examples, you must specify the continuing program point even if it is the end address of the snippet plus 1.

The excluded snippet is not restricted to a fragment of a single basic block. If it contains branches leaving the snippet, you should add the targets of these branches to the list of continuing program points. Yet note that there is only one timing, stack usage, and stack effect for each excluded snippet; you cannot make this information depend on the continuing program points.

Branches to addresses within the excluded snippet other than the start address should be avoided. They lead to the implicit creation of additional excluded snippets starting at the various branch targets. These snippets have no timing and stack usage specification and thus their runtime is assumed to be 0, which may lead to wrong WCET results, while stack analysis yields an unknown stack usage.

5.15.3 Execution Time of Excluded Snippets *(Timing Analysis)*

Since routines declared not-analyzed are not decoded, a^3 cannot determine their worst-case execution time. The same holds true for external routines, just because their code is not available. Therefore, timing information for external and not-analyzed routines must be provided as part of the snippet specification:

```
SNIPPET "name" IS EXTERNAL     AND TAKES time;
SNIPPET "name" IS NOT ANALYZED AND TAKES time;
```

The `time` information can be an expression as described in section 5.28, which must be of type *cycles*. This can be achieved by using time units as described in section 5.28.16. The basic time unit is `cycles`, which means core processor cycles. If the clock rate of the processor has been specified before (section 5.2), the "real-time" units `sec` (seconds), `msec` (milliseconds), `usec` (microseconds), and `nsec` (nanoseconds) can be used, too. These units are internally converted into cycles using the specified clock rate. Thus, the expression used as `time` information ultimately yields some amount of processor cycles.

In general, expressions evaluate to a set of possible values, which are fractions (see section 5.28.2 for details). For the `time` expression in a `takes` annotation, only the highest possible value (fraction) h matters. If h is finite and $h \geq 0$, then $ceil(h)$ (h rounded up to the nearest integer amount of cycles) is taken as the additional execution time. If h is negative or infinite, then a^3 issues a warning and assumes an execution time of 0 cycles.

Apart from the syntax described above, there are some other syntactic forms mainly to ensure backward compatibility:

- `TAKES EXACTLY` E is equivalent to `TAKES` E
- `TAKES MAX` E is equivalent to `TAKES` $(0..E)$
- `TAKES MIN` I `MAX` E is equivalent to `TAKES` $(I..E)$

where $(I..E)$ is an expression denoting the range from I to E (see section 5.28.6). The minimum execution time is irrelevant for WCET analysis since only the maximum is used, but the minimum may be specified for documentation purposes or future uses like in Best-Case Execution Time analysis.

Note on Backward Compatibility: Previously only constants were allowed, not the general AIS expressions as described in section 5.28. Since constants are special expressions, the old annotations still work. The only exception is something like

```
MIN 100 MAX 200 nsec
```

which causes the message

```
Annotation expects expression of type 'cycles',
but found type 'number'.
```

The unit must now be added to both parts:

```
MIN 100 nsec  MAX 200 nsec
```

Example with constants (Exclusion of entire routines):
To exclude a routine `"even"` with a fixed execution time of 150 cycles from analysis, you may write:

```
snippet "even" is not analyzed
        and takes 150 cycles;
```

or equivalently with real time

```
clock exactly 40 MHz;
snippet "even" is not analyzed
        and takes 3750 nsec;
```

If the WCET contribution of `even` to the entire task is 150 cycles and `even` is declared not-analyzed with WCET 150 cycles as in the example above, the overall WCET may increase considerably. The additional cycles are partly caused by cache and pipeline effects; the analyzer flushes the pipeline at the call of the non-analyzed routine and refills it after returning from the call. An additional penalty comes from the fact that the not-analyzed routine may modify the registers in any way so that the register settings are unknown after returning from it, which causes some imprecision in the remainder of the analysis. This WCET penalty can be avoided by specifying that the not-analyzed routine does not violate the calling conventions (see section 5.15.5):

```
snippet "even" is not analyzed
        and takes 150 cycles
        and does not violate calling conventions;
```

Examples involving expressions:
In the standard example of this manual, routine `divides` has two call sites, one in routine `even` and one inside a loop contained in routine `prime`. Suppose we want to exclude `divides` from analysis with an execution time of 50 cycles when called from `even` and 100 cycles when called from the loop in `prime`. This can be done by assigning these runtimes to a user register at appropriate places before the call sites of `divides` as described in section 5.10 and then using the user register in the runtime specification:

```
instruction "even" is entered with @dividesTime = 50;
instruction "prime" + 1 loop is entered with @dividesTime = 100;
snippet "divides" is not analyzed and takes @dividesTime cycles
        and does not violate calling conventions;
```

Next suppose the runtime of `divides` is still 50 cycles when called from `even`, but the runtime depends on the iteration number when it is called from the loop: in the first iteration, it needs 100 cycles, in the second 200 cycles, etc. This can be done by introducing a counter that is initialized to 0 and incremented by 1 whenever the loop is entered. The call from `even` is now characterized by a counter value of 0.

```
instruction "prime" is entered with @CTR = 0;
instruction "prime" + 1 loop is entered with @CTR = @CTR + 1;
snippet "divides" is not analyzed
        and takes (@CTR == 0? 50: @CTR * 100) cycles
        and does not violate calling conventions;
```

The above works well as long as every iteration has its own context. If the number of loop contexts is restricted to, say, 3 by `max-unroll = 3` (see section 5.4), then the third, the fourth, and

all following loop iterations share the same context, in which `@CTR` will be undefined since no consistent value is possible for it. Undefined user registers are evaluated as $(-inf..inf)$, i.e. are assumed to have all possible values. This causes the entire expression to be evaluated to $(-inf..inf)$ in the shared context. This does not mean that a^3 assumes an infinite execution time for `divides` in this context, but causes it to assume a time of 0 cycles.

If the expression specifying the execution time of a snippet excluded from analysis is unbounded because some part is undefined, then a^3 assumes a runtime of 0 cycles for the excluded snippet, which is likely to cause an underestimation of the overall WCET.

The case of an undefined/unbounded expression can be recognized by the following warning:

```
Unable to evaluate given expression to a concrete value
  for additional cycles.  Please review your annotations.
  Assuming 0 cycles.
```

There is a second warning

```
The user given wcet for EXCLUDED routine divides is 0 cycles.
```

which appears in addition to the first in case of an undefined/unbounded expression, but also on its own if the expression is defined/bounded, but evaluates to a number ≤ 0.

The problem with undefined expressions can be avoided by means of the `try` operator of section 5.28.6. The expression `try (A,B)` evaluates to A if this is bounded, and to B otherwise. If the maximum iteration number of the loop in `prime` is 10, then a safe way to specify the execution time of `divides` is

```
snippet "divides" is not analyzed
        and takes (@CTR == 0? 50: try (@CTR, 10) * 100) cycles
        and does not violate calling conventions;
```

Example (Exclusion of arbitrary snippets):
Figure 5.4 (left) shows the basic block graph of a routine called `even`, with results of timing analysis. It consists of two basic blocks, the first starting at address `0x50` with 6 instructions, and the second starting at address `0x68` with 5 instructions. The box labeled by `end` signifies the routine exit. The numbers labeled `max t` indicate the number of cycles the preceding basic block takes to execute. The five instructions of the first block have addresses `0x50`, `0x54`, `0x58`, `0x5c`, and `0x60`, respectively. Figure 5.4 (middle) shows the graph resulting from the specification

```
snippet 0x54 .. 0x5f is not analyzed
        and takes 3 cycles;
```

which excludes the second, third, and fourth instruction from the analysis. The excluded instructions are represented by the `dummy call` block and a new routine, which usually is an anonymous routine named `:Anon_54` after the start address of the excluded snippet. If however a name is

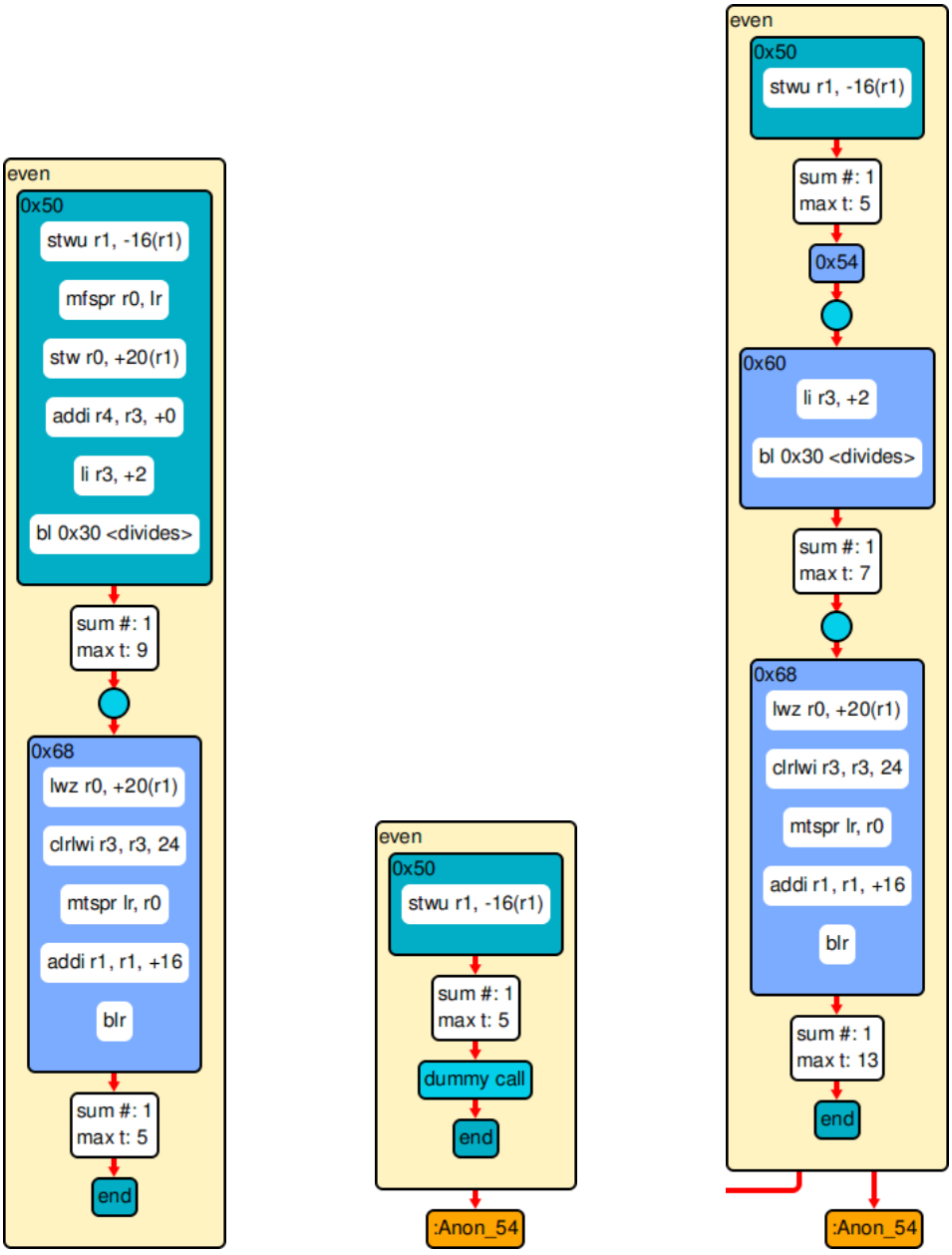


Figure 5.4: Effects of excluding a snippet from analysis

associated with this code address in the symbol table, that name is taken as the name of the new routine. Since no continuing program point has been specified, a^3 assumes that routine `even` is left after leaving the non-analyzed snippet. Therefore, the instructions starting from `0x60` are not reached and thus not contained in the graph.

Figure 5.4 (right) shows the graph resulting from the specification

```
snippet 0x54 .. 0x5f continuing 0x60
        is not analyzed and takes 3 cycles;
```

which again excludes the second, third, and fourth instruction from the analysis. The excluded instructions are now represented by the block labeled `0x54` after the start address of the excluded snippet, which does not contain any instructions, and by a new routine called by this block, which is named as in the example above. As specified, control flows to the address `0x60` after leaving the excluded snippet. Therefore, the instructions starting from `0x60` are now contained in the graph.

As you can see from these examples, you must specify the continuing program point even if it is the end address of the snippet plus 1.

The blocks indicating the place of the exclusion in the graph pictures (e.g., `dummy call` in Figure 5.4, middle and `0x54` in Figure 5.4, right) do not have a WCET box. The time specified for the excluded snippet (3 in this example) is attributed to the anonymous routine formed from the excluded snippet (`:Anon_54` in this example). The WCET contribution of this routine as stored in its second info field is a *multiple* of the WCET of the excluded snippet. The multiplication factor is the number of times the excluded snippet is executed in the worst case. It is 0 if the snippet does not lie on the worst-case path.

An exclusion may increase the WCET considerably since the excluded snippet may modify all registers and so a^3 has lost all information about register contents after the excluded code snippet. Claiming that the excluded snippet does not violate the calling conventions is not possible since the excluded snippet is not a routine and therefore the notion of calling conventions is meaningless. While it is impossible to declare for a general register that its value is not modified by the excluded snippet, this can be done for the stack pointers using the stack effect assertions presented in section 5.15.5, e.g.,

```
snippet 0x54 .. 0x5f continuing 0x60
        is not analyzed and takes 3 cycles
        and removes exactly 0 bytes of user stack
        and removes exactly 0 bytes of system stack;
```

5.15.4 Stack Usage of Excluded Snippets *(Stack Analysis)*

Since snippets declared not-analyzed are not decoded, a^3 cannot determine their stack behavior. The same holds true for external snippets, just because their code is not available. Therefore, the stack behavior of external and not-analyzed snippets must be provided as part of the snippet specification (abbreviated to "..." in sections 5.15.1 and 5.15.2).

The stack behavior falls into two parts: the stack usage, i.e. the maximum amount of stack space allocated within the excluded snippet, and the stack effect, i.e. the difference of the stack level after leaving the excluded snippet from the level before entering the snippet.

Example: Assume that the excluded snippet performs the following stack operations:

Allocate 4 bytes	leading to stack level 4;
Allocate 8 bytes	leading to stack level 12;
Free 4 bytes	leading to stack level 8;

Allocate 8 bytes	leading to stack level 16;
Free 8 bytes	leading to stack level 8;
Free 8 bytes	leading to stack level 0.

The given stack levels are relative to the level at the beginning of the snippet. Thus the stack usage is 16 (the maximum relative stack level) and the stack effect is 0 (the relative stack level at the end). Without the last stack operation, the stack effect would be $+8$, and with an additional operation "free 4 bytes", the stack effect would be -4.

In fact, stack analysis for TriCore processors computes results for *two* stacks called user stack and system stack. The user stack is the "normal" stack with stack pointer `A10`. The system stack is the Context Save Area CSA whose "stack pointer" is the `FCX` register. Hence system stack space is always allocated in portions of size 64 byte, which is the size of a context.

The stack usage (maximum relative level of the two stacks) is specified as follows:

```
SNIPPET ... AND USES EXACTLY Nu BYTES OF USER   STACK
            AND USES EXACTLY Ns BYTES OF SYSTEM STACK
```

The two numbers N_u and N_s must be non-negative integer constants (no expressions). Since N_s indicates the maximum CSA size used up by the excluded snippet *measured in bytes*, it should be a multiple of 64.

The specification of the stack effect is explained in the next section 5.15.5 since it is of relevance not only for stack analysis, but also for timing analysis.

5.15.5 Effects of Excluded Snippets *(Both Analyses)*

The notion of effect summarizes any modification of register values and values in memory. By default, a^3 assumes that an excluded snippet modifies all such values in an unknown way and thus destroys all value information collected by a^3. Therefore, it is important for both timing and stack analysis to specify that certain values remain unchanged or are changed in some specific way.

The simplest possibility is that the excluded snippet behaves like an ordinary routine, i.e. respects all the calling conventions of the compiler. This can be specified as follows:

```
... AND DOES NOT VIOLATE CALLING CONVENTIONS
```

This specification means that the callee-saved registers keep their values, and implies a specific stack effect: the user stack level remains the same, and the system stack level decreases by 64 bytes – the size of the upper context removed from the CSA area by the return instruction of an ordinary routine. (Stack effects have been introduced in the previous subsection 5.15.4.)

If the excluded snippet violates the calling conventions, the stack effect must be specified explicitly. The specification of the stack effect is similar to the specification of the stack usage as described in section 5.15.4, but the sign of the effect must be taken into account. If the effect is positive, i.e. the excluded snippet allocates some stack space that is not freed within the snippet, the specification looks as follows:

```
... AND LEAVES BEHIND EXACTLY N BYTES OF kind STACK
```

where *kind* is either `user` or `system`. If the effect is negative, i.e. the excluded snippet frees some stack space that has not been allocated by the snippet, the specification looks as follows:

```
... AND REMOVES EXACTLY N BYTES OF kind STACK
```

In both cases, the number *N* must be a non-negative integer constant (no expression). Of course, two such stack effect specifications are needed, one for the user stack and one for the system stack.

The specification `does not violate calling conventions` for a routine is therefore similar to the stack effect declaration

```
AND REMOVES EXACTLY  0 BYTES OF USER STACK
AND REMOVES EXACTLY 64 BYTES OF SYSTEM STACK
```

but includes the declaration that the callee-saved registers apart from the stack pointers are not modified.

It is possible – and sometimes useful – to combine an annotation for an explicit stack effect with a specification `does not violate calling conventions`. This means that the stack pointers are modified as specified explicitly, while all other registers are handled according to the calling conventions.

5.15.6 Cache and Pipeline Flushes *(Timing Analysis)*

Since an excluded snippet may modify the cache in unpredictable ways, a^3 normally assumes that the cache contents are completely unknown at the first instruction after the excluded snippet. This pessimistic cache flush may be suppressed by the AIS directive

```
global disable_external_cache_flush = 1;
```

When the pipeline analysis of a^3 encounters an excluded snippet, it flushes the pipeline, i.e. lets it run empty, and then refills it at the first instruction after the excluded snippet. The pipeline flush can be suppressed by the AIS directive

```
global disable_external_pipeline_flush = 1;
```

It causes the pipeline analysis to let the pipeline evolve as if the last instruction before the excluded snippet would be immediately followed by the first instruction after it.

As indicated by the word `global`, the directives for disabling cache and pipeline flushes are stand-alone directives that cannot be attached to specific exclusion annotations by means of `and`. They do not refer to individual excluded snippets, but to all of them without distinction.

If cache or pipeline flushes at excluded snippets are disabled, the WCET prediction of a^3 may be *wrong*, i.e. it may be smaller than the actual worst-case execution time.

5.15.7 Summary

Excluded snippets and their properties are specified as follows:

- What is excluded?
  (See 5.15.1 for entire routines and 5.15.2 for other code snippets)
- What is its execution time?
  (Only relevant for timing analysis – see 5.15.3)
- What is its stack usage?
  (Only relevant for stack analysis – see 5.15.4)
- What is its effect on register values (including the stack effect)?
  (Relevant for *both* analyses – see 5.15.5)
- Influence of excluded snippets on cache and pipeline analysis
  (Only relevant for timing analysis – see 5.15.6)

Here are some examples of complete specifications:

```
global disable_external_pipeline_flush = 1;

snippet "error" is not analyzed
  and takes 735 cycles
  and uses exactly  16 bytes of user stack
  and uses exactly 128 bytes of system stack
  and does not violate calling conventions;

snippet 0x8100..0x81ff continuing 0x82ff is not analyzed
  and takes 84 cycles
  and uses exactly 14 bytes of user stack
  and uses exactly  0 bytes of system stack
  and leaves behind exactly 6 bytes of user stack
  and removes exactly 64 bytes of system stack;
```

5.16 Calling Conventions

As explained in section 5.15.5, a^3 assumes by default that a not-analyzed routine violates the calling conventions of the compiler. In contrast, a^3 assumes by default that each analyzed routine satisfies the calling conventions of the compiler. If this is false for some routine, this fact should be made known to a^3 as follows:

```
ROUTINE "R" VIOLATES CALLING CONVENTIONS AND ...;
```

where *R* is the name or the start address of the violating routine. The part abbreviated by "..." should contain a specification of the stack effect of the routine using the same syntax as introduced in section 5.15.5. This specification is relevant not only for stack analysis, but also for timing analysis since the layout of the stacks is important for tracking the values in the registers and the processor memory.

5.17 Infeasible Code

Infeasible code is code that is never executed. You can *declare* that a certain block is never executed unconditionally (section 5.17.1) or conditionally (section 5.17.2), or *assert* that it should be executed or should not be executed (section 5.17.3). In the first case, a^3 accepts the declaration and does not produce analysis results for that block, while in the last case, it checks whether the assertion is satisfied and complains if not (cf. the discussion near the beginning of chapter 5).

5.17.1 Declaration of Unconditional Infeasibility

To *declare* that a certain block is never executed, you may write

```
SNIPPET ProgramPoint IS NEVER EXECUTED;
```

where *ProgramPoint* refers to an arbitrary instruction in the block, e.g., by its address.

Example: `snippet 0x1a0c is never executed;`

The *ProgramPoint* may also be specified relative to a routine entry.

Example: `snippet "prime" + 0x98 bytes is never executed;`

You can also specify that an entire routine is never executed by using its name as *ProgramPoint*:

Example: `snippet "prime" is never executed;`

This also works for loop routines:

Example: `snippet "prime" + 1 loop is never executed;`

Infeasible code is still decoded and thus appears in the disassembly window and in the control-flow graph pictures, but does not contribute to the calculated analysis results. In the graph pictures, infeasible routines are shown in gray. Infeasible basic blocks are marked by the word `infeasible` in WCET result graphs and by the symbol $\perp$ in stack result graphs.

Infeasibility is propagated through the control-flow graph: if a block is only reachable from an infeasible block, it is infeasible, too; there is no need for an explicit specification.

If all paths from the entry to the exit of the analyzed program part contain a block declared infeasible, timing analysis deduces that "This problem is infeasible".

5.17.2 Declaration of Conditional Infeasibility

There is also a conditional version of infeasibility that can be *declared* as follows:

```
SNIPPET ProgramPoint IS NOT EXECUTED IF Expr;
```

where *Expr* is an AIS expression as described in section 5.28. The expression is evaluated to a numerical value. Ideally, a value of 0 is considered as "false" and any other value as "true" as in the C programming language. Yet an expression evaluates to a single number in special cases only while in general, it results in a set of numbers. Such a set is considered as "true" if it does not contain 0.

Conditional infeasibility can be used to improve precision if a code snippet is never executed in certain contexts, but possibly executed in others. Consider the following example:

```
if (C) { A; R(); }
else   { B; R(); }
```

where routine `R` contains a code snippet `S` that is never executed if `R` is called after `A`, but may be executed if `R` is called after `B`. These facts may be expressed in AIS as follows:

```
instruction A is entered with @notS = 1;
instruction B is entered with @notS = 0;
snippet S is not executed  if @notS;
```

The first two annotations define a user register called `@notS` (see section 5.10). It is given value `1` (true) at the beginning of `A` and value `0` (false) at the beginning of `B`. These values are propagated through the calls of `R`, and the third annotation specifies that `S` is not executed if `@notS` is `1`, i.e. in the context of the call after `A`.

5.17.3 Assertions about Infeasibility *(Value Analysis)*

To specify the *assertion* that a certain block should be or should not be executed, you may write

```
SNIPPET ProgramPoint SHOULD BE REACHABLE;
SNIPPET ProgramPoint SHOULD NOT BE REACHABLE;
```

where *ProgramPoint* refers to an arbitrary instruction in the block, e.g., by its address. a^3 complains if a block that should be reachable is not reachable, or vice versa.

5.18 Values of Conditions

Among other things, value analysis can find that certain conditions are always true or always false in certain contexts. Yet value analysis cannot predict all register values, in particular if they depend on input data. Thus, if you happen to know the values of more conditions, you may declare these values unconditionally (section 5.18.1) or conditionally (section 5.18.2).

5.18.1 Unconditional Values of Conditions

An unconditional declaration of condition values applies to all contexts. It has two variants as follows:

```
CONDITION  ProgramPoint  IS  ALWAYS  TRUE   Action;
CONDITION  ProgramPoint  IS  ALWAYS  FALSE  Action;
```

The *ProgramPoint* specified should be a conditional branch (not a compare instruction). The *Action* indicates what to do with the branch that cannot be entered according to the specified condition value. If *Action* is **exclude**, this branch is not decoded at all (unless it can be reached from somewhere else). If *Action* is **make infeasible**, the branch is decoded, but marked as "infeasible" so that it does not contribute to the worst-case execution time. For backward compatibility, the *Action* may be omitted, which means `exclude`.

Example:

```
condition "prime" + 0x4c bytes is always false exclude;
condition 0x91a4 is always true make infeasible;
```

Since the code in the unreachable branch may be prefetched, we recommend to specify `make infeasible` so that a^3 is aware of the unreachable code. The action `exclude` should only be specified if the code in the unreachable path causes problems when trying to decode it.

The attributes `TRUE` and `FALSE` in a `CONDITION` specification refer to the situation in the executable, not to the condition in the source code. Be aware that compilers often translate a condition in the source to the opposite condition in the executable.

5.18.2 Conditional Values of Conditions

The declaration of a condition value can also include an AIS expression used as guard. There are one-sided and two-sided forms of these declarations. The one-sided form has the following two variants:

```
CONDITION  ProgramPoint  IS  TRUE   IF  Expr;
CONDITION  ProgramPoint  IS  FALSE  IF  Expr;
```

As in the unconditional form, the *ProgramPoint* specified should be a conditional branch (not a compare instruction). The declaration only takes effect in those contexts in which the *Expr* evaluates to "true", i.e. to a set not containing 0. There is no *Action* since there is no choice: both branches must be decoded to prepare for the case that the expression does not evaluate to "true".

The declaration

```
CONDITION  ProgramPoint  IS  TRUE  IF  Expr;
```

is one-sided since it only has an effect if *Expr* evaluates to "true" (then the condition is TRUE). There is no effect if *Expr* does not evaluate to "true"; in this case, the condition may be TRUE or FALSE. This is different in a two-sided declaration, which looks like

```
CONDITION  ProgramPoint  IS  EXACTLY  Expr;
```

In this case, the condition is TRUE if *Expr* evaluates to "true", i.e. to a set of values not containing 0, and the condition is FALSE if *Expr* evaluates to "false", i.e. to the exact value 0. There is no effect if the expression is ambiguous, i.e. evaluates to a set containing 0 and also some non-zero values. Note that the two-sided declaration

```
CONDITION  P  IS  EXACTLY  E;
```

is equivalent to the two one-sided declarations

```
CONDITION  P  IS  TRUE   IF   E  ;
CONDITION  P  IS  FALSE  IF  !(E);
```

where ! is the negation operator "not" (see section 5.28).

5.19 Recursion

The analyzed program may be recursive, i.e. a routine *R* may call itself directly or indirectly (via some other routines). To obtain estimations for stack usage and WCET, the recursion must be bounded by a user annotation. The way to do this is very different: for stack analysis, the maximum call depth must be specified (see section 5.19.1), while for timing analysis, the maximum number of recursive calls is required (5.19.2). A recursion bound is not strictly necessary for value analysis, but is recommended to improve its precision.

5.19.1 Maximum Call Depth *(Stack Analysis)*

A successful stack analysis requires knowledge about the maximum call depth of recursive clusters of routines. This information must be provided by suitable AIS specifications. If there are not enough specifications to bound all recursive clusters, stack analysis fails with an error message.

Call-depth specifications may take one of the following two forms:

```
ROUTINE  R  INCARNATES  Bounds;
ROUTINE  R1  CALLS  TO  R2  INCARNATES  Bounds;
```

Here, R, R_1, and R_2 may be routine names or start addresses of routines.

Routine names must be surrounded by double quotes (like all names).

The *Bounds* information can have either of the following forms:

- `MIN` *i* `MAX` *j*
  specifies the minimum and maximum number of occurrences on the call stack.
- `MAX` *j*
  specifies the maximum number of occurrences. There is no minimum, which is therefore assumed to be 0.
- `EXACTLY` *j*
  is an abbreviation for `MIN` *j* `MAX` *j*.

In general, a call limit l for a routine R restricts the call chains considered by a^3 to those with at most l occurrences of R. A limit l for a call from R_1 to R_2 restricts the call chains considered by a^3 to those which contain at most l occurrences of the call from R_1 to R_2.

Example 1: Assume `main` calls a routine `fac`, which calls itself. Then the specification

```
routine "fac" incarnates max 3;
```

restricts the call chains considered by a^3 to

```
main
main → fac
main → fac → fac
main → fac → fac → fac
```

i.e. at most 3 calls of `fac`, namely the call in `main` that starts the recursion plus two recursive calls. The same result can be achieved by the specification

```
routine "fac" calls to "fac" incarnates max 2;
```

which counts only the calls of `fac` in `fac` (therefore the limit must be one less to achieve the same effect).

More complicated cases are possible. If for instance A calls B and C, B calls A, and C calls A, unrestricted recursion can be avoided by a call limit on A, or by call limits on B and on C.

Note that a^3 has no possibility to check whether a given call limit is a true property of the application. If the user provides a call limit that is not supported by specific measures in the code, the results of a^3 can only be used as hint what would be the stack usage if the call limit were respected by the application. A call limit `limit` > 0 for a directly recursive routine R can for instance be enforced by a static counter in R:

```
... R (...) {
  static int counter = 0;
  counter += 1;
  if (counter < limit) { ...;  R(...); ...; }
  counter -= 1;
}
```

5.19.2 Maximum Number of Calls *(Timing Analysis, Value Analysis)*

In contrast to stack analysis, timing and value analysis work with the maximum number of recursive calls (not the call depth). Therefore, separate annotations are required for stack and timing/value analysis.

5.19.2.1 Direct Recursion

Direct recursion describes the situation in which a routine *R* calls itself by means of one or several recursive calls of *R*. For the following description, you have to distinguish between these recursive calls (located inside *R*) and outside calls (calls of *R* in routines other than *R*). A directly recursive routine requires a user annotation as a declaration specifying the maximum number of calls of *R* induced by each outside call of *R* (including that outside call). Such a specification may look as follows:

```
RECURSION R Bounds;
```

Here, *R* is the name or the start address of the recursive routine.

Routine names must be surrounded by double quotes (like all names).

The `Bounds` information can have either of the following forms:

- `MIN` *i* `MAX` *j*
  specifies the minimum and maximum number of calls of the routine for each outside call. This includes the outside call starting the recursion.
- `MAX` *j*
  specifies the maximum number of calls. There is no minimum, which is therefore assumed to be 0.
- `EXACTLY` *j*
  is an abbreviation for `MIN` *j* `MAX` *j*.

Example: The following routine `fib` is directly recursive with 2 recursive calls:

```
int fib (int n) {
    if (n<=2) return 1;
    else      return fib(n-1)+fib(n-2);
}
```

(This is not the most efficient way to implement that function, but it serves as a good example.) Suppose you know that all outside calls of `fib` are performed with arguments ≤ 10. Then the correct annotation for `fib` can be derived as follows:

- A call `fib(1)` induces one call of `fib`, namely that call `fib(1)` itself.

- The same holds true for `fib(2)`.
- A call `fib(3)` induces recursive calls `fib(2)` and `fib(1)`. The call numbers for these calls are 1, so that the total call number for `fib(3)` (including itself) is 3.
- A call `fib(4)` induces recursive calls `fib(3)` and `fib(2)`. The call number for `fib(3)` is 3 and the call number for `fib(2)` is 1. Thus the total call number for `fib(4)` (including itself) is $3+1+1=5$.
- In general, the call number for `fib(`n`)` is the call number for `fib(`$n-1$`)` plus the call number for `fib(`$n-2$`)` plus 1 (standing for `fib(`n`)` itself).
- An iteration therefore shows that the call number for `fib(10)` is 109. The correct annotation for this example is thus

```
recursion "fib" max 109;
```

In contrast, the maximum call depth of `fib` in this example is 9 since the call stack of maximum length is `fib(10)` $\rightarrow$ `fib(9)` $\rightarrow \cdots \rightarrow$ `fib(2)`. Therefore, the correct annotation for stack analysis is

```
routine "fib" incarnates max 9;
```

5.19.2.2 Simple Direct Recursion

The situation is much simpler if R contains a single recursive call to itself (simple direct recursion). In this case, the call number is the same as the call depth, i.e. the maximum number of calls of R in a call chain (including the call that starts the recursion). Nevertheless, two different annotations are required for stack and timing analysis.

Example 1: The following routine is implemented with simple direct recursion:

```
int fac (int n) {
    if (n <= 1)  return 1;
    else         return n * fac(n-1);
}
```

Suppose you know that all outside calls of `fac` are performed with arguments ≤ 10. Thus the worst case is `fac(10)`, which calls `fac(9)`, which calls `fac(8)`, etc. all the way down to `fac(1)`, which does not call `fac` any more. Therefore, the call number for `fac(10)` equals the call depth, which is 10. The correct annotation for this example is thus

```
recursion "fac" max 10;              # for timing analysis
routine "fac" incarnates max 10;     # for stack analysis
```

Example 2:

```
... R (...) {
  static int counter = 0;
  counter += 1;
  ...
  if (... && counter < 2)  R(...);
  ...
  counter -= 1;
}
```

Under the assumption that `counter` is not modified anywhere else, routine `R` has a maximum call number = call depth of 2: when called from outside `R`, it sets `counter` to 1 and then may call itself. This second incarnation of `R` sets `counter` to 2 and then cannot call `R` again. So the appropriate specification for this case is

```
recursion "R" max 2;              # for timing analysis
routine "R" incarnates max 2;     # for stack analysis
```

5.19.2.3 General Recursion

The simplest form of general (indirect) recursion consists of two routines R_1 and R_2 such that R_1 calls R_2 and R_2 calls R_1. Here, two cases must be distinguished:

1. If only one routine, say R_1, has an outside call, while the other one (R_2) is only called within R_1, then a `recursion` annotation can be applied to R_1. The bound to be used in the annotation is the maximum number of calls of R_1 for each outside call of R_1 (including that outside call).
   ⚠ A `recursion` annotation applied to R_2 takes no effect.
2. If both routines have outside calls, then `recursion` annotations cannot be applied.

This distinction also applies to the most general case in which many routines call each other recursively. If only one of them has an outside call, a `recursion` annotation can be applied to it, but this is impossible if more than one routine has outside calls. For timing analysis, flow constraints (section 5.22) can be used instead, but these do not work for value analysis.

5.19.2.4 Implementation Restriction

A `recursion` annotation applied to the start point of the analysis may lead to a gross overestimation of the worst-case execution time, up to an unbounded time, and to imprecision in value analysis caused by extra execution contexts or an unbounded call number (as if there were no annotation). The problem with this case is that there is no outside call to start with.

5.20 Loop Bounds

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. Value analysis becomes more precise if loop bounds are known, but knowledge of the bounds is not strictly required. Stack analysis does not need loop bounds unless the loop body has a non-zero stack effect. If for instance 4 bytes are pushed on the stack in each iteration, then a^3 must know the number of iterations to calculate the overall stack usage.

If loop bound analysis is included in your license, a^3 tries to determine the number of loop iterations by a static analysis, but succeeds in doing so for simple loops only (see section 4.4.1). The results of loop bound detection can be found in the report file whose name is provided in the `Files` view (see section 3.4.1), and in the first info fields of the loop call nodes (see section 7.4.4). Bounds for the iteration numbers of the loops for which loop analysis did not succeed must be provided by the user in the AIS file or in source code annotations (see section 5.31).

Neither automatic loop bound analysis nor loop bound annotations can be applied to multi-entry loops (also called irreducible loops) because such loops are not detected by the loop transformation (see section 4.1.4). For timing analysis, bounds for multi-entry loops can be specified by means of flow annotations (see section 5.22). Flow annotations are ignored by stack analysis and value analysis.

Loop bound specifications can be classified in two different ways. On the one hand, there is a distinction between local loop bound specifications applying to a single loop (section 5.20.1) and global annotations applying to all loops in a set of routines (section 5.20.3). On the other hand, there is a distinction between absolute specifications with higher priority than automatic results and default specifications with lower priority. All global specifications are default specifications, and so all absolute specifications are local specifications.

Absolute specifications may specify loop bounds by arbitrary expressions as described in section 5.28 while default specifications are restricted to use immediate constants as loop bounds. Loop bounds given by non-trivial expressions (containing not only constants) are called *parametric loop bounds*. In general, AIS specifications uniformly apply to all contexts, but since the value of a non-trivial expression may depend on the context, parametric loop bounds are a way to specify context-sensitive loop bounds. Note that a parametric loop bound may fail to evaluate to a concrete loop bound since it may be based on registers with unknown values.

The various ways to obtain loop bounds interact as follows:

1. Absolute loop bound specifications have highest priority. They supersede any automatic results and default specifications. This is even true for parametric loop bounds not evaluating to concrete loop bounds; if there is an automatic result that is overwritten by an absolute loop bound with an unbounded expression, then the automatic result is lost.

2. A default local specification for a loop takes only effect if there is no result from Step 1, i.e. either no absolute specification and no automatic result, or an absolute parametric specification failing to produce a result (no matter whether automatic analysis produced a result or not).

3. Global loop annotations apply only to loops for which there are no results from Step 2.

If the results of automatic analysis are overwritten by a user annotation, a^3 tries to run the loop as often as the user has specified, but this may prove impossible so that the automatic bound seems to prevail, or may lead to contradictions if the user specification is wrong.

⚠ Loop bound analysis is performed separately for each execution context, but AIS specifications uniformly apply to all contexts. So it may happen that loop bound analysis finds a loop bound for Context 1, but fails for Context 2. Then default loop bounds take only effect in Context 2, not in Context 1.

The concept of execution contexts is introduced in section 4.3.

⚠ If the specified loop bounds are not correct, the results of a^3 may be incorrect. This remark includes loops whose loop bound can be found by a^3 since user-given absolute loop bounds supersede the results of the automatic loop bound detection.

If there are still unbounded loops after reading all user annotations and running the automatic loop bound detection, a^3 states "This problem is unbounded" when it performs a timing analysis, and fails to obtain a bound for the worst-case execution time.

a^3 complains about missing loop bounds in timing analysis by messages listing the start addresses of the loops that lack loop constraints.

⚠ a^3 issues such messages only for loops that are recognized by the loop transformation. If path analysis says "The problem is unbounded" (see section 4.4.4) and there are no "Loop bound missing" messages, then watch out for loops with several entry points (irreducible loops). Such loops cannot be annotated directly; see section 5.22 for other ways to annotate them for timing analysis. (There is no way to bound them for stack and value analysis.)

5.20.1 Local Loop Bound Specifications

A local loop bound specification may have the following forms:

```
LOOP  ProgramPoint  Qualifier  Bounds;
LOOP  ProgramPoint  Bounds  Qualifier;
LOOP  ProgramPoint  Qualifier  SimpleBounds  BY DEFAULT;
LOOP  ProgramPoint  SimpleBounds  Qualifier  BY DEFAULT;
```

The first two forms are equivalent, and so are the second two forms. The forms without `DEFAULT` specify absolute local loop bounds taking priority over the automatic loop bound analysis, while the forms ending in `BY DEFAULT` specify default local loop bounds that are overridden by automatic results (and absolute local bounds).

A *`ProgramPoint`* is either an address or a loop expression. The address must be the start address of the loop; this is the same address as in the messages about missing loop bounds. A loop expression looks like

```
"R"  +  n  LOOPS
```

which means the *n*th loop in routine *R*, counted from 1. (The plural ending may be omitted; `loop` and `loops` are equivalent.)

Loop counting refers to the *executable*. If for instance the source code of a routine contains two loops and the compiler decides to completely unroll the first loop, then `+ 1 loop` refers to the second loop in the source. The unrolled loop does not require an annotation because it is not a loop in the executable.

Loop routines are numbered according to the address of their entry point. If two nested loops are entered by fall-through from the code before the loop, the entry of the outer loop is before the entry of the inner loop, and so the number of the outer loop is smaller than the number of the inner loop. Yet if two nested loops are entered by jumps to their last basic blocks, then the entry of the outer loop lies behind the entry of the inner loop, and so the inner loop has the smaller number.

The *Bounds* information can have either of the following forms:

- `MIN` *I* `MAX` *J*
  specifies the minimum and maximum execution count for the loop body *in the executable*. ⚠ It may differ from the execution count derived by looking at the source since the compiler may partially unroll loops.

  Here, *I* and *J* may be expressions as described in section 5.28. Such expressions evaluate to a set of fractions in general, which may be unbounded (extend till infinity). If *I* evaluates to a set with least element $i_1 \geq 0$, then $\mathit{floor}(i_1)$ (i_1 rounded down to the nearest integer) is taken as the minimum execution count of the loop. If i_1 is negative, it is replaced by 0. If *J* evaluates to a set with greatest element $j_2 \geq 0$, then $\mathit{ceil}(j_2)$ (j_2 rounded up to the nearest integer) is taken as the maximum execution count of the loop. If j_2 is negative or smaller than the lower bound i_1, the entire annotation is ignored; the loop remains unbounded in this case.

- `MAX` *J*
  specifies only the maximum execution count. There is no minimum, which is therefore assumed to be 0.

- `EXACTLY` *J*
  combines the effects of `MIN` *J* and `MAX` *J*: if *J* evaluates to a set with least element j_1 and greatest element $j_2 \geq 0$, then $\mathit{max}(0, \mathit{floor}(j_1))$ is taken as the minimum execution count and $\mathit{ceil}(j_2)$ as the maximum execution count of the loop. If j_2 is negative, the annotation is ignored; the loop remains unbounded in this case.

- *J*
  is the same as `EXACTLY` *J*. The keyword `EXACTLY` was introduced at a time when loop bounds had to be immediate constants. While the wording `EXACTLY` *J* is appropriate for constants *J*, e.g., `EXACTLY 10`, it is a bit misleading for expressions *J* producing a set of values, e.g., `EXACTLY (0..10)`. Therefore, the keyword `EXACTLY` became optional.

Default loop bound annotations involve *SimpleBounds* instead of *Bounds*. *SimpleBounds* have the same structure as *Bounds* (described above), but the expressions I and J are restricted to be immediate constants, namely non-negative integers such as 5 or 11. Yet this restriction does not reduce the expressivity of the AIS language: instead of *E* `by default`, you may write `try (value,` *E*`)` using the `try` operator described in section 5.28.6. The value of `try (value,` *E*`)` is the result of loop bound analysis if this is a bounded set, and the value of *E* otherwise (see also the examples presented in section 5.20.2).

Qualifier is one of the following:

- `begin` indicates that the loop test is at the beginning of the loop, as for C's `while`-loops.
- `end` indicates that the loop test is at the end of the loop, as for C's `do-while`-loops.
- `unknown`, which lets a^3 assume the worst case of the two possibilities, which is `begin` where the loop test is executed one more time.
- An unknown qualifier may be omitted.

The `begin`/`end` information refers to the *executable*, not to the source code; the compiler may move the loop test from the beginning to the end, or vice versa.

More exactly, `begin` means that the loop test is near the entry point of the loop, no matter whether the entry point is at the beginning or end. Thus you should specify `begin` if the loop test is in the entry block of the loop – unless the loop consists of a single block only; in that case you should specify `end`.

A different view on these qualifiers is that you specify the number of iterations of the loop *body* when writing `begin`, and the number of iterations of the loop *test* when writing `end`. Internally, a^3 always works with the number of iterations of the loop test, which is 1 more than the number of iterations of the body.

The maximum number of executions of the loop *test*, which is taken as the number of loop contexts if the `max-unroll` parameter in the context specification is sufficiently high (see section 5.4), equals the specified maximum value only if *Qualifier* is `end`. If *Qualifier* is `begin` or `unknown`, it is one more.

The *minimum* number of executions of the loop test equals the specified minimum value if *Qualifier* is `end` or `unknown`. If *Qualifier* is `begin`, it is one more.

In its output, a^3 also shows the number of iterations of the loop *test* (see section 7.4.4). Therefore, the reported number of loop iterations is one more than the specified number if the specification contains the `begin` qualifier.

Example 1: `loop "prime" + 1 loop end max 10;`
specifies that the first loop in `prime` has the loop test at the end and is executed at most 10 times. This specification is absolute, i.e. overrides any automatic results.

Example 2: `loop "main" + 2 loops max 5 begin by default;`
specifies that the second loop in `main` has the loop test at the beginning and is executed at most 5 times. This is a default specification that is ignored if there are any automatic results for this loop (or an absolute local specification for the same loop).

5.20.2 Examples with Parametric Loop Bounds

In principle, an AIS annotation uniformly applies to all contexts (contexts have been introduced in section 4.3). Thus, a loop bound specification with a constant defines the same loop bound for all contexts. A parametric loop bound specification with an expression also applies to all contexts, but may lead to context-dependent loop bounds since the value of the expression may depend on the context. Consider the following example:

```
if (C) { A; R(); }
else   { B; R(); }
```

where routine `R` contains a loop with at most 10 loop iterations if `R` is called after `A`, but at most 20 loop iterations if `R` is called after `B`. These facts may be expressed in AIS as follows:

```
instruction A is entered with @RLoopMax = 10;
instruction B is entered with @RLoopMax = 20;
loop "R" + 1 loop max @RLoopMax;
```

The first two annotations define a user register called `@RLoopMax` (see section 5.10). It is given value `10` at the beginning of `A` and value `20` at the beginning of `B`. These values are propagated through the calls of `R`, and the third annotation specifies that the maximum number of iterations of the loop in `R` is given by `@RLoopMax`, i.e. is 10 in the context of the call after `A` and 20 in the context of the call after `B`.

Parametric loop bounds are also useful if the loop bound depends on function parameters held in some registers. For example if the number of iterations of a loop depends on the number of padding bytes that can occur in a frame according to the formula $floor\,(\frac{T-D}{4})$ where the total frame length T is held in register `r0` and the number D of data bytes is held in register `r1`, then the annotation could look like this:

```
loop "FrameByID" + 4 loops begin  max  floor ((r0 - r1) / 4);
```

The advantage here is that the evaluated loop bound may depend on the context in which the loop is executed. Without the possibility of parametric loop bounds, only a fixed bound could have been specified for all contexts. So the parametric loop bound annotation reduces overestimation in WCET analysis.

A problem is that a parametric loop bound annotation like the above fails to produce a concrete bound if the value analysis performed by a^3 does not find the values of all the parameter registers (`r0` and `r1` in this example). To cope with this, an additional default value may be supplied using the `try` operator described in section 5.28.6. An expression `try (`E_1`,` E_2`)` is equivalent to E_1 if this expression yields a bounded result set, and equivalent to E_2 otherwise. The expression E_2 thus supplies a default value if E_1 fails to produce a bounded set. If you want to supply for instance a loop bound of 256 if `r0` or `r1` cannot be evaluated, then you may write

```
...  max  try (floor ((r0 - r1) / 4), 256);
```

Instead of supplying a default for the entire expression, you may supply defaults for the register values themselves. Assume you know that `r0` is at most 1024 and `r1` is not negative, then you may write

```
...  max  floor ((try (r0, 1024) - try (r1, 0)) / 4);
```

This gives the same result as the previous expression if both registers are unknown, but gives more precise results if one of them is known.

A more complicated example may involve binary search with variable arguments. This case can also be handled by a parametric loop bound depending on the value in some register. Here is an example with a source code annotation, which relies on the fact that parameter `size` is passed in register `R2`:

```
function search (array a, int size, int x) {
  ...
  while (...) {
    /* ai: loop here  max ceil (log (R2, 2)); */
    /* Binary search over a */
    ...
  }
}
```

The expression `log (R2, 2)` evaluates to the binary logarithm of the value of `R2` (approximately – see section 5.28.8). The `ceil` operator rounds up to the nearest integer. Actually, it is redundant in this example since it is implied by `max` as described in section 5.20.1.

Now suppose the value of `R2` is modified between the entry of `search` and the beginning of the loop. To refer to the original value in the loop bound, it may be saved in a user register:

```
function search (array a, int size, int x) {
/* ai: instruction "search" is entered with @size = R2; */
  ...
  while (...) {
    /* ai: loop here  max ceil (log (@size, 2)); */
    /* Binary search over a */
    ...
  }
}
```

Parametric loop bounds based on constants and (user) registers overwrite the results of the automatic loop bound analysis. This holds even if the parametric loop bound cannot be evaluated to a bounded set; in this case, the results of the automatic analysis are lost. To overcome this problem, a special variable **`value`** has been introduced whose value is the result of the automatic analysis; it is an unbounded set if the automatic analysis failed (see section 5.28.14).

Using this special variable, the results of the automatic analysis may be amended or modified. For the following examples, assume *E* is an expression not containing `value`.

- `loop ... max` *E*
  replaces the result of value analysis by the result of *E* (even if this result is unbounded).

- `loop ... max restrict (value,` *E*`)`
  restricts the result of value analysis by the result of *E*, by forming the intersection of the two result sets (see section 5.28.6 for `restrict`).

  If the analysis produces nothing, the result of *E* is taken, and vice versa. Otherwise the two results are combined. If for instance *E* yields the interval $(5..7)$, then a less precise analysis result of $(2..9)$ is restricted to $(5..7)$, but a more precise result of 6 is preserved. The two pieces of information are even combined: an analysis result of $(2..6)$ is restricted to $(5..6)$.

- `loop ... max try (value,` *E*`)`
  supplies *E* as a default value that is taken if the automatic analysis produces an unbounded or undefined result (see section 5.28.6 for `try`).

  If the analysis produces nothing, the result of *E* is taken, and vice versa. Yet if the analysis produces a bounded set, *E* is ignored even if it would be more precise. If for instance *E* yields the interval $(5..7)$, then a less precise analysis result of $(2..9)$ is still preserved. Only if the automatic analysis fails to produce a bounded set, $(5..7)$ is taken.

- If you trust in the automatic analysis, but if it fails, you have an expression *E*, and if that fails, too, you wish to fall back to 1000, you should write

  `loop ... max try (value,` *E*`, 1000).`

- If you believe that your expression *E* is better than value analysis, you may write

  `loop ... max try (`*E*`, value, 1000).`

- If you believe that your expression *E* is in principle as good as value analysis, so one should take whichever yields more precise results, then you can write

  `loop ... max try (restrict (value,` *E*`), 1000).`

  Here, 1000 serves as a fall-back solution if both the automatic analysis and the expression *E* fail to produce a bounded set.

- If you are sure that 1000 is a valid upper bound that should even be taken if value analysis and *E* yield, say 2000, you can write

  `loop ... max restrict (value,` *E*`, (0..1000)).`

5.20.3 Global Loop Bound Specifications

Global loop bound specifications provide information about all loops in a set of routines or even in the entire executable. They exist in three different forms:

```
LOOP-ITERATION  DEFAULT  Routines  IS ...;
GLOBAL  LOOP-ITERATION  DEFAULT  IS ...;
GLOBAL  LOOP-ITERATION  DEFAULT  EXCEPT Routines  IS ...;
```

In the first and third forms, ***Routines*** stands for a comma-separated list of routine names (enclosed in double quotes as usual). Each annotation applies to a certain part of the executable only, called the *application domain* of the annotation. The application domain of the first form consists of the code of the listed routines (including all loop routines extracted from these routines). The application domain of the second form (`GLOBAL`) consists of the entire executable (as far as it is analyzed). The application domain of the third form (`GLOBAL ... EXCEPT`) is the entire executable minus the listed routines. As indicated by the word `DEFAULT`, global loop bound specifications apply to all loops in the application domain that are not bounded by other means (loop bound analysis or local loop bound specifications (absolute or default)).

The part after the keyword `IS` has the same syntax and meaning as the end of a default local loop bound specification: it consists of an optional *`Qualifier`* and *`SimpleBounds`* for the loop in any order as described in section 5.20.1. Recall that *`SimpleBounds`* are restricted to integer constants (no expressions).

Example 1: You want to express that all loops in the application perform at most 20 iterations.

```
global loop-iteration default is max 20;
```

This specification is even valid if there are a few loops with more than 20 iterations provided that each of these loops is annotated by a local specification, or their bounds are found automatically.

Example 2: Loops with more than 10 iterations occur in routines `large1` and `large2` only. Then you may specify

```
global loop-iteration default except "large1", "large2"
       is max 10;
```

If several different global loop specifications apply to the same loop, their loop bound intervals are intersected. For instance, `min 10 max 30` and `min 20 max 40` are combined to `min 20 max 30`.

Sometimes, a^3 cannot find a loop bound by itself, but is able to prove that a loop bound given by a default specification is wrong. In this case, a^3 states “This problem is infeasible” when it performs a timing analysis.

Global loop bound annotations do not apply to loops with multiple entries, which are not detected by the loop transformation (see section 4.1.4). Such loops can only be bounded by local specifications as presented in section 5.22. These specifications take effect in timing analysis only. There is no way to bound multi-entry loops for stack and value analysis.

5.21 Timing Specifications for Loops *(Timing Analysis)*

Timing specifications for loops are relevant for timing analysis only.

Sometimes it is natural to provide a bound on the execution time that is spent in a loop instead of specifying the maximum number of iterations. For instance, the abort conditions of loops used for

busy waiting depend on the change of the value in an operational register but are not expressed in terms of a variable that counts the loop iterations.

Timing annotations for loops have the following form:

```
SNIPPET ProgramPoint TAKES time;
```

ProgramPoint is either an address or a loop expression (see section 5.20.1). *time* can have either of the following forms:

- `MIN` *i* `MAX` *j* `CYCLES`
  specifies the minimum and maximum number of cycles required for executing the loop.
- `MAX` *j* `CYCLES`
  specifies the maximum number of cycles required for executing the loop. There is no information about the minimum number, which is therefore assumed to be 0.
- `EXACTLY` *j* `CYCLES`
  is an abbreviation for `MIN` *j* `MAX` *j* `CYCLES`.

If the clock rate of the processor has been specified before (see section 5.2), the execution time can also be specified in seconds `sec`, milliseconds `msec`, microseconds `usec`, or nanoseconds `nsec`. The numbers *i* and *j* may be integers, floats, or fractions.

The above declaration specifies the execution time of one execution of the loop routine that corresponds to the loop that is referred to by *ProgramPoint*. The code snippet that is covered by the timing annotation thus includes the loop routine itself and all routines that are reached from there.

Timing specifications for loops are evaluated during path analysis (see section 4.4.4). Therefore, their usage is only possible if the `Path analysis variant` is set to `ILP based` in the `General` tab of the `Analyses` view, see section 3.4.3.7.

Timing specifications for loops are not applicable to multi-entry loops (also called irreducible loops) because they cannot be transformed to loop routines (see section 4.1.4).

Example: The following annotation expresses that the execution of the second loop in routine `"reset"` needs at most 10 milliseconds.

```
snippet routine "reset" + 2 loops takes max 10 msec;
```

5.22 Flow Constraints *(Timing Analysis)*

Flow constraints are ignored by stack and value analysis and therefore relevant for timing analysis only.

The loop annotations described in section 5.20 above can only be applied to loops that have been turned into loop routines by the loop transformation. Loops with several entries (also called irreducible loops) are not transformed into loop routines (section 4.1.4) and thus cannot be annotated

by loop specifications. Worse yet, they are hard to find because there is no easily recognizable loop routine, and a^3 does not produce a message such as "loop bound missing at start node `0x10f8`" that would reveal the address of the beginning of the loop.

Once a loop with several entry points has been identified, a loop bound can be specified for it by means of a *flow constraint* that bounds the relative execution counts of two basic blocks. Yet flow constraints are not restricted to giving loop bounds to multi-entry loops. They can be applied to any program points to specify some relationship between the execution counts of these points. Flow constraints exist in two versions: a^3 originally supported only constraints specifying lower and upper bounds for the ratio of two execution counts. Later, arbitrary linear constraints were introduced. Anything that could be expressed by the old ratio constraints can also be expressed by linear constraints – and much more. Hence, ratio constraints are now obsolete, but they are still supported. We first describe the more powerful linear constraints (section 5.22.1), and thereafter the old ratio constraints (5.22.2) with hints how to translate them into linear constraints. Finally, section 5.22.3 describes the interaction between constraints and contexts.

5.22.1 Linear Flow Constraints

Linear flow constraints basically look as follows:

```
FLOW  Qualifier  Expression  rel  Expression;
```

- *Qualifier* is SUM or EACH, which specify the behavior w.r.t. contexts (see section 5.22.3). The *Qualifier* may be omitted with default SUM.
- *rel* is <=, >=, ==, or =. The two variants of equality (== and =) carry the same meaning.
- An *Expression* is built from one or more *Term*s by addition or subtraction, i.e. looks like

  ```
  Term op ... op Term
  ```

  with *op* being + or −. It may as well consist of a single *Term*.
- A *Term* can have either of two forms:

  ```
  constant
  factor ( ProgramPoint )
  ```

  The first form is an absolute constant, while the second form stands for the execution count of the *ProgramPoint* multiplied by the given *factor*. The *constant* and the *factor* must be *non-negative* integer numbers. A *factor* of 1 can be omitted.

Syntactic restriction: The first term on the left-hand side of the constraint must not be a *constant*.

For instance, to specify that a loop with several entries has a loop bound of 4, find a block B_1 in the loop body that is executed in each loop iteration, and a block B_2 before or after the loop. Assume the start addresses of B_1 and B_2 are `0x8100` and `0x8200`, respectively. Then the constraint

```
flow each (0x8100) <= 4 (0x8200);
```

means that the number of executions of B_1 is at most 4 times the number of executions of B_2, i.e. the loop body is executed at most 4 times as often as the code around the loop.

Program points are not restricted to absolute addresses (see section 5.29). Annotations with addresses relative to a routine entry remain valid when the routine moves in the memory:

```
flow each ("R" + 0xAC bytes) <= 4 ("R" + 0xC4 bytes);
```

Note that flow constraints may be used for other purposes than to provide bounds for the number of iterations of multi-entry loops. They can be applied whenever there is a known relationship between the execution counts of some basic blocks. Here are some more examples:

```
flow each ("R" + 1 computed) <= 1;
flow sum  3 (0x2456) + 4 (0x3260)
          <= 5 ("prime") - 2 ("even");
```

5.22.2 Ratio Flow Constraints

The older ratio flow constraints are less powerful than the more modern linear flow constraints. They only admit (limited) statements about the relationship between two program points. Their syntax is as follows:

```
FLOW  Qualifier  ProgramPoint1 / ProgramPoint2
      IS  MIN min  MAX max;
```

- The *Qualifier* is the same as in linear constraints: it is SUM or EACH, which specify the behavior w.r.t. contexts (see section 5.22.3). The *Qualifier* may be omitted with default SUM.
- *ProgramPoint*$_1$ and *ProgramPoint*$_2$ denote the start addresses of two basic blocks, say b_1 and b_2.
- *min* is the minimum value of the ratio between the execution count of b_1 and the execution count of b_2.
- *max* is the maximum value of the ratio between the execution count of b_1 and the execution count of b_2.
- As in loop specifications, MIN 0 can be omitted and MIN *n* MAX *n* can be replaced by EXACTLY *n*.

For instance, to specify that a loop with several entries has a loop bound of 4, find a block B_1 in the loop body that is executed in each loop iteration, and a block B_2 before or after the loop. Assume the start addresses of B_1 and B_2 are 0x8100 and 0x8200, respectively. Then the constraint

```
flow each 0x8100 / 0x8200 is max 4;
```

means that the number of executions of B_1 is at most 4 times the number of executions of B_2, i.e. the loop body is executed at most 4 times as often as the code around the loop.

Clearly, each ratio constraint can be replaced by one or two equivalent linear constraints.

```
flow Q P1 / P2 is min a max b;
```

is equivalent to the two constraints

```
flow Q (P1) >= a (P2);
flow Q (P1) <= b (P2);
```

and

```
flow Q P1 / P2 is exactly n;
```

is equivalent to the constraint

```
flow Q (P1) = n (P2);
```

5.22.3 Constraints and Contexts

The qualifiers `SUM` and `EACH` define whether the flow constraint applies to all contexts cumulatively or to each context separately (contexts are introduced in section 4.3). With `SUM` (the default qualifier), the constraint does not apply to the execution counts in individual contexts, but to the sum over the number of executions in all contexts. For instance with the constraint

```
flow sum (0x8100) <= 4 (0x8200);
```

and two contexts, path analysis may assign to block `0x8200` an execution count of 1 in each context, and to block `0x8100` a count of 8 in the first context and 0 in the second. These assignments satisfy the constraint since $(8+0) \leq 4 \cdot (1+1)$.

In contrast, a flow constraint qualified by `EACH` applies to each context individually. For instance with the constraint

```
flow each (0x8100) <= 4 (0x8200);
```

and two contexts, block `0x8100` gets a count of at most 4 in each context if `0x8200` has a count of 1 in each context. Therefore `flow each` constraints lead to more precise WCET estimates than `flow sum` constraints (unless they contain absolute constants – see below). On the other hand, the usage of `flow each` constraints is restricted: Their meaning is well-defined only if the two program points are in the same routine so that they have the same contexts. Here "in the same routine" refers to the situation *after* loop transformation i.e. loop routines are distinguished from ordinary routines. This does not restrict the usage of `flow each` for multi-entry loops since such loops are not transformed into special loop routines. If a `flow each` constraint is applied to program points in different routines, a^3 complains with the message

```
pathan2: Error:
Nodes for user constraint are not in the same routine.
```

In contrast, `flow sum` constraints can be applied to arbitrary program points.

If a linear constraint does not contain absolute constants, it is safe to replace `each` by `sum`, in particular if `each` is rejected because the program points involved occur in different routines. This is no longer true if the constraint contains non-zero constants. The constraint

```
flow each ("prime" + 16 bytes) <= 1;
```

means that the indicated program point in `prime` is executed at most once in every context, while

```
flow sum ("prime" + 16 bytes) <= 1;
```

means that the program point is executed at most once in all contexts together, which is much stronger.

Constants should be used with care anyway. If you want to express that `"prime" + 16 bytes` is executed at most once in every call of `prime`, you may be tempted to write

```
flow each ("prime" + 16 bytes) <= 1;
```

assuming that every call gives a new context. However, this is only true if the call string length is large enough because otherwise, several different calls may share the same context, which makes the constraint above incorrect. The constraint with the intended meaning avoids the use of absolute constants:

```
flow each ("prime" + 16 bytes) <= ("prime");
```

5.23 Output from Value Analysis

Note that these commands have only an effect if a value analysis is performed (see section 3.5.1).

The AIS format admits certain directives that do not specify declarations or assertions, but instruct a^3 to write certain kinds of output into the report file when performing a value analysis: who calls some routine (5.23.1), who accesses some memory area (5.23.2), the control-flow successors of an instruction (5.23.3), and the contents of registers and memory cells as far as known by a^3 (5.23.4). Section 5.24 presents commands for producing output when performing a *timing* analysis.

5.23.1 Routine Callers

The directive

```
ROUTINE Point IS WATCHED;
```

instructs a^3 to report all calls to the specified routine that occur in the analyzed code. Here, *`Point`* must be a program point defining a routine, i.e. a routine name (quoted as usual) or the start address of a routine.

5.23.2 Memory Accessors

```
AREA Area IS WATCHED;
```

instructs a^3 to report all accesses to the specified memory area that occur in the analyzed code. Here, *Area* is a memory area definition as described in section 5.12.1.

5.23.3 Control-Flow Successors

```
INSTRUCTION Point FEATURES "show" = "successors";
```

instructs a^3 to display all possible control-flow successors of the specified instruction (of course only those that occur in the analyzed piece of code).

5.23.4 Output of Register and Memory Contents

```
ROUTINE Point IS INTERESTING;
```

causes a^3 to output the value analysis results that hold at the beginning of the specified routine. These are the contents of the registers and memory cells as far as they are known to a^3. The output is written in AIS syntax as described in sections 5.10 (registers) and 5.12.5 (memory). The rationale for using AIS syntax is that these AIS annotations can be used to specify the initial state for a new analysis starting at the specified routine. Since AIS syntax does not admit any reference to contexts, the value ranges written by a^3 are joins over all contexts. If for instance `r2` has value 1 in one context and 7000 in another one, then the value range for `r2` will be the interval $(1\,..\,7000)$ that includes all intermediate values as well.

5.24 Output from Timing Analysis

Note that these commands have only an effect if a timing analysis is performed (see section 3.5.1).

This section presents directives for writing certain kinds of output into the XML report file during timing analysis. Section 5.23 presents commands for output during value analysis.

5.24.1 Counting Accesses to Some Memory Area

It is possible to specify a *single* memory area for which accesses on the WCET path are counted for every start point, i.e. for the overall start point of the analysis and any additional start points (section 5.5):

```
AREA  Area  COUNT ACCESSES;
```

where *Area* is an area description as introduced in section 5.12.1. A typical example is

```
area 0x3000 .. 0x3fff count accesses;
```

The results of access counting appear in the XML report file (but not in the textual report files). For each start point, there is a subelement `wcet_analysis` of `wcet_analysis_task`. The results of access counting for a specific start point are contained in the element `wcet_analysis` → `wcet_results` → `wcet_access_infos` of the `wcet_analysis` element corresponding to this start point. A typical example is

```
<wcet_access_infos>
  <wcet_area begin=0x3000" end="0x3fff" />
  <wcet_access_info routine="r15"
                    reads="1" writes="2" unknown="0" />
</wcet_access_infos>
```

The element `wcet_area` documents the memory area specified in the AIS file. The routine name stated in `wcet_access_info`, e.g., `"r15"`, is an internal name. Internal names are mapped to the "real" routine names in another part the XML report (`wcet_analysis_task` → `decode` → `routines`).

- `reads` is the number of definite read accesses on the WCET path into the area. Definite means that the range of possible read addresses as determined by a^3 is included in the given area.
- `writes` is the number of definite write accesses on the WCET path into the area.
- `unknown` is the number of potential, but not definite accesses on the WCET path into the area, i.e. of those accesses whose computed address range overlaps the given area without being included in it.

The counts represent numbers of accesses, not numbers of instructions. For instance, an instruction accessing the specified memory area in a loop with 100 iterations is counted 100 times if that loop is on the calculated WCET path.

Accesses can be counted for a single memory area only.

Only accesses on the WCET path as calculated by a^3 are counted. Accesses in routines or basic blocks that are not on the WCET path are not counted.

5.25 Symbolic Names for Program Points and Areas

The purpose of a label specification is to introduce an artificial label as a name for a program point or an area. These labels may be used in subsequent specifications as convenient abbreviations for complex program point or area descriptions. The format of label specifications is as follows:

```
LABEL  ProgramPoint   =   "Name";
LABEL  ProgramPoint1 .. ProgramPoint2   =   "Name";
```

In both cases, *Name* is the name of the new label. In the first case, *ProgramPoint* is the program point the new label refers to. In the second case, $ProgramPoint_1$ .. $ProgramPoint_2$ is the area the new label refers to. Whenever you want to refer to this program point or area later on, you may write

LABEL "*Name*" or "*Name*"

instead of the original description of the program point or area.

If a label name *N* is also a routine name, then the bare name "*N*" refers to the label. The ambiguity can be resolved by writing LABEL "*N*" or ROUTINE "*N*".

Symbolic loop descriptions of the kind *R* + *n* loops are not supported as program points in label specifications.

If a label *L* happens to denote the same program point as a routine *R*, e.g., because of label "*R*" = "*L*", then the routine name *R* is still used in the output of a^3. This behavior can be changed by adding the keyword **force** to the label name:

LABEL *ProgramPoint* = FORCE "*Name*";

Then the routine name *R* is replaced by the label name *L* in the graph pictures and the report files produced by a^3. However, the routine name *R* remains visible in the symbol table in the Symbols view (section 3.6.2) and the list of possible start points for analyses (section 3.5.5); the label name *L* does not show up there. Nevertheless, labels defined in the AIS file can be used as start points (section 3.5.2) although they do not show up in the list of possible start points, no matter if force is used or not.

5.26 Avoiding Problems with Unreachable Program Points

exec2crl complains if a *ProgramPoint* referred to in an AIS file is not reachable from the start point of the analysis. If you use for instance the specifications

```
instruction "MC" + 1 computed ...;
instruction "MC" + 2 computed ...;
```

presented in section 5.8.2, but routine MC is not reachable from the selected start point, then **exec2crl** complains with error messages of the following kind:

```
Unable to resolve 0x181cc + 1 computed.
Unable to resolve 0x181cc + 2 computed.
```

where 0x181cc is the start address of MC in the analyzed executable. This kind of error can be avoided by writing **TRY** in front of the specifications:

```
try instruction "MC" + 1 computed ...;
try instruction "MC" + 2 computed ...;
```

A `TRY` directive can also be applied to several specifications grouped by braces as in

```
try { instruction "MC" + 1 computed ...;
      instruction "MC" + 2 computed ...;
    }
```

In fact, `TRY` does not only suppress the above-mentioned error messages, but *all* error messages resulting from the annotations in its scope. Thus, `TRY` should not be applied before the annotations are completely debugged.

5.27 Include Directives

Include directives in AIS files are similar to the `#include` directives of the C preprocessor: they tell a^3 to suspend reading the current AIS file, to read the contents of another file instead, and then to resume reading the original file. The format of include directives is as follows:

```
INCLUDE  'Name';
```

where *Name* is the name of the file to be read. This way you may for instance include an AIS file describing features of the C library.

The single quotes around *Name* let the path name separator `\` of MS Windows stand for itself. With double quotes, `\` would have to be written as `\\` like in C-strings (see also section 5.30.2).

If source code reading is enabled by your license, the file to be included is not restricted to be an AIS file again. It may as well be a `.c` or `.h` file. In this case, **`exec2crl`** reads the source annotations in the included file (see section 5.31). The purpose of this feature is to let **`exec2crl`** read source files whose names have not been included in the debug information of the executable.

5.28 Expressions and their Evaluation

In this section, we describe the full syntax and semantics of the expressions that may occur at various syntactic places:

- In specifications of register values and in definitions of user registers, e.g.,
  `instruction` *P* `is entered with r1 =` *Expr*$_1$`, @X =` *Expr*$_2$`;`
  (see section 5.10);
- In assertions, possibly about values of registers or memory cells, e.g.,
  `assert instruction` *P* `is entered with` *Expr*$_1$`, ...,` *Expr*$_n$`;`
  (see section 5.11);

- In specifications of the execution time of code snippets excluded from analysis, e.g.,
  `snippet` *P* `is not analyzed and takes` *Expr*`;`
  (see section 5.15.3);
- In specifications of additional execution time, e.g.,
  `instruction` *P* `additionally takes` *Expr*`;`
  (see section 5.13);
- In declarations of conditional infeasibility, e.g.,
  `snippet` *S* `is not executed if` *Expr*`;`
  (see section 5.17.2);
- In declarations of the values of conditions, e.g.,
  `condition` *P* `is true if` *Expr*`;`
  `condition` *P* `is exactly` *Expr*`;`
  (see section 5.18.2);
- In non-default loop bound specifications, e.g.,
  `loop` *L* `min` $Expr_1$ `max` $Expr_2$`;`
  (see section 5.20.1 and the examples in section 5.20.2).

5.28.1 Priorities and Grouping

The following subsections describe the various kinds of expressions in detail. Before that, we list the available operators with their precedence as a quick overview.

Subexpressions can be grouped by parentheses (...) as usual in mathematics and programming languages. The operands of unary operators such as `address` and `sqrt` must be put in parentheses. If no parentheses are used, subexpressions are grouped according to operator precedences. In the following, the operators are listed in order of decreasing precedence (binding power), where all operators in a row have the same precedence.

- Prefix operators with one or several arguments in parentheses: `address` (section 5.28.4); `mem` (5.28.5); `min`, `max`, `join`, `restrict`, `low`, `high`, `bounded`, `exact`, `try`, `abs` (5.28.6); `sqr` (5.28.7); `sqrt`, `log` (5.28.8); `floor`, `ceil`, `int` (5.28.9); `frac`, `uint`, `sint` (5.28.10); `!` (not), `if`, `always` (5.28.12).
- Exponentiation `**` (5.28.7);
- Multiplicative operators `*`, `/` (5.28.7), and modulo `%` (5.28.10);
- Additive operators + and binary − (5.28.7);
- Relational operators <, <=, >, >=, ==, !=, <=>, and `in` (5.28.11);
- Logical "and" `&&` (5.28.12);
- Logical "or" `||` and "xor" `^^` (5.28.12);

- Conditional `- ? - : -` (5.28.12);
- The `before` operator (5.28.15);
- Size and time units as postfix operators e.g., `bytes` and `cycles` (5.28.16);
- The range operator `..` (5.28.6);
- The `with` operator used for specifying modulo information (5.28.10);
- The separators `=>` and `,` of `switch` expressions and the `,` in argument lists (5.28.13).

Because of their low precedence, the operators `..` and `with` require parentheses in most cases. See also the examples in section 5.28.16 for possible problems caused by the low precedence of size and time units.

5.28.2 Results of Expressions

In principle, the value of an expression is meant to be a fraction, and indeed, simple expressions such as $1+1$ or $1/2$ evaluate to a fraction. Possible values are also $-inf$ (minus-infinity) and inf (plus-infinity). In general however, the result of evaluating an expression is not a single such value, but a set of possible values. There are two reasons for this complication: First, the values of registers as determined by value analysis can be imported into expressions, but value analysis often can only determine a set of possible values instead of an exact value. Second, expressions can be used to specify the values of registers at certain program points, but often the exact value is not known or may vary, and only some set of possible values is known, e.g., the set of integers between 0 and 1000.

Neither value analysis nor AIS expressions produce arbitrary sets of values though. All result sets are non-empty and described by the combination of three kinds of constraints:

- a lower bound a, which may also be $-inf$ or inf;
- an upper bound b, which may also be $-inf$ or inf;
- optional modulo information saying that all elements x of the set satisfy $x \,\%\, p == m$ where p is a power of 2. This kind of modulo information was introduced to be able to express alignment properties.

If there is no modulo information, lower and upper bound together describe an interval $(a..b)$ standing for the set of all fractions x with $a \leq x \leq b$. The notation $(a..b)$ is not the standard mathematical notation for such intervals, which would be $[a,b]$. Yet $(a..b)$ is the way in which intervals can be specified in AIS expressions (section 5.28.6), e.g., $(1..2)$ is valid AIS syntax evaluating to the interval $(1..2)$ ($[1,2]$ in mathematical notation). A special case is the interval $(a..a)$ that contains a single value a; in this case, we identify the interval with its single element. This is also true in AIS; for instance, the expressions `1` and `(1..1)` are equivalent.

With modulo information, sets of the form $\{x \mid a \leq x \leq b \text{ and } x \,\%\, p = m\}$ can be described. We abbreviate this to $(a..b \textit{ with } x \,\%\, p == m)$, which is also valid AIS syntax for this set (see section 5.28.10 for the `with` operator). Examples are $(3..15 \textit{ with } x \,\%\, 4 == 3) = \{3,7,11,15\}$, $(1..20 \textit{ with } x \,\%\, 8 == 0) = \{8,16\}$, and $(2..3 \textit{ with } x \,\%\, 1/2 == 0) = \{2,2.5,3\}$ (p in $x \,\%\, p == m$ may be a negative power of 2). A special case of modulo information is $x \,\%\, 1 == 0$ which means restriction to integers. For instance, $(2..5 \textit{ with } x \,\%\, 1 == 0) = \{2,3,4,5\}$ and $(0..1 \textit{ with } x \,\%\, 1 == 0) = \{0,1\}$ which contains only 0 and 1, while $(0..1)$ also contains fractions such as $1/2$ and $2/3$.

Note that not all two-element sets can be results of expressions because of the restriction to powers of 2: although $\{a,b\} = (a..b \textit{ with } x \,\%\, (b-a) == a \,\%\, (b-a))$ holds mathematically whenever $a < b$, this can only be an expression result if $b-a$ is a power of 2. So $\{0,1\}$ and $\{1,3\}$ can be results, but not $\{0,3\}$. The restriction to powers of 2 was a pragmatic decision to simplify the implementation.

a^3 sometimes describes result sets in its output with a different syntax $[a..(+p)..b]$, which is closer to its internal representation and stands for $\{a, a+p, a+2p, \ldots, b\} = (a..b \textit{ with } x \,\%\, p == m)$ where $m = a \,\%\, p = b \,\%\, p$, for instance $(3..15 \textit{ with } x \,\%\, 4 == 3) = [3..(+4)..15]$ or $(0..1 \textit{ with } x \,\%\, 1 == 0) = \{0,1\} = [0..(+1)..1]$. This syntax however is *not* valid AIS syntax and we avoid it in this manual.

5.28.3 Atomic Expressions

Expressions are built from atomic expressions using various operators. Atomic expressions may be integers in various representations including decimal and hexadecimal (see section 5.30.3), floating point numbers, the names of processor registers and user registers. *Processor registers* are specified by name (`A0`–`A15` or `D0`–`D15`). *User registers* have arbitrary names starting with `@` (see section 5.10). Register names may be preceded by the optional keyword `reg`.

The results of evaluating integers and floating point numbers are obvious. The values of processor registers and user registers are determined by value analysis; they are integer intervals with optional modulo information. The values of user registers are also restricted to integers. They ultimately stem from defining annotations (described in section 5.10). If there are no such annotations, they evaluate to $(-\textit{inf}..\textit{inf})$.

Since register values are restricted to integers, thay are imported into AIS expressions with modulo information $x \,\%\, 1 == 0$ if they do not already come with their own modulo information.

There is also a special constant **`inf`** whose value is larger than any representable number. Therefore the interval `(-inf..inf)` contains all numbers. Thus, a previous definition of a user register can be withdrawn by setting it to `(-inf..inf)`.

The special variable `value` described in section 5.28.14 is also syntactically atomic.

5.28.4 Addresses of Program Points and Memory Areas

There are means to import the addresses of program points and memory areas into expressions. In the sequel, *`Point`* means a program point and *`Area`* an area description, which may be a pair of program points `from` *`Start`* `to` *`End`*, or *`Start`* `..` *`End`*, or an array or section name. Area

descriptions are presented in more detail in section 5.12.1 and program points in section 5.29.

- `address (`*`Point`*`)` evaluates to the address of the program point.
- `address (`*`Area`*`)` evaluates to the interval of addresses in the given area. If the extent of the area is not provided explicitly and cannot be derived from the symbol table information in the executable, only the start address of the area is considered.
- The interval can be restricted to the start address in any case by writing `start address (`*`Area`*`)` or `start (`*`Area`*`).`
- The interval can be restricted to the last address in the area by writing `last address (`*`Area`*`)` or `last (`*`Area`*`).`

The parentheses around *`Point`* or *`Area`* may be omitted if these consist of a single routine or section name, e.g., `address "prime"`.

5.28.5 Contents of Memory Cells

The contents of memory cells can be referenced by means of the following expressions:

- `mem (`*`Address`*`, `*`Width`*`)`
  yields the contents of the memory at the given *`Address`*, as far as it is known to value analysis. *`Width`* is an expression that may optionally contain a size unit such as `byte` (see section 5.30.4 for a complete list and section 5.28.16 for the usage of units in expressions). If no size unit is used, `byte` is assumed. Only the access widths supported by the actual hardware are supported by a^3.

  Examples: `mem (0x1000, 4)` `mem (0x1000, 4 bytes)`
- `mem (`*`Address`*`, `*`Unit`*`)`
  is similar, but the access width is given by the *`Unit`*, e.g., `byte`. All possibilities for *`Unit`* are listed in section 5.30.4.

  Example: `mem (0x1000, word)`
- `mem (`*`Address`*`)`
  is a form without explicit access width. In this case, the standard access width of the target processor is taken.

  Example: `mem (0x1000)`
- In any case, there may be a further argument `lbf` (low byte first) or `hbf` (high byte first) to specify the endianness of the memory. This is rarely needed since a^3 employs the standard endianness of the target architecture if no explicit endianness is specified.

  Examples:
  ```
  mem (0x1000, 4, hbf)
  mem (0x1000, word, lbf)
  mem (0x1000, hbf)
  ```

The part described as `Address` can be an arbitrary expression whose value is taken as the address of the access. Examples for `Address`:

- `0x9020` is an absolute address.
- *R* `+ 0x10` is an address relative to register *R* with offset `0x10` bytes. In particular, *R* may be the stack pointer so that you can refer to the contents of stack cells.
- `@ADR` is the address derived from the value of the user register `@ADR`.

5.28.6 Minimum, Maximum, and Interval Operations

We start the presentation of the possible operations in expressions by listing some operations involving minimum and maximum.

- **`min (`**E_1**`, ..., `**E_n**`)**
  is a valid expression if E_1, ..., E_n are valid expressions. (A sentence like this should be written for each operator, but will be omitted in the sequel.)

  If the subexpressions $E_1, \ldots, E_n$ evaluate to exact values, then the result of the entire expression `min (`E_1`,...,`E_n`)` is the minimum, i.e. the smallest of these values. On intervals, the `min` operator works by forming the minimum of the lower ends and of the upper ends:

  $$min((a_1..b_1),\ldots,(a_n..b_n)) \;=\; (min(a_1,\ldots,a_n)..min(b_1,\ldots,b_n))$$

  Since an exact value v is considered to be the same as the interval $(v..v)$, the definition for intervals subsumes the definition for exact values.

  Modulo information in the operands is preserved in the result as far as possible.

  Examples: $min(2,1,4,3) = 1$; $min((1..5),(2..3)) = (1..3)$;
  $min((2..4\ with\ x\,\%\,1 == 0),(1..5\ with\ x\,\%\,1 == 0)) = (1..4\ with\ x\,\%\,1 == 0)$;
  $min((2..4\ with\ x\,\%\,2 == 0),(1..5\ with\ x\,\%\,2 == 1)) = (1..4\ with\ x\,\%\,1 == 0)$

- **`max (`**E_1**`, ..., `**E_n**`)**
  is analogous to `min`, but uses maximum (greatest value) instead of minimum (smallest value). On intervals, it operates as follows:

  $$max((a_1..b_1),\ldots,(a_n..b_n)) \;=\; (max(a_1,\ldots,a_n)..max(b_1,\ldots,b_n))$$

  Examples: $max(2,1,4,3) = 4$; $max((1..5),(2..3)) = (2..5)$;
  $max((2..4\ with\ x\,\%\,1 == 0),(1..5\ with\ x\,\%\,1 == 0)) = (2..5\ with\ x\,\%\,1 == 0)$;
  $max((2..4\ with\ x\,\%\,2 == 0),(1..5\ with\ x\,\%\,4 == 1)) = (2..5\ with\ x\,\%\,1 == 0)$

- **`join (`**E_1**`, ..., `**E_n**`)**
  evaluates to the least representable set containing the result sets of all the subexpressions E_1, ..., E_n. The lower end of the result is the minimum of the lower ends of the operands, and the upper end of the result is the maximum of the upper ends of the operands. On proper intervals (not single values) without modulo information, the result is

  $$join((a_1..b_1),\ldots,(a_n..b_n)) \;=\; (min(a_1,\ldots,a_n)..max(b_1,\ldots,b_n))$$

Again modulo information is preserved as much as possible. If exact values are joined, the best possible modulo information consistent with all operands is generated.

Examples:
$join(3,4) = (3..4\ with\ x\,\%\,1 == 0) = \{3,4\}$ since both arguments are integers;
$join(1,9) = (1..9\ with\ x\,\%\,8 == 1) = \{1,9\}$ since $1\,\%\,8 = 9\,\%\,8 = 1$;
$join(1,7) = (1..7\ with\ x\,\%\,2 == 1) = \{1,3,5,7\}$
since $1\,\%\,2 = 7\,\%\,2 = 1$, but $1\,\%\,4 = 1 \neq 3 = 7\,\%\,4$;
$join(2,1,4,3) = (1..4\ with\ x\,\%\,1 == 0)$;
$join((1..5),(2..3)) = (1..5)$; $join((1..5),(0..3)) = (0..5)$;
$join((2..4\ with\ x\,\%\,2 == 0),(1..5\ with\ x\,\%\,4 == 1)) = (1..5\ with\ x\,\%\,1 == 0)$

- **`restrict (`**E_1**`,`**$\ldots$**`,`**E_n**`)`**
evaluates to the intersection of the result sets of the subexpressions $E_1, \ldots, E_n$. If there is no modulo information, the lower end of the result is the maximum of the lower ends of the operands, and the upper end of the result is the minimum of the upper ends of the operands. Modulo information can cause the result to be smaller. If the intersection is empty, i.e. the lower end would be larger than the upper end or the modulo information is inconsistent, the result is $(-inf..inf)$, i.e. contains all possible numbers.

$restrict((a_1..b_1),\ldots,(a_n..b_n)) = (max(a_1,\ldots,a_n)..min(b_1,\ldots,b_n))$
where $(a..b) = (-inf..inf)$ if $a > b$

Examples: $restrict((1..5),(2..4)) = (2..4)$; $restrict((1..5),(0..4)) = (1..4)$;
$restrict((1..3),(3..5)) = 3$; $restrict((1..3),(4..5)) = (-inf..inf)$;
$restrict((1..5\ with\ x\,\%\,2 == 1),(2..4\ with\ x\,\%\,1 == 0)) = 3$;
$restrict((1..5\ with\ x\,\%\,2 == 1),(2..4\ with\ x\,\%\,2 == 0)) = (-inf..inf)$

- **`low (`**E**`)`** and **`high (`**E**`)`**
extract the lowest value and the highest value respectively from the result set produced by E. For intervals, this means

$low(a..b) = a$ $\quad$ $high(a..b) = b$

An exact value v produced by E is considered as singleton set $\{v\}$, so `low` and `high` reproduce the value v in this case.

Examples: $low(5) = 5$; $high(5) = 5$;
$low(3..7) = 3$; $low(3..7\ with\ x\,\%\,4 == 3) = 3$;
$high(3..7) = 7$; $high(3..7\ with\ x\,\%\,4 == 3) = 7$;
$low(high(3..7)) = 7$

- **`(`**E_1 **`..`** E_2**`)`**
is the AIS syntax for an interval. If E_1 evaluates to an exact value v_1 and E_2 to an exact value v_2, then the result is the interval $(v_1..v_2)$ if $v_1 \leq v_2$, and the error value $(-inf..inf)$ otherwise. The general case in which the two operands evaluate to sets is reduced to the case of exact values by taking the lowest value from the first set and the highest from the second:

$$(V_1..V_2) = \begin{cases} (low(V_1)..high(V_2)) & \text{if } low(V_1) \leq high(V_2) \\ (-inf..inf) & \text{otherwise} \end{cases}$$

Examples: $(1..2) = (1..2)$; $(2..1) = (-inf..inf)$;
$((0..4)..(1..5)) = (0..5)$; $((1..5)..(0..4)) = (1..4)$; $((4..5)..(0..2)) = (-inf..inf)$;
$((0..4\ with\ x\,\%\,4 == 0)..(1..5\ with\ x\,\%\,4 == 1)) = (0..5)$;
$((1..5\ with\ x\,\%\,4 == 1)..(0..4\ with\ x\,\%\,4 == 0)) = (1..4)$

- **`bounded (`***E***`)`**
  yields 1 if the expression E yields a bounded set, and 0 otherwise. A set is bounded if both its lowest element and its highest element are finite (not $-inf$ or inf). The modulo information of the set does not matter for this test.

  A user register `@X` that has not been given a value yields an unbounded interval $(-inf..inf)$. Therefore, `bounded (@X)` is 0 in this case.

- **`exact (`***E***`)`**
  yields 1 if the expression E yields an exact value, and 0 otherwise.

  Examples: $exact(5) = 1$; $exact(3..7) = 0$; $exact(low(3..7)) = 1$

- **`try (`***E*$_1$**`, `***E*$_2$**`)`**
  is E_1 if `bounded` E_1, and E_2 otherwise. Thus E_2 is a kind of default value that is taken if E_1 does not evaluate to a bounded interval. Section 5.20.2 contains some examples for this operator.

- **`try (`***E*$_1$**`, ..., `***E*$_n$**`)`**
  is alternative syntax for `try (`E_1`, try (`E_2`, ..., `E_n`) )` .

- **`abs (`***E***`)`**
  computes the absolute value of E. This is straightforward for exact values, but a bit more tricky for general sets of values. For intervals without modulo information we have:

$$abs(a..b) = \left\{ \begin{array}{ll} (a..b) & \text{if } 0 \le a \\ (-b..-a) & \text{if } b \le 0 \\ (0..max(-a,b)) & \text{if } a < 0 < b \end{array} \right\} = (max(0,a,-b)..max(-a,b))$$

  Examples: $abs(2) = 2$; $abs(-2) = 2$; $abs(2..4) = (2..4)$; $abs(-4..-2) = (2..4)$;
  $abs(-1..3) = (0..3)$; $abs(-3..1) = (0..3)$

  Modulo information from the argument is taken over into the result as far as possible. The result is straightforward if the argument set is all-positive or all-negative:

  $abs(1..5\ with\ x\,\%\,4 == 1) = (1..5\ with\ x\,\%\,4 == 1)$
  $abs(-5..-1\ with\ x\,\%\,4 == 3) = (1..5\ with\ x\,\%\,4 == 1)$

  If some values are negative and some are positive, more complicated things can happen:

  $abs(-16..8\ with\ x\,\%\,8 == 0) = abs\{-16,-8,0,8\} = \{0,8,16\} = (0..16\ with\ x\,\%\,8 == 0)$
  $abs(-12..4\ with\ x\,\%\,8 == 4) = abs\{-12,-4,4\} = \{4,12\} = (4..12\ with\ x\,\%\,8 == 4)$
  $abs(-6..10\ with\ x\,\%\,8 == 2) = abs\{-6,2,10\} = \{2,6,10\} = (2..10\ with\ x\,\%\,4 == 2)$
  Ideally, $abs(-7..1\ with\ x\,\%\,8 == 1) = abs\{-7,1\} = \{1,7\}$, but this set is not representable with power-2-modulo information, so the result is $(1..7\ with\ x\,\%\,2 == 1) = \{1,3,5,7\}$.

5.28.7 Rational Arithmetic

While the effect of the arithmetical operations on exact values is more or less clear, care must be taken when they are applied to arguments producing non-trivial sets of values. The general idea is to apply the operation to all elements of the sets and then to take as result the least representable set containing all possible result values. Of course, this is not an effective way to implement the operations. Instead, methods must be found to compute the bounds and the modulo information of the result from the bounds and the modulo information of the arguments.

- E_1 **+** E_2
  sums the results of E_1 and E_2. On intervals, this is done by summing the respective ends:
  $(a_1..b_1) + (a_2..b_2) = (a_1+b_1 .. a_2+b_2)$
  The modulo information is in principle handled by taking over the smaller power of two and adding the two remainders (modulo the new power of two).
  Examples: $1+2=3$; $\frac{1}{2}+\frac{1}{3}=\frac{5}{6}$;
  $2+(4..5)=(6..7)$; $(1..4)+(2..5)=(3..9)$;
  $(1..9 \textit{ with } x\%4==1)+(2..18 \textit{ with } x\%8==2) = (3..27 \textit{ with } x\%4==3)$;
  $(1..9 \textit{ with } x\%4==1)+(5..21 \textit{ with } x\%8==5) = (6..30 \textit{ with } x\%4==2)$

- **−** E
  negates the result of E, e.g., maps 2 to -2 and -2 to 2. The effect on intervals is as follows:
  $-(a..b) = (-b.. -a)$
  In the modulo information, the power of two is unchanged; only the remainder is adapted.
  Examples: $-(1..2)=(-2..-1)$; $-(-2..4)=(-4..2)$;
  $-(1..9 \textit{ with } x\%4==1) = (-9..-1 \textit{ with } x\%4==3)$

- E_1 **−** E_2
  corresponds to $E_1+(-E_2)$. On intervals, this means
  $(a_1..b_1) - (a_2..b_2) = (a_1-b_2 .. a_2-b_1)$
  Examples: $3-2=1$; $2-3=-1$; $4-4=0$; $\frac{1}{2}-\frac{1}{3}=\frac{1}{6}$;
  $(4..5)-2=(2..3)$; $(4..5)-(1..2)=(2..4)$; $(3..5)-(3..5)=(-2..2)$;
  $(2..26 \textit{ with } x\%8==2)-(1..13 \textit{ with } x\%4==1) = (-11..25 \textit{ with } x\%4==1)$

- E_1 ***** E_2
  multiplies the results of E_1 and E_2. When computing the product of two intervals, one has to take into account that multiplication with a negative number turns the order around. Thus it is not sufficient to multiply corresponding ends.
  Examples: $2*3=6$; $-2*-3=6$; $\frac{1}{2}*\frac{1}{3}=\frac{1}{6}$;
  $2*(3..4)=(6..8)$; $-2*(3..4)=(-8..-6)$;
  $(2..3)*(4..5)=(8..15)$; $(-2..3)*(-4..5)=(-12..15)$;
  $4*(1..9 \textit{ with } x\%2==1) = (4..36 \textit{ with } x\%8==4)$;
  $3*(1..9 \textit{ with } x\%2==1) = (3..27 \textit{ with } x\%2==1)$;
  $1/3*(1..9 \textit{ with } x\%2==1) = (1/3..3)$;

In the last example, the modulo information disappears entirely since the step width $2/3$ is not an integer multiple of any power of 2.

- E_1 **`/`** E_2
  divides E_1 by E_2. Division by 0 does not result in a specific value, so the result is $(-inf..inf)$ if the result interval of E_2 contains 0.

  Examples: $12/4 = 3$; $3/4 = \frac{3}{4}$; $(-6..8)/2 = (-3..4)$;
  $1/(2..3) = (\frac{1}{3}..\frac{1}{2})$; $1/(-1..1) = (-inf..inf)$;
  $(8..10)/(2..5) = (\frac{8}{5}..5)$; $(-8..10)/(2..5) = (-4..5)$; $(2..4)/(2..4) = (\frac{1}{2}..2)$;
  $(1..5\ with\ x\,\%\,2 == 1)/2 = (1/2..5/2\ with\ x\,\%\,1 == 1/2)$;
  $(1..5\ with\ x\,\%\,2 == 1)/3 = (1/3..5/3)$; $1/(1..5\ with\ x\,\%\,2 == 1) = (1/5..1)$

  The last two examples show that division destroys modulo information in many cases.

  Although `/` is not integer division, there is a modulo operator `%` presented in section 5.28.10.

- **`sqr (`**E**`)`**
  computes the square of E. It is defined for intervals as follows:

$$sqr(a..b) = \begin{cases} (a^2..b^2) & \text{if } 0 \le a \\ (b^2..a^2) & \text{if } b \le 0 \\ (0..max(a^2,b^2)) & \text{if } a < 0 < b \end{cases}$$

  Examples: $sqr(2) = 4$; $sqr(-2) = 4$; $sqr(2..4) = (4..16)$; $sqr(-4..-2) = (4..16)$;
  $sqr(-1..3) = (0..9)$; $sqr(-3..1) = (0..9)$;
  $sqr(2..8\ with\ x\,\%\,2 == 0) = (4..64\ with\ x\,\%\,4 == 0)$;
  $sqr(3..9\ with\ x\,\%\,2 == 1) = (9..81\ with\ x\,\%\,8 == 1)$;
  $sqr(-3..3\ with\ x\,\%\,2 == 1) = (1..9\ with\ x\,\%\,8 == 1)$

  For exact values v, $sqr(v)$ is simply the same as $v * v$. This is not true for intervals or for general sets; $sqr(a..b)$ is in general smaller than $(a..b) * (a..b)$ since in the latter numbers drawn from the two operand intervals may vary independently.

  Example: $sqr(-3..1) = (0..9)$, but $(-3..1) * (-3..1) = (-3..9)$.

The above specifications are idealized because they abstract from overflow. If the sum or product of two big positive numbers gets too large to be represented, the result is undefined; there is no wrap-around to negative numbers as in most processor arithmetics.

Since all operations are implemented via arithmetic on fractions, an overflow in the denominator may happen even if the result is not too large. In such a case, the result x is replaced by a tiny interval containing x whose end points have representable denominators.

5.28.8 Irrational Operations

AIS expressions also offer a few operations that go beyond the field of rational numbers (fractions). This is possible at the expense of precision: genuine irrational results are replaced by tiny rational intervals containing the exact result.

Modulo information is generally destroyed by irrational operations; even in the few cases where modulo information in the result would be possible mathematically, the result is an interval without modulo information.

- **`sqrt (`**E**`)`**
  computes the square root of E. The square root of a negative number is undefined ($(-inf..inf)$). On intervals, `sqrt` operates as follows:

  $$sqrt(a..b) = \begin{cases} (sqrt^-(a)..sqrt^+(b)) & \text{if } 0 \le a \\ (-inf..inf) & \text{if } a < 0 \end{cases}$$

  Here $sqrt^-(a)$ is $\sqrt{a}$ if this is rational, and a rational number slightly below $\sqrt{a}$ if this is irrational. Similarly, $sqrt^+(b)$ is a rational number slightly above $\sqrt{b}$ if this is irrational.

  Examples: $sqrt(9) = 3$; $sqrt(0) = 0$; $sqrt(-1) = (-inf..inf)$; $sqrt(\frac{1}{9}..9) = (\frac{1}{3}..3)$;
  $sqrt(2)$ = tiny interval containing $\sqrt{2}$;
  $sqr(sqrt(2))$ = tiny interval containing 2;
  $exact(sqrt(9)) = 1$; $exact(sqrt(2)) = 0$;
  $sqrt(0..1 \text{ with } x\,\%\,1 == 0) = (0..1)$

- E_1 **`**`** E_2
  computes E_1 to the power of E_2. The base of the power must not be negative, and base 0 is only allowed if the exponent is positive. For exact values $v_1 > 0$ and v_2, $v_1 ** v_2$ is in general a tiny interval containing the exact result $v_1^{v_2}$. Of course, this extension to a tiny rational interval cannot be avoided if $v_1^{v_2}$ is irrational, but in contrast to square root, the extension may even happen for rational results; there is no guarantee here that a rational result is exactly reproduced.

  Examples: $2 ** 3 = 8$; $3 ** 2 = 9$;
  $3 ** (-2) = \frac{1}{9}$; $(2..3) ** (3..4) = (8..81)$;
  $(0..1 \text{ with } x\,\%\,1 == 0) ** 2 = (0..1)$,
  but $sqr(0..1 \text{ with } x\,\%\,1 == 0) = (0..1 \text{ with } x\,\%\,1 == 0)$ (more precision)
  $9 ** \frac{1}{2} = 3$ (exact)
  $27 ** \frac{1}{3}$ = tiny interval containing 3 (not exact)
  $81 ** \frac{1}{4} = 3$ (exact)

- **`log (`**E_1**`,`** E_2**`)`**
  computes the logarithm of E_1 to the base E_2. For two exact values $v_1 > 0$ and $v_2 > 0$, $v_2 \neq 1$, $log(v_1, v_2)$ is in general a tiny interval containing the exact result $log_{v_2} v_1$. Rational logarithms may be exactly reproduced.

  Examples: $log(1,2) = 0$; $log(2,2) = 1$; $log(4,2) = 2$;
  $log(81,3) = 4$; $log(81, \frac{1}{3}) = -4$; $log(1000, 10) = 3$;
  $log(3,2)$ = tiny interval around $log_2 3$

5.28.9 Rounding Operations

In this section, we describe various operations for rounding to integers. All these operations preserve modulo information as much as possible, and they are able to create new modulo information: even if the argument set does not contain any modulo information, the result set is known to consist only of integers and thus to satisfy at least the modulo information $x \,\%\, 1 == 0$.

- **`floor (`**E**`)`**
  rounds the value of E down to the nearest integer below it. The effect on a set of values is to round down each element of the set. For intervals without modulo information, the result is therefore

  $floor(a..b) = (floor(a)\,..\,floor(b)\ with\ x \,\%\, 1 == 0)$

  Examples: $floor(2) = 2$; $floor(4.2) = 4$; $floor(-4.2) = -5$; $floor(1.4..1.7) = 1$;
  $floor(1.2..3.7) = (1..3\ with\ x \,\%\, 1 == 0)$;
  $floor(-2.3..8.2) = (-3..8\ with\ x \,\%\, 1 == 0)$;
  $floor(1.2..5.2\ with\ x \,\%\, 2 == 1.2) = (1..5\ with\ x \,\%\, 2 == 1)$

- **`ceil (`**E**`)`**
  rounds the value of E up to the nearest integer above it. The effect on a set of values is to round up each element of the set. For intervals without modulo information, the result is therefore

  $ceil(a..b) = (ceil(a)\,..\,ceil(b)\ with\ x \,\%\, 1 == 0)$

  Examples: $ceil(2) = 2$; $ceil(4.2) = 5$; $ceil(-4.2) = -4$; $ceil(1.4..1.7) = 2$;
  $ceil(1.2..3.7) = (2..4\ with\ x \,\%\, 1 == 0)$;
  $ceil(-2.3..8.2) = (-2..9\ with\ x \,\%\, 1 == 0)$;
  $ceil(1.2..5.2\ with\ x \,\%\, 2 == 1.2) = (2..6\ with\ x \,\%\, 2 == 0)$

- **`int (`**E**`)`**
  applied to an exact value x yields $\{floor(x), ceil(x)\}$. Thus x is reproduced identically if it is an integer and replaced by the two integers around it if not. The result for a set of values is the union of the results for the individual values in the set. For intervals without modulo information, the result is therefore

  $int(a..b) = (floor(a)\,..\,ceil(b)\ with\ x \,\%\, 1 == 0)$

  Examples: $int(2) = 2$; $int(4.2) = (4..5\ with\ x \,\%\, 1 == 0)$;
  $int(-4.2) = (-5..-4\ with\ x \,\%\, 1 == 0)$; $int(1.4..1.7) = (1..2\ with\ x \,\%\, 1 == 0)$;
  $int(1.2..3.7) = (1..4\ with\ x \,\%\, 1 == 0)$;
  $int(-2.3..8.2) = (-3..9\ with\ x \,\%\, 1 == 0)$;
  $int(1.2..5.2\ with\ x \,\%\, 2 == 1.2) = (1..6\ with\ x \,\%\, 1 == 0)$;
  $int(sqr(sqrt(9))) = 9$;
  $int(sqr(sqrt(2))) = (1..3\ with\ x \,\%\, 1 == 0)$

  The explanation for the last example is that $sqrt(2)$ yields a tiny interval around $\sqrt{2}$, and so $sqr(sqrt(2))$ yields a tiny interval around 2. Its left end point is slightly below 2 and thus gives 1 when rounded down, and its right end point is slightly above 2 and thus gives 3 when rounded up. This does not happen in the second but last example since $sqrt(9)$ is exactly 3.

5.28.10 Modulo Operations

In this section, we describe the *with* operation for specifying modulo information, the % operation for computing it, and the related operations of extracting the fractional part and of casting to a given bit width (unsigned and signed).

- **`(`** *E* **`with x % `** *A* **`==`** *R* **`)`**
  restricts the result set of E so that (ideally) only those elements x with $x\%A = R$ are retained. This works properly only if A is an exact power of two since only modulo information w.r.t. powers of two can be handled by the present implementation.

  The above is special syntax; the "identifier" `x` must be `x`; there is no other choice here. Likewise, `%` is the only operator allowed after `x`, and `==` is the only relational operator allowed here; you cannot use `<=` or `!=` or the like. On the other hand, the parts A and R are not restricted to constants, but may be expressions – arbitrary expressions in case of R and atomic expressions in case of A (numeric constants, register names, or anything in parentheses).

  Examples:

  If the modulo constraint involves a power of two and E yields a pure interval whose end points satisfy the modulo constraint, then the `with` expression directly corresponds to the resulting set of values:
  $(2..8\ with\ x\%2 == 0) \ = \ (2..8\ with\ x\%2 == 0)$

  If the end points of the interval do not satisfy the modulo constraint, the interval is shrunk until the end points do satisfy it:
  $(1..9\ with\ x\%2 == 0) \ = \ (2..8\ with\ x\%2 == 0)$

  Sometimes this shrinking leads to an exact value:
  $(1..9\ with\ x\%8 == 0) \ = \ 8$

  ... or to no value at all. As usual in AIS, the empty set is replaced by the set of all values:
  $(1..7\ with\ x\%8 == 0) \ = \ (-inf..inf)$

  An unbounded interval cannot carry modulo information in the current implementation:
  $(0..inf\ with\ x\%8 == 0) \ = \ (0..inf)$

  A modulo constraint with a denominator d that is not a power of two is not ignored completely, but replaced by a weaker constraint whose denominator is the largest power of 2 contained in d:
  $(5..29\ with\ x\%12 == 5) \ = \ (5..29\ with\ x\%4 == 1)$

  If the result set of E already has a modulo constraint, this constraint is combined with the new one if they are compatible:
  $((5..21\ with\ x\%8 == 5)\ with\ x\%2 == 1) \ = \ (5..21\ with\ x\%8 == 5)$
  $((5..21\ with\ x\%2 == 1)\ with\ x\%8 == 5) \ = \ (5..21\ with\ x\%8 == 5)$

  If the constraints are incompatible, the result set would be the empty set, which is replaced by the set of all values:
  $((5..21\ with\ x\%2 == 1)\ with\ x\%8 == 2) \ = \ (-inf..inf)$

- E_1 `%` E_2
  computes E_1 modulo E_2, the remainder of division of E_1 by E_2. For two exact values v_1 and v_2, $v_1 \,\%\, v_2$ has the same sign as the divisor v_2. For $v_2 > 0$, $0 \leq (v_1 \,\%\, v_2) < v_2$ holds, and for $v_2 < 0$, $v_2 < (v_1 \,\%\, v_2) \leq 0$. For $v_2 = 0$, $v_1 \,\%\, v_2$ is still defined and equals 0.

  Examples: $6\,\%\,3 = 0$; $7\,\%\,3 = 1$; $7.24\,\%\,3 = 1.24$; $8\,\%\,3 = 2$;
  $8.88\,\%\,3 = 2.88$; $9\,\%\,3 = 0$; $-1\,\%\,3 = 2$; $-2\,\%\,3 = 1$; $-3\,\%\,3 = 0$;
  $6\,\%\,{-3} = 0$; $7\,\%\,{-3} = -2$; $8\,\%\,{-3} = -1$; $8.88\,\%\,{-3} = -0.12$;
  $9\,\%\,{-3} = 0$; $-1\,\%\,{-3} = -1$; $-2\,\%\,{-3} = -2$; $-3\,\%\,{-3} = 0$;
  $3\,\%\,0.7 = 0.2$; $3\,\%\,0 = 0$; $0\,\%\,0 = 0$

  The behavior on intervals without modulo information is quite complicated and depends on whether the modulo wraps around.

  Examples: $(31\,..\,37)\,\%\,10 = (1\,..\,7)$; $(33\,..\,40)\,\%\,10 = (0\,..\,10)$;
  $5\,\%\,(2\,..\,3) = (0\,..\,3)$; $35\,\%\,(10\,..\,11) = (2\,..\,5)$

  Modulo information in the left argument is of course taken into account.

  Examples:
  $(1\,..\,11 \mathit{\ with\ } x\,\%\,2 == 1)\,\%\,2 = 1$;
  $(1\,..\,11 \mathit{\ with\ } x\,\%\,2 == 1)\,\%\,1 = 0$;
  $(1\,..\,11 \mathit{\ with\ } x\,\%\,2 == 1)\,\%\,4 = (1\,..\,3 \mathit{\ with\ } x\,\%\,2 == 1)$;
  $(1\,..\,11 \mathit{\ with\ } x\,\%\,2 == 1)\,\%\,3 = (0\,..\,2 \mathit{\ with\ } x\,\%\,1 == 0)$

- **`frac (`**E**`)`**
  computes the fractional part of the result of E, which is the same as $E \,\%\, 1$; hence $0 \leq \mathit{frac}\,(v) < 1$ always holds. For an exact value v, $\mathit{frac}\,(v)$ equals $v - \mathit{floor}\,(v)$. For intervals without modulo information, one has to distinguish whether the floors of the two end points are equal or not. If not, the fractional part wraps around from 1 to 0 within the interval, and so the result is $(0..1)$.

  $$\mathit{frac}\,(a\,..\,b) = \begin{cases} (\mathit{frac}\,(a)\,..\,\mathit{frac}\,(b)) & \text{if } \mathit{floor}\,(a) = \mathit{floor}\,(b) \\ (0\,..\,1) & \text{if } \mathit{floor}\,(a) < \mathit{floor}\,(b) \end{cases}$$

  Examples: $\mathit{frac}\,(2) = 0$; $\mathit{frac}\,(4.2) = 0.2$; $\mathit{frac}\,(-4.2) = 0.8$;
  $\mathit{frac}\,(1\,..\,1.3) = (0\,..\,0.3)$; $\mathit{frac}\,(1.4\,..\,1.7) = (0.4\,..\,0.7)$; $\mathit{frac}\,(1.6\,..\,2) = (0\,..\,1)$;
  $\mathit{frac}\,(-4.7\,..\,-4.2) = (0.3\,..\,0.8)$; $\mathit{frac}\,(-5\,..\,-4.2) = (0\,..\,0.8)$;
  $\mathit{frac}\,(-4.7\,..\,-4) = (0\,..\,1)$;
  $\mathit{frac}\,(1\,..\,10 \mathit{\ with\ } x\,\%\,0.5 == 0) = (0\,..\,0.5 \mathit{\ with\ } x\,\%\,0.5 == 0)$;
  $\mathit{frac}\,(1\,..\,10 \mathit{\ with\ } x\,\%\,1 == 0) = 0$;
  $\mathit{frac}\,(0.85\,..\,1.6 \mathit{\ with\ } x\,\%\,0.25 == 0.1) = (0.1\,..\,0.85 \mathit{\ with\ } x\,\%\,0.25 == 0.1)$

- **`uint (bits `**B**`,`** E**`)`**
  casts its argument to an *unsigned integer* of bit width B. Here E is an arbitrary expression and B is an atomic expression (a constant, a register, or an arbitrary expression in parentheses). The cast is performed by transforming the result set of E into a set of integers by applying the `int` operator from section 5.28.9 and then the modulo-2^B operation. Since the result set of `uint` consists of integers, it always carries modulo information, at least $x\,\%\,1 == 0$.

  Examples:

$uint\,(bits\,1, 10) = 0$; $uint\,(bits\,2, 10) = 2$;
$uint\,(bits\,3, 10) = 2$; $uint\,(bits\,4, 10) = 10$;
$uint\,(bits\,8, 3.6) = (3..4\ with\ x\,\%\,1 == 0)$; (`int` produces the two adjacent integers)
$uint\,(bits\,8, (300..400)) = (44..144\ with\ x\,\%\,1 == 0)$;
$uint\,(bits\,8, (500..600)) = (0..255\ with\ x\,\%\,1 == 0)$;
$uint\,(bits\,8, (500..600\ with\ x\,\%\,4 == 0)) = (0..252\ with\ x\,\%\,4 == 0)$

- **`sint (bits` B`,` E`)`**
  casts its argument to a *signed integer* of bit width B. Here E is an arbitrary expression and B is an atomic expression (a constant, a register, or an arbitrary expression in parentheses). The cast is performed by transforming the result set of E into a set of integers by applying the `int` operator from section 5.28.9, then the modulo-2^B operation, and finally sign adjustment if the highest of the B bits is set. Since the result set of `sint` consists of integers, it always carries modulo information, at least $x\,\%\,1 == 0$.

  Examples:
  $sint\,(bits\,1, 10) = 0$; $sint\,(bits\,2, 10) = -2$;
  $sint\,(bits\,3, 10) = 2$; $sint\,(bits\,4, 10) = -6$; $sint\,(bits\,5, 10) = 10$;
  $sint\,(bits\,8, 3.6) = (3..4\ with\ x\,\%\,1 == 0)$; (`int` produces the two adjacent integers)
  $sint\,(bits\,8, (300..350)) = (44..94\ with\ x\,\%\,1 == 0)$;
  $sint\,(bits\,8, (300..400)) = (-128..127\ with\ x\,\%\,1 == 0)$;
  $sint\,(bits\,8, (500..600)) = (-12..88\ with\ x\,\%\,1 == 0)$;
  $sint\,(bits\,8, (500..600\ with\ x\,\%\,4 == 0)) = (-12..88\ with\ x\,\%\,4 == 0)$

5.28.11 Relational Operators

All relational operators except for <=> yield only three possible results:

- The exact value 1, to be interpreted as “true”.
- The exact value 0, to be interpreted as “false”.
- The result set $(0..1\ with\ x\,\%\,1 == 0) = \{0,1\}$ containing both 0 and 1, to be interpreted as “ambiguous” (not “true” and not “false”).

The following operators exist:

- E_1 **`<`** E_2
  operates on exact values as expected. When applied to two sets of values, the result is only 1 if the $<$ relation holds for all pairs of elements drawn from the two sets, and 0 only if the negation of $<$ holds for all pairs of elements. More formally, this means

$$S_1 < S_2 = \begin{cases} 1 & \text{if } v_1 < v_2 \text{ holds for all } v_1 \text{ in } S_1 \text{ and } v_2 \text{ in } S_2 \\ 0 & \text{if } v_1 \not< v_2 \text{ holds for all } v_1 \text{ in } S_1 \text{ and } v_2 \text{ in } S_2 \\ \{0,1\} & \text{otherwise} \end{cases}$$

The two conditions given above illustrate the idea behind the definition. The actual implication is based on two simpler, but logically equivalent conditions involving the lowest and highest elements of the two sets:

$$S_1 < S_2 \;=\; \begin{cases} 1 & \text{if } \mathit{high}(S_1) < \mathit{low}(S_2) \\ 0 & \text{if } \mathit{low}(S_1) \geq \mathit{high}(S_2) \\ \{0,1\} & \text{otherwise} \end{cases}$$

Since only the lowest and highest elements are considered, modulo information in the arguments does not matter.

Examples: $(1 < 4) \;=\; 1$; $(1 < 1) \;=\; 0$; $((1..4) < (6..9)) \;=\; 1$;
$((1..4) < (4..9)) \;=\; \{0,1\}$; $((1..4) < (0..9)) \;=\; \{0,1\}$;
$((1..4) < (0..2)) \;=\; \{0,1\}$; $((1..4) < (0..1)) \;=\; 0$

- E_1 **<=** E_2
  is similar to <. We immediately present the simple conditions:

$$S_1 \leq S_2 \;=\; \begin{cases} 1 & \text{if } \mathit{high}(S_1) \leq \mathit{low}(S_2) \\ 0 & \text{if } \mathit{low}(S_1) > \mathit{high}(S_2) \\ \{0,1\} & \text{otherwise} \end{cases}$$

  Examples: $(1 \leq 4) \;=\; 1$; $(1 \leq 1) \;=\; 1$; $(1 \leq 0) \;=\; 0$;
  $((1..4) \leq (6..9)) \;=\; 1$; $((1..4) \leq (4..9)) \;=\; 1$;
  $((1..4) \leq (0..9)) \;=\; \{0,1\}$; $((1..4) \leq (0..1)) \;=\; \{0,1\}$;
  $((1..4) \leq (0..0.5)) \;=\; 0$

- E_1 **>** E_2 is equivalent to E_2 **<** E_1.

- E_1 **>=** E_2 is equivalent to E_2 **<=** E_1.

- E_1 **==** E_2
  works in the same spirit as the operators listed above. When applied to two sets, the result is only 1 if each member of the first set equals each member of the second, and 0 only if each member of the first set is different from each member of the second:

$$S_1 == S_2 \;=\; \begin{cases} 1 & \text{if } v_1 = v_2 \text{ for all } v_1 \text{ in } S_1 \text{ and } v_2 \text{ in } S_2 \\ 0 & \text{if } v_1 \neq v_2 \text{ for all } v_1 \text{ in } S_1 \text{ and } v_2 \text{ in } S_2 \\ \{0,1\} & \text{otherwise} \end{cases}$$

  The two conditions can be simplified as follows:

$$S_1 == S_2 \;=\; \begin{cases} 1 & \text{if } S_1 = S_2 = \{v\} \text{ for some } v, \\ & \text{i.e. both arguments are exact values and identical} \\ 0 & \text{if } S_1 \text{ and } S_2 \text{ are disjoint, i.e. have no value in common} \\ \{0,1\} & \text{otherwise} \end{cases}$$

  Examples: $(1 == 4) \;=\; 0$; $(1 == 1) \;=\; 1$; $((1..2) == (1..2)) \;=\; \{0,1\}$;
  $((1..2) == (2..4)) \;=\; \{0,1\}$; $((1..2) == (3..4)) \;=\; 0$;
  $((0..8 \;\mathit{with}\; x\,\%\,2 == 0) == (1..9 \;\mathit{with}\; x\,\%\,2 == 1)) \;=\; 0$

  Thus, == is *not* equality on result sets. It rather considers the result sets as sets of possible values, and only returns "true" if these values are equal no matter which possibility is chosen.

For instance, $R == 0$ where R is a register only returns "true" if value analysis can prove that the value of R is 0, i.e. if the set of possible values of R is exactly $\{0\}$, not some set containing 0.

Equality of the result sets of E_1 and E_2 can be checked by the operator `exactly ==` described below.

- E_1 **`!=`** E_2
  yields 1 iff $E_1 == E_2$ yields 0, and vice versa. Thus, the outcome is 1 iff the result sets of E_1 and E_2 are disjoint, and 0 iff the result sets are identical exact numbers.

- E_1 **`<=>`** E_2
  differs from the operators listed above in that it can also return -1 and sets involving -1. For two exact values v_1 and v_2, the result is

$$(v_1 <=> v_2) = \begin{cases} -1 & \text{if } v_1 < v_2 \\ 0 & \text{if } v_1 = v_2 \\ 1 & \text{if } v_1 > v_2 \end{cases}$$

  For sets, we have

$$(S_1 <=> S_2) = ((low(S_1) <=> high(S_2)) .. (high(S_1) <=> low(S_2)) \; with \; x \,\%\, 1 == 0)$$

  Since only the lowest and highest elements are considered, modulo information in the arguments does not matter.

  Examples: $(1 <=> 2) = -1$; $(2 <=> 2) = 0$; $(4 <=> 2) = 1$;
  $((0..2) <=> (3..5)) = -1$;
  $((0..2) <=> (2..5)) = (-1..0 \; with \; x \,\%\, 1 == 0) = \{-1, 0\}$;
  $((1..2) <=> (1..2)) = (-1..1 \; with \; x \,\%\, 1 == 0) = \{-1, 0, 1\}$;
  $((2..3) <=> (1..2)) = (0..1 \; with \; x \,\%\, 1 == 0) = \{0, 1\}$;
  $((3..5) <=> (1..2)) = 1$

- E_1 **`exactly==`** E_2
  differs from the operators presented above in that it provides a real set operation. The result of E_1 *exactly*$==$ E_2 is 1 if E_1 and E_2 have exactly the same result set (same bounds and same modulo information), and 0 otherwise. Thus the outcome of E_1 *exactly*$==$ E_2 is always an exact value.

  ⚠ Syntactic issue: The operator `exactly ==` binds tighter than ==, in fact tighter than the arithmetic operators. Thus its operands E_1 and E_2 should be put in parentheses unless they are atomic (constants or register names).

  Examples: $(1 \; exactly == 4) = 0$; $(1 \; exactly == 1) = 1$;
  $((1..2) \; exactly == (1..2)) = 1$; $((1..2) \; exactly == (0..2)) = 0$;
  $((1..2) \; exactly == (1..2 \; with \; x \,\%\, 1 == 0)) = 0$

- E_1 **`in`** E_2
  tests whether all values produced by E_1 are contained in the result set of E_2. Hence, if E_1 yields an exact value v_1 and E_2 a set S_2, then E_1 *in* E_2 is 1 if v_1 is contained in S_2, and 0 if not. If E_2 is also exact, this reduces to an equality check, but in general, it is different.

If E_1 yields a set S_1 and E_2 a set S_2, then E_1 *in* E_2 yields 1 if S_1 is a subset of S_2, 0 if S_1 and S_2 are disjoint, i.e. have no value in common, and $\{0,1\}$ otherwise.

Examples: $(1\ in\ 2) = 0$; $(1\ in\ 1) = 1$; $(1\ in\ 0) = 0$;
$(1\ in\ (3..7)) = 0$; $(3\ in\ (3..7)) = 1$; $(4\ in\ (3..7)) = 1$; $(5\ in\ (3..7)) = 1$;
$(3\ in\ (3..7\ with\ x\,\%\,2 == 1)) = 1$;
$(4\ in\ (3..7\ with\ x\,\%\,2 == 1)) = 0$;
$(5\ in\ (3..7\ with\ x\,\%\,2 == 1)) = 1$;
$((0..1)\ in\ (3..7)) = 0$; $((2..6)\ in\ (3..7)) = \{0,1\}$;
$((4..6)\ in\ (3..7)) = 1$; $((3..7)\ in\ (3..7)) = 1$;
$((3..7\ with\ x\,\%\,2 == 1)\ in\ (3..7)) = 1$;
$((3..7\ with\ x\,\%\,2 == 1)\ in\ (3..7\ with\ x\,\%\,2 == 1)) = 1$;
$((3..7)\ in\ (3..7\ with\ x\,\%\,2 == 1)) = \{0,1\}$;
$((4..6\ with\ x\,\%\,2 == 0)\ in\ (3..7\ with\ x\,\%\,2 == 1)) = 0$

A possible application of the `in` operator is to check whether all possible values of a register R that is used for addressing are addresses within a certain section, say `.text`:

```
R in address(section ".text")
```

5.28.12 Logical Operators

a^3 supports some logical operators, mainly for combining the results of relational operators (section 5.28.11). Except for <=>, these operators only yield the result sets $\{1\} = 1$, $\{0\} = 0$, and $\{0,1\}$.[1] The logical operators can be applied to other arguments as well, but perform a "logical interpretation" of their arguments first, which again yields only the three afore-mentioned sets.

Logical interpretation transforms any result set S to one of the three "logical" result sets $\{1\}$, $\{0\}$, and $\{0,1\}$ according to the following rules:

- The result is $\{1\}$ ("true") if the set S lies entirely to the left or entirely to the right of 0.
- The result is $\{0\}$ ("false") if the set S contains 0 and nothing else, i.e. $S = \{0\}$.
- The result is $\{0,1\}$ ("ambiguous") in all other cases.

For instance $(0.5..3)$ and $(-2..-1)$ become $\{1\}$ under the logical interpretation, while $(-1..0)$ and $(-1..1)$ become $\{0,1\}$.

- **!***E*
  applies the logical interpretation to the result of E and then negates the outcome, i.e. replaces 1 by 0 and vice versa.

E	0	$\{0,1\}$	1
$!E$	1	$\{0,1\}$	0

[1] Recall that we identify singleton sets with their element.

- E_1 `&&` E_2
  applies the logical interpretation to the results of E_1 and E_2 and then computes the logical "and" of the outcomes.

&&	0	{0,1}	1
0	0	0	0
{0,1}	0	{0,1}	{0,1}
1	0	{0,1}	1

- E_1 `||` E_2
  applies the logical interpretation to the results of E_1 and E_2 and then computes the logical "or" of the outcomes.

||	0	{0,1}	1
0	0	{0,1}	1
{0,1}	{0,1}	{0,1}	1
1	1	1	1

- E_1 `^^` E_2
  applies the logical interpretation to the results of E_1 and E_2 and then computes the "xor" of the outcomes.

^^	0	{0,1}	1
0	0	{0,1}	1
{0,1}	{0,1}	{0,1}	{0,1}
1	1	{0,1}	0

- E_1 `?` E_2 `:` E_3 or `if (`E_1`,` E_2`,` E_3`)`
  Let V_1, V_2, V_3 be the results of E_1, E_2, E_3. First the logical interpretation is applied to V_1; let the outcome be V_1'. Then the overall result is

$$(V_1' \,?\, V_2 : V_3) \;=\; \begin{cases} V_2 & \text{if } V_1' = 1 \\ V_3 & \text{if } V_1' = 0 \\ \mathit{join}\,(V_2, V_3) & \text{if } V_1' = \{0,1\} \end{cases}$$

  The only remarkable case is the third, saying that if the condition is ambiguous, the least representable set containing both the then-part and the else-part is taken (see section 5.28.6 for *join*).

  Examples: $1\,?\,2:3 \;=\; 2;$ $\quad 0\,?\,2:3 \;=\; 3;$
  $\{0,1\}\,?\,2:3 \;=\; (2..3\ \mathit{with}\ x\,\%\,1 == 0) \;=\; \{2,3\};$
  $\{0,1\}\,?\,3:9 \;=\; (3..9\ \mathit{with}\ x\,\%\,2 == 1) \;=\; \{3,5,7,9\}$

- `always (`E`)`
  applies the logical interpretation to the result of E and then replaces an "ambiguous" result $\{0,1\}$ by 0 ("false"). This operator is especially useful in AIS assertions (see section 5.11).

E	0	$\{0,1\}$	1
$always\,(E)$	0	0	1

Examples:

$(3 == 3) \;=\; 1;$ $\qquad$ $always\;(3 == 3) \;=\; 1;$

$(3 == 4) \;=\; 0;$ $\qquad$ $always\;(3 == 4) \;=\; 0;$

$((3..4) == 4) \;=\; \{0,1\};$ $\qquad$ $always\;((3..4) == 4) \;=\; 0;$

5.28.13 Switch Expressions

The general syntax of switch expressions is either of

```
switch (E) {
  L1 => R1,
     ⋮
  Ln => Rn,
  default D
}
```

```
switch (E) {
  L1 => R1,
     ⋮
  Ln => Rn
}
```

where E, L_1, …, L_n, R_1, …, R_n, and D are expressions. The basic idea is as usual: The head expression E is compared with the left-hand sides L_1, …, L_n. If L_i is the first match, then R_i is the result; and if there is no match, D is taken if present, and if there is no match and no default, the result is undefined.

As usual, the details are more complex because of the set semantics of expressions. The comparison between E and L_i is not done by the `==` operator, but by the `in` operator described in section 5.28.11 (the examples below provide some motivation for this choice). The evaluation can be described by the following algorithm, which uses an auxiliary counter i and a list $\mathcal{R}$ of sets (the list of sets contributing to the final result).

1. Let $i = 1$ and $\mathcal{R}$ be the empty list.
2. Let b be the result of evaluating E *in* L_i, which is 0 (false), 1 (true), or $\{0,1\}$ (ambiguous).
3. If $b = 1$, i.e. the set resulting from E is included in the one resulting from L_i, add the result set of R_i to the list $\mathcal{R}$ and go to 7.
4. If $b = 0$, i.e. the result sets of E and L_i are disjoint, replace i by $i+1$ and go to 6.
5. If $b = \{0,1\}$, add the result set of R_i to the list $\mathcal{R}$, replace i by $i+1$ and go to 6.
6. If $i \le n$, go back to 2. If $i = n+1$ and there is a default D, add the result set of D to the list $\mathcal{R}$.
7. If $\mathcal{R}$ is the empty list, which happens if all comparisons resulted in 0 (disjoint sets) and there is no default, the overall result is undefined, which is represented by $(-inf..inf)$. Otherwise the final result is the join of the result sets contained in the list $\mathcal{R}$, i.e. the least representable set including all those intervals (see section 5.28.6 for join).

Example 1:
`switch (E) { 1 => 100, 2 => 200, 3 => 300 }`
If the left-hand sides L_i are numerical constants like here, it does not matter whether == or `in` is used for comparison.
If E is 2, the result is 200.
If E is 5, the result is undefined, represented as $(-inf..inf)$.
If E is $(1..2)$, the result is $join(100,200) = (100..200\ with\ x\,\%\,4 == 0)$.
If E is $(2..5)$, the result is $join(200,300) = (200..300\ with\ x\,\%\,4 == 0)$.
If E is $(0..8\ with\ x\,\%\,2 == 0) = \{0,2,4,6,8\}$, the result is 200.
If E is $(0..8\ with\ x\,\%\,4 == 0) = \{0,4,8\}$, the result is undefined $((-inf..inf))$.

Example 2:
`switch (E) { 1 => 100, 2 => 200, 3 => 300, default 400 }`
If E is 2, the result is 200.
If E is 5, the result is 400 (the default).
If E is $(1..2)$, the result is $join(100,200,400) = (100..400\ with\ x\,\%\,4 == 0)$ since there is no perfect match ($b = 1$), but only false or ambiguous results of comparison so that the iteration continuous till the end of the switch and thus reaches the default.
If E is $(2..6)$, the result is $join(200,300,400) = (200..400\ with\ x\,\%\,4 == 0)$,
but $(2..6\ with\ x\,\%\,2 == 0) = \{2,4,6\}$, leads to $join(200,400) = (200..400\ with\ x\,\%\,8 == 0)$.

Example 3:
`switch (E) { 1 => 100, 2 => 200, 3 => 300, default (-inf..inf) }`
If E is 2, the result is 200.
If E is 5 or $(1..2)$ or $(2..5)$, the result is undefined, i.e. $(-inf..inf)$, since the default is reached. This shows that an undefined default is different from a missing default.

Example 4:
`switch (E) { 0 => 0, 1..7 => 200, 8 => 100, default 300 }`
If E is 1, the result is 200, and likewise if E is 2, or $(3..5)$. The reason is that the comparisons $1\ in\ (1..7)$ and $2\ in\ (1..7)$ and $(3..5)\ in\ (1..7)$ yield 1 and so the evaluation stops without reaching the default. If == had been used in place of `in`, these comparisons would have given the ambiguous result $\{0,1\}$ and thus the default would have been reached giving result $join(200,300)$.
If however E is $(7..8)$, then the `in`-comparisons with $(1..7)$ and 8 yield ambiguous results so that the default is reached and the final result is the join of 200, 100, and 300, which is $(100..300\ with\ x\,\%\,4 == 0)$.

5.28.14 The Result of Value Analysis

The special variable `value` can only occur in parametric loop bounds (section 5.20.1). When evaluated, it yields the result of a^3's built-in loop bound analysis. This is an unbounded interval if the loop bound analysis did not find anything useful. In any case, the `value` result can be modified by other expressions, in particular the `restrict` and `try` operators (section 5.28.6). Examples for the usage of these operators are presented in section 5.20.2.

5.28.15 The `before` Operator

An expression is always evaluated with reference to a certain program point, and it is evaluated once for every context of that point. For instance, the expression in a parametric loop bound (section 5.20.1) is evaluated at the program point immediately before the entry of the loop in all contexts of the routine from which the loop has been extracted. The point of reference matters if the expressions contains registers; for each register, its value at the point of reference is inserted.

Normally, all subexpressions are evaluated at the same program point as the overall expression. The point of reference for a subexpression *E* can be modified by writing

E **`before`** *`ProgramPoint`*

Here, `before` indicates that the new point of reference for *E* is immediately before the given *`ProgramPoint`*. More exactly, *E* is evaluated in the situation

- after executing any `is entered with` annotations (section 5.10) at the *`ProgramPoint`*,
- but before executing the machine instruction at the *`ProgramPoint`*.

The context in which the subexpression is evaluated is the same as the context of the overall expression if the new point of reference is in the same routine as the old. Otherwise, the subexpression is evaluated in all contexts that may contribute to the context of the overall expression, and all resulting sets of values are joined.

Usage of `before` is not very transparent, may be expensive (the subexpression may be evaluated in many contexts), and suffers from various restrictions: the new point of reference must occur somewhere before the old, and:

The `before` operator can only occur in parametric loop bounds (section 5.20.1). It cannot be used in specifications of register values (5.10), declarations of conditional infeasibility (5.17.2), and declarations of condition values (5.18.2).

An alternative method to refer to the values of registers at a certain point is to save the register value into a user register at that point and then to use the user register later. This method appears to be more transparent and works for all kinds of expressions, not only parametric loop bounds.

Example:

A loop implementing binary search is typically bounded by the binary logarithm of the size of the array in which the search is performed. Here is an example with a source code annotation, which relies on the fact that parameter `size` is passed in register `R2`:

```
function search (array a, int size, int x) {
  ...
  while (...) {
    /* ai: loop here  max ceil(log(R2, 2)); */
    /* Binary search over a */
    ...
  }
}
```

Now suppose the value of R2 is modified between the entry of search and the beginning of the loop. To refer to the original value in the loop bound, the before operator can be used:

```
function search (array a, int size, int x) {
  ...
  while (...) {
    /* ai: loop here  max ceil(log(R2 before "search", 2)); */
    ...
  }
}
```

Alternatively, the original value of R2 may be saved in a user register:

```
function search (array a, int size, int x) {
/* ai: instruction "search" is entered with @size = R2; */
  ...
  while (...) {
    /* ai: loop here  max ceil(log(@size, 2)); */
    ...
  }
}
```

5.28.16 Types and Units

There are three types of expressions: those that produce pure numbers (type *number*), those that produce an amount of bytes (type *bytes*), and those that produce an amount of processor cycles (type *cycles*). The expression evaluator performs type checks, e.g., it is impossible to sum up an amount of bytes and a pure number.

Bytes and cycles expressions result from the usage of space and time units, e.g., `10 bytes` or `50 cycles`. The available space units apart from `bytes` are listed in section 5.30.4. Time units are `cycles`, and the "real-time" units `sec`, `msec`, `usec`, and `nsec`. In contrast to `cycles`, the "real-time" units can only be used if the clock rate of the processor has been specified as described in section 5.2.

Syntactically, units are postfix operators with very low precedence (see also section 5.28.1). Thus, `2500 + @X nsec` is a legal expression parsed as `(2500 + @X) nsec`.

However, `100 cycles + 50 usec` is parsed as `(100 cycles + 50) usec` which gives a type error:

```
Argument 2 of function 'add':
Expected same type as first argument,
but found 'cycles' vs. 'number'
```

To solve this problem, use parentheses like this: `(100 cycles) + (50 nsec)`.

The only binary operator that binds weaker than units is the range operator `..` (see section 5.28.6). Thus, `(100 cycles .. 200 cycles)` is correct, but `(100 .. 200 cycles)` is not

since it is parsed as `(100 .. (200 cycles))`. The separators `=>` and `,` used in `switch` expressions (section 5.28.13) also bind weaker than units so that you can write `switch (@X) {1 => 10 cycles, 2 => 20 usec}`.

5.28.17 The Interpretation of the Values of Overall Expressions

As explained above, expressions in general yield a set of fractions that are possible values. Yet the various places where expressions may occur cannot cope with such general results and need to interpret them in various ways.

- Assertions, possibly about values of registers or memory cells (section 5.11), e.g.,
  `assert instruction` *P* `is entered with` *Expr*$_1$`, ...,` *Expr*$_n$`;`
  declarations of conditional infeasibility (section 5.17.2), e.g.,
  `snippet` *S* `is not executed if` *Expr*`;`
  and declarations of condition values (section 5.18.2), e.g.,
  `condition` *P* `is true if` *Expr*`;`
  `condition` *P* `is exactly` *Expr*`;`
  require a logical value. Therefore, the result of the *Expr* in these syntactic environment is transformed by the logical interpretation introduced in section 5.28.12. The result of logical interpretation is 1 meaning "true", 0 meaning "false", or $\{0,1\}$ meaning "ambiguous".

- Specifications of register values and definitions of user registers (section 5.10), e.g.,
  `instruction` *P* `is entered with r1 =` *Expr*$_1$`, @X =` *Expr*$_2$`;`
  and loop bound specifications (section 5.20.1), e.g.,
  `loop` *L* `min` *Expr*$_1$ `max` *Expr*$_2$`;`
  require sets of integers. These are obtained by applying an implicit `int` operator (section 5.28.9) to the results of the expressions ("integer interpretation"). The `int` operator rounds down the lowest value in the set to the nearest integer below it and rounds up the highest value to the nearest integer above it. It preserves modulo information as much as possible, and in any case adds the modulo information $x \,\%\, 1 == 0$ for sets of integers.

- After the integer interpretation, negative numbers are removed from the results of loop bound expressions, e.g., $(-2..5$ *with* $x \,\%\, 1 == 0)$ is replaced by $(0..5$ *with* $x \,\%\, 1 == 0)$. If nothing remains after removing the negative numbers, the loop bound is ignored.

- Specifications of execution times of code snippets excluded from analysis (section 5.15.3), e.g.,
  `snippet` *P* `is not analyzed and takes` *Expr*`;`
  and of additional execution times (section 5.13), e.g.,
  `instruction` *P* `additionally takes` *Expr*`;`
  require an integer amount of processor cycles. The expression *Expr* therefore has to have type *cycles*, which can be achieved by using time units (see section 5.28.16). The given time units are converted into cycles and then the integer interpretation is applied. Only the upper

end is taken from the resulting integer set. If it is unbounded or negative, a warning is issued and a value of 0 is assumed instead.

Because of the integer interpretation, extracting a subexpression and assigning it to a user variable may change the value of an expression. Consider the following (artificial) example:

```
loop ... exactly 10 * (1/2);
```

gives a loop bound of exactly 5. However,

```
instruction ... is entered with @half = 1/2;
loop ... exactly 10 * @half;
```

gives minimum loop count 0 and maximum loop count 10. The reason is that the value of `@half` is not really $1/2$, but $int\,(1/2) = (0..1\ with\ x\,\%\,1 == 0)$ because of the implicit integer interpretation, and $10 * (0..1\ with\ x\,\%\,1 == 0) = (0..10\ with\ x\,\%\,2 == 0)$.

Extracting a subsexpression may also change the value of an expression because of the added integer modulo constraint. For instance, the condition $4.2\ in\ (3..6)$ yields 1, but

```
instruction ... is entered with @Y = (3..6);
... 4.2 in @Y ...
```

will give 0 since the value of `@Y` is not $(3..6)$, but $(3..6\ with\ x\,\%\,1 == 0)$.

5.29 Program Points and Targets

Most AIS specifications refer to *program points* and/or *targets*. In the simplest cases, these are routine names or addresses, but much more complex descriptions are possible (see for instance the annotation of `MC` in section 5.8.2). Here, a detailed description of the syntax of program points and targets follows.

A *`ProgramPoint`* is the same as a *`Target`*. A *`Target`* may be any of the following:

- *`Address`*
  A simple address such as `0x91a4`. This is a byte address as all TriCore addresses.
- `ROUTINE "`*`Name`*`"`
  A routine given by its name, or more exactly the entry point of a routine. Routine names must be quoted, e.g., `"main"`.
- `LABEL "`*`Name`*`"`
  A reference to a label as defined by a `LABEL` specification (see section 5.25). Program labels cannot be referenced like this.
- `"`*`Name`*`"`
  This may be a label or a routine name. a^3 first tries to resolve it as a label, then as a routine.

- `file 'Name' line Number`
  refers to line *Number* in source file *Name*. This feature can only be used if source file reading is included in your license. It is inherently imprecise and relies on the line information in the executable. See section 5.31.1 for the details of translating source lines into program points.

  The single quotes around *Name* let the path name separator \ of MS Windows stand for itself. With double quotes, \ would have to be written as \\ like in C-strings (see also section 5.30.2).

- `here`
  can only be used in source code annotations and then refers to the code the annotation is attached to (see section 5.31). This feature is inherently imprecise and relies on the line information in the executable.

- `pc`
  can only be used when specifying the targets of a computed call or branch and then denotes the address of the computed call or branch instruction (see section 5.8.2).

- *Target* `+` *n Unit* or *Target* `-` *n Unit*
  where *Unit* is a unit of addressing, such as `byte`/`bytes`, or `word`/`words` (see section 5.30.4 for a complete list of units). One `word` corresponds to 4 bytes.

  To evaluate *Target* `+/-` *n Unit*, **`exec2crl`** first finds the address *Target* refers to and then adds *q* to this address or subtracts *q* from it, where *q* is the amount of bytes corresponding to *n Unit* (1 word = 4 bytes).

  Example: `routine "R" + 0x1a bytes`
  refers to byte `0x1a` relative to the beginning of routine *R*.

- *Target* `+` *n Things* or *Target* `-` *n Things*
  Here, *Things* refers to a class of instructions described by a predicate. The following atomic predicates exist:

`INSTRUCTIONS`	(arbitrary instructions),
`CALLS`	(call instructions),
`BRANCHES`	(branch instructions),
`RETURNS`	(return instructions),
`BORINGS`	(instructions that do not alter the control flow),
`CERTAIN`	(instructions classified exactly by **`exec2crl`**),
`UNCERTAIN`	(instructions that cannot be classified exactly by **`exec2crl`**),
`CONDITIONAL`	(conditional instructions),
`UNCONDITIONAL`	(unconditional instructions),
`COMPUTED`	(control-relevant instructions with computed targets),
`UNCOMPUTED`	(control-relevant instructions with static fixed targets),
`PREDICTABLE`	(control-relevant instructions with predictable targets),
`UNPREDICTABLE`	(control-relevant instructions with unpredictable targets),
`READS`	(instructions reading from memory),
`WRITES`	(instructions writing to memory),
`LOOPS`	(loop entries as recognized by loop transformation).

Plural endings may be omitted, e.g., `BRANCH` is equivalent to `BRANCHES`.

Several of these atomic predicates refer to the classification of instructions as presented in section 5.8.1: so-called *boring* instructions cannot alter the control flow, while control-relevant instructions may perform control-flow alterations. The latter are classified further into *branches*, *calls*, and *returns*. These classes are not disjoint; they provide "may" information, i.e. there are instructions classified as branch-or-call without **`exec2crl`** being able to decide between these two possibilities. Such a branch-or-call satisfies the predicates `UNCERTAIN`, `BRANCH`, and `CALL`, while a definite branch satisfies `CERTAIN` and `BRANCH`.

Predicates can be combined by logical operators `not`, `&&` (and), `||` (or), and `^^` (xor). For instance, `CONDITIONAL && BRANCH` refers to a conditional branch, and `COMPUTED && CALL` to a computed call (in contrast to a computed branch). As usual, `&&` binds stronger than `||` and `^^`.

`BRANCH && CALL` refers to instructions classified as potential branch and also as potential call, i.e. to branch-or-call instructions and to branch-or-call-or-return instructions. `BRANCH || CALL` refers to instructions classified as potential branch or as potential call, i.e. to certain branches, certain calls, but also to branch-or-call instructions, branch-or-return instructions etc.

To evaluate a target of the form *`Target +/- n Things`*, **`exec2crl`** first finds the address *`Target`* refers to and then moves forward or backward till the *n*th instruction satisfying the given predicate. In doing so, **`exec2crl`** does not follow the control flow, but moves from instruction to instruction in textual order.

Example: `routine "`*R*`" + 3 computed && call`
refers to the third computed call in routine *R*. If there are less than three computed calls in routine *R*, an error results. **`exec2crl`** might have resolved the first computed call in routine *R*. In this case, `routine "`*R*`" + 1 unpredictable && call` is equivalent to `routine "`*R*`" + 2 computed && call`.

If some future version of a^3 with improved capabilities can resolve the second computed call, too, then the annotation `routine "`*R*`" + 1 unpredictable && call` suddenly refers to the third computed call instead of the second. In contrast, annotations involving `computed` are stable w.r.t. such changes of the prediction power.

When resolving *`Target +/- n Things`* the instruction at the position referenced by *`Target`* is the first that is examined. Thus, `routine "prime" + 1 computed` always refers to the first computed call or branch in `prime`, which may be the very first instruction of `prime`. Similarly, `routine "prime" + 1 instruction` refers to the first instruction of `prime`.

Consequently, `0x1000 + 1 instruction` refers to the first instruction at or after address `0x1000`, *not* to the instruction after the instruction at address `0x1000`. Therefore `0x1000 + 1 instruction` is equivalent to `0x1000` if this address is in a code segment.

- *Target* `align +` *n Unit* or *Target* `align -` *n Unit*
  where *Unit* is a unit of addressing, such as `byte`/`bytes`, or `word`/`words` (see section 5.30.4).

  Target declarations with alignment modifiers are evaluated as follows: First, *Target* is evaluated; then, the least possible number of bytes is added to or subtracted from this location such that the new location is aligned w.r.t. *n Unit* (if *Target* is already aligned, the location is not modified at all).

  Example: `0x8143 align - 4 bytes`
  refers to the first address before `0x8143` that is divisible by 4 bytes; this is `0x8140`.

The description above is recursive. Wherever *Target* appears, it may be replaced with any of the forms above. This way, more complex target descriptions can be built. Here are some examples:

```
routine "bla" + 2 branches + 5 bytes
"main" + 2 calls + 3 instructions
```

Such descriptions refer to the original extension of the routines *before loop transformation* (see section 4.1). You should never refer to loop routines such as `prime.L1` in AIS annotations.

Implementation restriction: The target description *Target* `+/-` *n* `LOOPS` can only be used in the positive form (with `+`) and cannot be combined with any of the other descriptions. The *Target* must be a routine name (with or without the qualifier `ROUTINE`), a label name (with or without the qualifier `LABEL`), or an address.

In resolving target descriptions of the form *Target* `+/-` *n Things*, **`exec2crl`** does not move over gaps, i.e. non-reachable instruction addresses. (Gaps are no problem for `+/-` *n* `bytes`.)

Data areas such as switch tables do not count as gaps if **`exec2crl`** has recognized them or if they have been declared by a data area specification (see section 5.12.4). In fact, an address may be in a known code area, a known data area, a gap, i.e. an area not read by **`exec2crl`**, or outside the executable.

Example to the Gap Problem: Consider the following code fragment that only lists the control-relevant instructions (addresses on the left, instruction width 4 bytes, byte addresses):

`0x8000`	start of routine *R*
`0x8100`	conditional computed branch with targets `0x8110` and `0x8120`
`0x810C`	return
`0x811C`	return
`0x8200`	conditional computed branch with targets `0x8210` and `0x8220`
`0x8214`	return
`0x8300`	conditional computed branch with targets `0x8310` and `0x8320`
`0x8400`	return

This code contains a gap: code addresses `0x8218–0x821F` are unreachable. Assume there is an AIS specification

```
instruction "R" + 1 computed branches to 0x8110, 0x8120;
instruction "R" + 2 computed branches to 0x8210, 0x8220;
instruction "R" + 3 computed branches to 0x8310, 0x8320;
```

Then **`exec2crl`** resolves `"R" + 1 computed` to `0x8100` and `"R" + 2 computed` to `0x8200`, but does not resolve `"R" + 3 computed` because of the gap. The rationale is that **`exec2crl`** does not decode the instructions in the gap, which might include further computed branch instructions. Note that it would resolve `"R" + 0x300 bytes`; targets involving `bytes` are not affected by gaps. The gap problem can be overcome by declaring the contents of the gap as data:

```
area 0x8218 .. 0x821F contains data;
```

Such a specification is recommended even if the gap does not really contain data, but is filled with NOP instructions or other nonsense code. After this specification, `"R" + 3 computed` is resolved to `0x8300`.

Note that without a prior specification of the targets of the first computed branch, **`exec2crl`** does not resolve `"R" + 2 computed` because – not knowing the targets of the branch at address `0x8100` – it does not decode at these targets and thus creates a gap starting at `0x8110`. Therefore, target specifications for computed branches should always be provided in-order, starting from the first branch.

Finally, the gap problem is the reason why targets of computed calls and branches cannot be specified in the form `+/-` *n* *Things*. This is because these targets usually lie in a gap before the computed call or branch has been resolved. Filling this gap requires evaluating the targets, which is impossible as long as the gap exists. The solution is to use targets of the form `+/-` *n* `bytes` that work even in presence of gaps.

5.30 Atomic Parts of AIS Annotations

5.30.1 Keywords

The language of AIS annotations contains a large number of keywords such as `END` or `INSTRUCTION`. In writing a keyword, you may use lower and upper case letters in arbitrary combination. Thus, a keyword such as `END` may also be written as `End` or `end`, or even as `eNd`.

5.30.2 Names

In AIS files, various entities such as routines, labels, or files can be referred to by their name. Such names must be enclosed in single or double quotes to distinguish them from keywords and to separate them from their environment. Examples for names are `"prime"`, `"$activate"`, `"END"`, `"SynCPU(0x140a)"`, and `'program file'`; the last example shows that names

may contain spaces. In contrast to keywords, case matters, i.e. `"prime"` and `"Prime"` are two different names.

Names in double quotes are treated like strings in the C programming language: The `\` character is special and used to insert non-printable characters or double quotes into the name. Consequently, you must write `\\` to include the `\` character in a name with double quotes. On the other hand, `\` has no special meaning in names delimited by single quotes.

Example: `"a\\b"` is the same name as `'a\b'`, but different from `"a\b"`, which consists of the letter `a` and a backspace character.

The above conventions apply to MS Windows path names. For instance, the name `"E:\a3\tricore\prime.elf"` is illegal. The intended meaning is obtained by writing `'E:\a3\tricore\prime.elf'` or `"E:\\a3\\tricore\\prime.elf"`.

Routine names must be written as in the executable. For instance, the C routine `main` must be written as `"_main"` if the compiler adds underscores to routine names, or as `"main"` if it does not add underscores.

5.30.3 Integers

AIS files may contain integers in many different syntactic positions, serving various purposes: as addresses, as iteration counts of loops, as worst-case times of not-analyzed routines, as counters for computed calls and branches, etc. All these integers are unsigned. They can be written in decimal, hexadecimal, octal, or binary notation.

- Hexadecimal numbers must be introduced by `0x`, e.g., `0x2a`.
- Binary numbers must be introduced by `0b`, e.g., `0b101010`.
- Octal numbers must start with a leading `0`, e.g., `052`.
- Numbers starting with `1`–`9` are decimal numbers, e.g., `42`.

To improve readability, integers in any notation may be structured by means of underscores '_' that do not modify the numerical value. For instance, `3_000_000` is the same number as `3000000`.

5.30.4 Units of Addressing

Units of addressing such as `byte`/`bytes` are used in program point descriptions such as `"main" + 0x10 bytes` (see section 5.29). They also appear as *size qualifiers* in specifications of values stored in memory (see section 5.12.5) and as units in expressions (see sections 5.28.16 and in particular 5.28.5).

The basic unit of addressing is `byte`. The following additional units exist:

`halfword` or `hword`	2 bytes
`word`	4 bytes
`doubleword` or `dword`	8 bytes
`singlefloat` or `sfloat`	4 bytes
`doublefloat` or `dfloat`	8 bytes

5.31 Source Code Annotations

If your license includes this feature, AIS specifications can also be included in C source code files as special comments:

```
/* ai: specification_1; ... specification_n; */
```

They are marked by the key string `ai:`. Use only one `ai:` per comment!

The exact behavior of a^3 w.r.t. source code annotations is governed by the flag `Extract annotations from source files` in the `General` tab of the `Analyses` view (see section 3.4.3.3). If the flag is switched on, **`exec2crl`** looks for source code annotations in all source files whose names are listed in the debug information of the executable. Section 4.5 describes how these files are found in the file system. Apart from the files named in the debug information, **`exec2crl`** reads all `.c` and `.h` files that are included from these files by the C include mechanism, or are included from the AIS file by the AIS include mechanism (section 5.27), or from any included source file by the AIS include mechanism used in a special `ai` comment.

If the flag `Extract annotations from source files` is switched off, **`exec2crl`** does not look for annotations in the source files that are only mentioned in the executable. It does read source files mentioned in the AIS file via the AIS include mechanism or the `file ... line ...` program point description. It may also read other source files for the purpose of source code visualization, but not for extracting annotations.

Whenever source code annotations are extracted from a source file, the file is scanned for `ai` comments independent from any `if`, `ifdef`, or `ifndef` directives.

All specifications possible in AIS files may appear as source code annotations. Such annotations may be written at arbitrary places. There is an additional feature that can only be used in source code annotations: They admit a special program point or target `here`, which roughly denotes the place where the annotation occurs. More exactly, `here` means a source code line in the first place, which is then translated into a code address (see section 5.31.1 below). The same translation applies to program points of the form `file '`*`Name`*`' line` *`Number`*.

5.31.1 Translation of Source Code Lines into Code Addresses

Program points may refer to source code lines, either explicitly via `file '`*`Name`*`' line` *`Number`* or implicitly via `here`, which refers to the line where it occurs. **`exec2crl`** translates source code

lines into code addresses using the line information in the executable. This only works if the AIS file specifies the compiler as described in section 5.3. Even in this case, the line information is not always precise. The rule is that line n points to the first instruction associated with a line number $\geq n$. A line without any source code is thus equivalent to the first line behind it containing source code. A line containing source code in general corresponds to many assembly instructions. A reference to such a line points to the *first* such instruction.

If a^3 cannot resolve a source-code reference, it complains with a message such as

```
Unable to resolve source position ...:
No address in debug info corresponds.
```

A reason for such a problem may be inconsistencies between relative and absolute file names. Therefore, setting the flag `Strip compilation path mentioned in the executable` in the `General` tab of the `Analyses` view (see section 3.4.3.3) sometimes solves the problem. Path replacements (see sections 3.4.5 and 4.5) may also be helpful.

Examples for source-code annotations:

```
/* ai: recursion here max 6; */
uint fac (uint n) {
  ...
}
```

is equivalent to

```
uint fac (uint n) { /* ai: recursion here max 6; */
  ...
}
```

and both are equivalent to

```
recursion "fac" max 6;
```

written somewhere in the AIS file.

Similarly, the annotation

```
/* ai: loop here max 20; */
for (i=3; i*i <= n; i += 2) {
  ...
}
```

is equivalent to

```
for (i=3; i*i <= n; i += 2) { /* ai: loop here max 20; */
  ...
}
```

yet these two annotations do not work since the code corresponding to `for (...)` starts with the initialization `i=3`, which is not in the loop. (It does not matter that the second annotation lies inside the braces of the loop body.)

Writing `here + 1 loop` does not work either because of the implementation restriction mentioned in section 5.29.

Yet the following works:

```
for (i=3; i*i <= n; i += 2) {
    if (divides (i, n))        /* ai: loop here max 20; */
        return 0;
}
```

Note that it is not required that `here` exactly denotes the loop start address. It suffices that it resolves to an address anywhere in the loop.

There is another potential problem with source annotations: the line information in the executable is created by the compiler. It becomes invalid when lines are added or deleted in the source file.

If you add lines while annotating, you need to recompile!

On the other hand, making lines longer does not spoil the line information. Thus you need not recompile if you only append annotations to existing lines or put annotations into already existing empty lines. Nevertheless **`exec2crl`** issues a warning if a source file is younger than the executable.

If **`exec2crl`** reads a source file, then it considers all specifications in it irrespective of any `if`, `ifdef`, or `ifndef` directives. Thus, source references like `here` may lead to errors if they occur in conditional code not compiled into the executable.

Chapter 6

Requirements and Restrictions

First, the machine where a^3 is run must provide enough resources to allow for a successful analysis and visualization. The larger the application to be analyzed, the more resources in terms of memory and execution speed are needed. In addition to this basic requirement, certain conditions must be satisfied in order to obtain correct analysis results.

6.1 Hardware Design and System Configuration

The TriCore processor maps different address ranges to the same physical memory. Thus, some memory cells can be accessed using different addresses. The address used for a particular access decides whether the access is cached or uncached.

cached:	`0x8000 0000` – `0x9fff ffff`
uncached:	`0xa000 0000` – `0xbfff ffff`

a^3 treats those address ranges as separate memory areas so that writes via one address range do not influence reads via the other address range.

6.2 Sequentiality of the Analyzed Tasks *(Timing Analysis)*

During timing analysis, a^3 computes an upper bound of the worst-case execution time of a *task*. A task must be a sequentially executed piece of code, i.e. there must not be any threads, parallelism, or external events. a^3 assumes no interference from the outside. Effects of exceptions, interrupts, DRAM refreshes, input/output, timers, and other processors or co-processors are not reflected in the predicted runtime and have to be considered separately. This remark applies in particular to the cache effect of an interrupt. The worst-case time penalty caused by the effect of an interrupt on the cache is the time needed for a complete refilling of the cache. A better approximation may be obtained by taking the foot print of the interrupt into account, i.e. the cache lines that may possibly be replaced because of the cache accesses of the interrupt routine.

6.3 Restrictions on the Analyzed Code

The code of the analyzed task must have been compiled by the Tasking C Compiler or the GNU C compiler (GCC) from a restricted subset of ANSI-C.

- For timing and value analysis: Dynamic data structures are not supported. Functions from the `alloc` family like `malloc` and `calloc` must not occur in the analyzed code.

  For stack analysis: `alloca` must not be used.

- For timing and value analysis: `setjmp` and `longjmp` must not be used in the analyzed code.

  For stack analysis: `setjmp` and `longjmp` should be used in a safe way, as specified in their description.

- a^3 assumes that all analyzed routines satisfy the calling conventions of the compiler, while all external routines and routines declared not-analyzed violate the conventions. If an analyzed routine violates the calling conventions or a not-analyzed routine satisfies them, this should be made known to a^3 by an annotation as introduced in sections 5.16 and 5.15.5, respectively. The stack effect of routines violating the conventions, be they analyzed or not, should be declared as explained in section 5.15.5.

- Return addresses must not be modified.

- Timing analysis does not support accesses spanning multiple memory areas with different timing or cache characteristics.

- The TC1766 and TC1796 processor models as used in timing analysis do not support instruction fetches from the Data Flash and the Boot Rom both as part of the Program Memory Unit, and from the Local Memory Interconnect (LMI).

- The analyzed code must not contain any task switches.

These restrictions only apply to the code that it is being analyzed. The remainder of the executable may contain the excluded features.

6.4 User Annotations

The user must provide certain input in order to get any analysis results at all:

- The settings in the `TriCore` view should be appropriate (see section 3.4.2).

- If a routine or any other code snippet is declared not-analyzed, its properties must be correctly specified by the user (see section 5.15).

- Timing analysis requires that for each loop in the analyzed task whose loop bound cannot be found automatically, a maximum iteration number must be specified in the AIS file (see sections 5.20 and 5.22) or as source code annotation (see section 5.31). This includes ordinary loops found by the loop transformation (see section 4.1), but also multi-entry loops that cannot be handled by the loop transformation. The maximum iteration bounds for the latter loops must be expressed indirectly by flow constraints (see section 5.22).

  Stack analysis only requires loop bounds for loops whose body has a non-zero stack effect. Such bounds can only be specified for ordinary loops found by the loop transformation; flow constraints do not take effect in stack analysis.

  Value analysis does not strictly require upper loop bounds; yet, usually it yields more precise results if the maximum iteration numbers of loops are known.

- For stack and timing analysis: For recursive routines, the maximum call depth (for stack analysis) and the maximum call number (for timing analysis) must be specified in AIS format (see section 5.19).

  For value analysis: Recursive calls may be bounded by user annotations, however, this is not strictly necessary to get analysis results, but may improve their precision.

- The possible call targets of the indirect calls and unresolved computed branches in the analyzed task must be specified in AIS format (see section 5.8.2).

The above-mentioned user annotations are required to obtain any analysis results at all. Of course, these results can only be safe if all user annotations are correct, i.e. the specified maximum iteration numbers are upper bounds of the true iteration numbers, all possible targets of a computed call or jump must be among the specified targets, etc.

To ensure correct results from timing and value analysis, any volatile memory areas including areas corresponding to volatile global variables must be declared as volatile in the AIS file (see section 5.12.3).

6.5 Requirements Specific to Stack Analysis

Stack analysis may fail in three ways: it may fail altogether, producing nothing; it may produce unknown results **?** that are correct in a technical sense, but of course not very satisfactory; and it may produce results that look like real results, but are wrong. The latter may happen if there are wrong AIS specifications, or if there is an absolute setting of some stack pointer because such settings reset the computed local stack level to 0. Such absolute stack-pointer settings can be detected since they cause a^3 to issue a warning. Generally, the results of a^3 should not be trusted if any errors or warnings appear in the messages view of a^3 (or in the report file if reporting is enabled). Here is a list of requirements that should be met to obtain meaningful stack analysis results.

- As mentioned above, there should be no absolute settings of any stack pointer because such settings reset the computed local stack level to 0.

- The stack usage of *external routines* and *not-analyzed routines* is marked as unknown if it is not declared by a user annotation (see section 5.15.4). External routines are routines that are not contained in the executable, and not-analyzed routines are routines excluded from analysis by an AIS specification (see section 5.15).
- Instructions modifying a stack pointer in an unexpected way, i.e. other than adding a constant or setting it to a constant, lead to an unknown value for this stack pointer. (Absolute settings do not lead to an unknown value, but cause a warning and reset the value to 0.)
- An unknown value results if a dynamic amount of space is allocated on the stacks, i.e. an amount whose size cannot be statically determined. This in particular happens if the size of the allocation depends on input data.
- Generally, there is a risk of unknown stack values if hand-written assembly code is analyzed in which the stack is not used in a way the compiler would do.

When stack analysis yields the unknown value **?**, it usually prints a diagnostic message explaining the reasons for **?** in the a^3 message window.

The joining process employed to produce results for larger program parts from the results for their constituents propagates unknown values up the hierarchy. Thus, the result node at the whole program shows **?** if **?** occurs anywhere in the program.

There is one other special result value, namely $\bot$, which stands for the stack usage of unreachable code. Unlike **?**, $\bot$ is ignored by the minimization process computing the results for higher hierarchy levels.

Stack analysis is aborted to the effect that there are no analysis results at all if the application contains unrestricted recursion. Unrestricted recursion can be avoided by declaring call limits (see section 5.19.1).

Chapter 7

The Analyses of a^3

Various a^3 commands cause the generation of a combined call graph and control-flow graph of the part of the executable that is accessible from a selected start point. Depending on the command, the generated graph may or may not be annotated with analysis results.

- A start operation (`Analysis` → `Start` or `F7` or) in a control-flow graph "analysis" view, or a control-flow graph operation (`Analysis` → `Control-Flow Graph` or) in one of the other analysis views causes a visualization without timing and stack analysis. Consequently, the result is a pure call graph and control-flow graph without any timing and stack annotations. It is described in section 7.1.

- An "interactive" operation (`Analysis` → `Interactive` or) in a value-analysis view starts a value analysis for the start point specified in the analysis view. The analysis results for a particular instruction can be accessed by selecting the corresponding node in the graph and pressing `v`. The resulting presentation of the value analysis results is described in section 7.2.

- A start operation (`Analysis` → `Start` or `F7` or) in a stack-analysis view starts a stack analysis for the start point specified in the analysis view. The analysis results are merged into the combined call graph and control-flow graph of the analyzed code. Stack analysis and the features of the generated graphs are described in section 7.3.

- A start operation (`Analysis` → `Start` or `F7` or) in a timing-analysis view starts a timing analysis for the start point specified in the analysis view. The analysis results are merged into the call and control-flow graph, as described in section 7.4.

- An "interactive" operation (`Analysis` → `Interactive` or) in a timing-analysis view also starts a timing analysis for the start point specified in the analysis view. The difference to the start operation lies in the fact that the generated graph provides access to the cache and pipeline states of the analyzer, which are displayed as separate graphs on demand (see section 7.5).

Since there are separate actions for performing stack and timing analysis, it is impossible to obtain graphs showing stack and timing analysis results together.

Figure 7.1: Call graph without analysis results

7.1 Combined Call Graphs and Control-Flow Graphs

When a start operation (`Analysis` → `Start` or `F7` or) is performed in a control-flow graph "analysis" view, or a control-flow graph operation (`Analysis` → `Control-Flow Graph` or) is performed in one of the other analysis views, a^3 starts the visualization of the accessible part of the combined call graph and control-flow graph of the application without a prior stack or timing analysis.

The combined graph has a hierarchical structure. The outermost level is a call graph visualizing the routines of the application and their calling relationships. Each routine contains a subgraph visualizing the basic blocks of the routine and the control-flow in between, and each basic block contains a subgraph visualizing its individual assembly instructions. Besides the routines of the original application, there may be loop routines resulting from the loop transformation, e.g., `prime.L1` in Figure 7.1, left.

This section describes the features of the combined graphs and their visual representation (meaning of colors etc.), based on the assumption that – unless stated otherwise – all options in the `Visualization` view are disabled except for `Manhattan edges` (see section 3.4.6). See chapter 8 for a description of commands for manipulating the visual representation.

The color scheme described in the following is the default color scheme. It can be customized via the `Visualization` view (see section 3.4.6). A legend explaining the meaning of the colors in the current color scheme can be invoked from a graph-browser view by `File` → `Toggle legend` or by clicking on the button in the `File` tool bar.

7.1.1 Call Graphs

The combined call and control-flow graphs generated by a^3 represent the input program down to the level of individual assembly instructions, however these details are initially hidden so as to get an overview of the entire program. When the visualization of the program is started, it shows its *call graph*. An example call graph is depicted in Figure 7.1.

The handling of the loop routines resulting from loop transformation (section 4.1) depends on the setting of the flag `Inline loop routines` in the `Visualization` view (section 3.4.6). If `Inline loop routines` is off, loop routines are shown as separate routines like `prime.L1` in the graph in Figure 7.1, left. If `Inline loop routines` is set, loop routines are visualized as part of their host routines, which means that they do not appear on the call graph level (see Figure 7.1, right).

7.1.1.1 Routines

The nodes of the call graph represent *routines*, i.e., assembly procedures, which have several different origins as described in section 4.2. In short, there are routines coming from the executable, virtual loop routines such as `prime.L1` created by loop transformation (see section 4.1), and anonymous routines `:Anon_`*A* with start address *A*.

Routines from the executable, loop routines, and anonymous routines are usually visualized as (light) yellow boxes containing the name of the routine. Infeasible routines, i.e. routines that are – according to the knowledge of a^3 – never executed, either because value analysis found some conditions that are always true or always false, or because the routines have been declared infeasible by a `never executed` annotation (see section 5.17), are shown as gray boxes.

Routine boxes usually have a black border. Routines that contain computed calls whose targets could not be determined by a^3 are displayed with a red border. The possible call targets of computed calls can be declared as described in section 5.8.

Certain routines can be declared as not-analyzed in the AIS file (see section 5.15). a^3 does not read the code of not-analyzed routines. Hence it does not find any routines that are only reachable from not-analyzed routines. By declaring library routines as not-analyzed, you can concentrate on the routines in your own application. Not-analyzed routines are depicted as orange boxes. Generally, orange routines do not contain code, while yellow routines do.

7.1.1.2 Info Fields at Routines

The nodes of the graphs may carry up to three info fields, which are normally hidden, but can be popped up by special commands of the visualization view (see section 8.1.7).

The *first and second info fields* of routine nodes are empty in graphs without analysis results. The *third info field* of a node representing routine *R* contains the full path name of the file where *R* is defined.

7.1.1.3 Calling Relationships

The edges of the call graph represent calling relationships. An arrow from routine R_1 to routine R_2 usually means that routine R_1 contains at least one call of routine R_2. The arrow may also be caused by a jump from R_1 to the beginning of R_2, or from the fact that R_2 is a loop routine resulting from a loop within routine R_1.

In the example depicted in Figure 7.1, left, routine `prime` calls `even`, which calls `divides`. Furthermore, `prime` includes a loop that has been transformed into routine `prime.L1`. This loop contains a call of `divides`.

Two or more calls of or jumps to R_2 in R_1 are represented by a single edge.

Recursive calls from a routine *R* to itself are *not* represented by a call edge from the node representing *R* to itself. The call edge is rather part of the basic block graph of *R*. This also holds for the virtual recursive calls resulting from loop transformation. Thus, there is no call

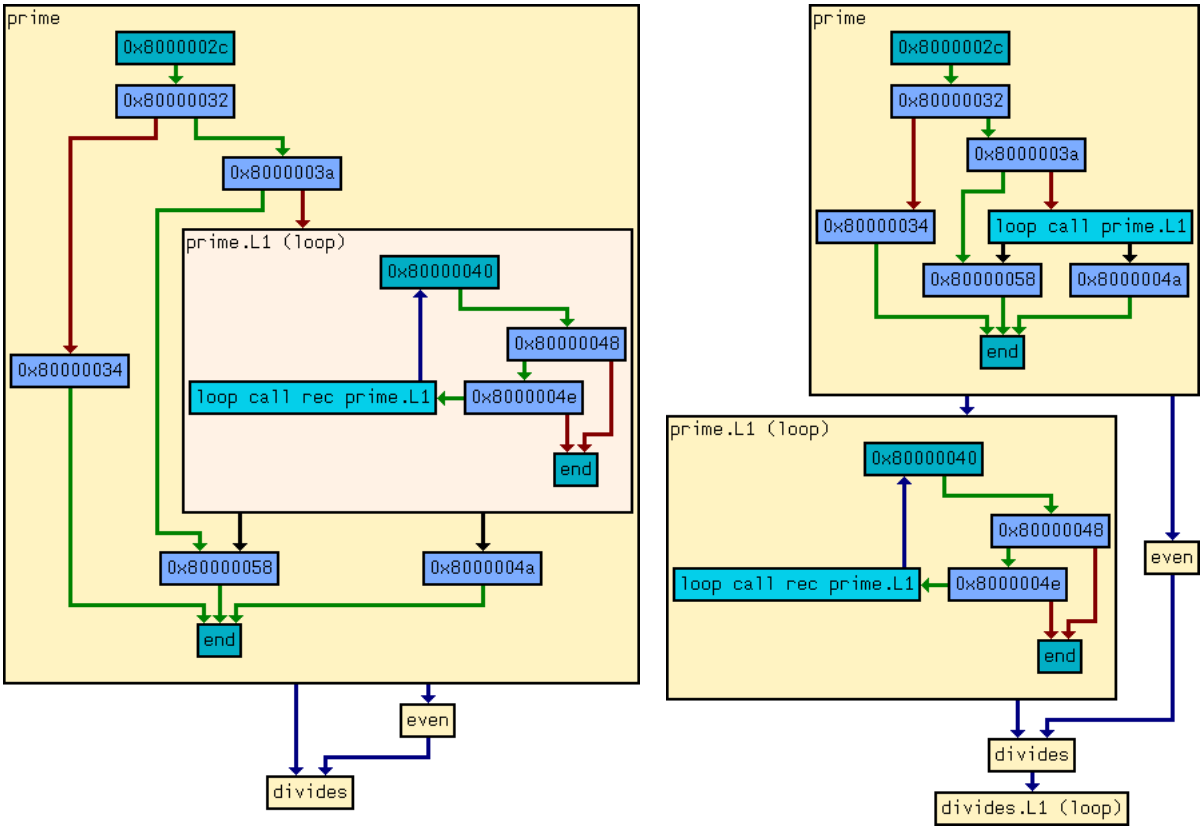


Figure 7.2: Graph with inlined loop routine and with separate loop routine

edge from `prime.L1` to itself in Figure 7.1, left. This call edge is rather part of the basic block graph of `prime.L1` (see Figure 7.2).

7.1.2 Basic Block Graphs

The internal and analyzed routines (those with the lighter color) contain *basic block graphs*, whose nodes represent basic blocks and whose edges indicate the control flow within the routine. These basic block graphs are hidden in the call graph shown as initial visualization (see Figure 7.1). They can be made visible for some or all routines using the commands described in section 8.2.

Figure 7.2 presents the basic-block graphs of routine `prime` and of the loop routine `prime.L1` extracted from it by loop transformation (section 4.1). Figure 7.2, left, shows the graph with inlined loop routines, i.e. with `Inline loop routines` enabled in the `Visualization` view (section 3.4.6). Therefore, `prime.L1` is shown as a box within its host routine. Figure 7.2, right, shows the same graph without loop-routine inlining. In this case, the loop routine `prime.L1` appears as separate routine in the graph. All these visualizations were obtained with the visualization option `Display source code` disabled. The effect of enabling this option is described in section 7.1.5.

7.1.2.1 Basic Blocks

The nodes of the basic block graph represent basic blocks. A *basic block* is a sequence of instructions that have consecutive addresses and are executed sequentially, without any branching of control flow. Thus, jump, call, and return instructions can only occur at the very end of a basic block, and their targets always are at the beginning of a basic block.

In the visualization of the basic block graph, the instruction sequences in the basic blocks are initially hidden, but can be made visible by special commands (see section 8.2). Their visual representation is described in section 7.1.3.

In each routine, there are two special basic blocks: the entry block and the exit block. These special blocks are dark green, the other blocks are blue or light green. The entry block is the first basic block that is executed when the routine is called. It usually contains an instruction sequence like the ordinary basic blocks. In Figure 7.2, the entry block of `prime` is the block labeled `0x8000002c`, and the entry block of `prime.L1` is `0x80000040`. Note that the entry block is not necessarily placed at the top of its routine box.

The exit block is an artificial construct that symbolizes the exit from the routine. It does not contain any instructions (not even a return instruction). The exit block is visualized as a green box labeled by the word `end`.

Loop transformation introduces artificial loop-call blocks that do not contain any instructions either. Such loop-call blocks are depicted as light green boxes. Figure 7.2, right, shows two loop-call blocks, labeled `loop call prime.L1` and `loop call rec prime.L1`. Generally, a loop routine *R*`.L`*n* is called from two kinds of blocks, labeled `loop call` *R*`.L`*n* and `loop call rec` *R*`.L`*n*. Block `loop call` *R*`.L`*n* occurs in the routine where *R*`.L`*n* is extracted from and marks the place from which the loop has been extracted. The blocks labeled `loop call rec` *R*`.L`*n* indicate the recursive calls of *R*`.L`*n* in *R*`.L`*n*. If the loop routine is inlined as in Figure 7.2, left, it replaces the loop-call block `loop call` *R*`.L`*n*, which is therefore no longer visible.

When a^3 finds that control flows to a different routine without a call instruction, it introduces a *dummy call*. Dummy calls may result from a jump or even from fall-through to the next instruction if the target or next instruction happens to be in a different routine. More details can be found in sections 4.2 and 7.1.4.

All blocks are visually represented by boxes with some label. The border of the box is red if the block contains computed calls whose targets could not be determined by a^3. The possible call targets of computed calls can be declared in the AIS file (see section 5.8).

If the visualization option `Display source code` is disabled, ordinary blocks are labeled by the hexadecimal start address of the block, e.g., `0x8000002c`. If `Display source code` is enabled, ordinary blocks are labeled by source code provided that this is permitted by your license (see section 7.1.5).

7.1.2.2 Info Fields at Basic Blocks

The nodes of the graphs may carry up to three info fields, which are normally hidden, but can be popped up by special commands of the visualization view (see section 8.1.7).

The *first info fields* of basic blocks are empty. The *second info field* contains a symbolic block identifier. The *third info field* of all basic blocks (except for exit blocks and loop call blocks) contains the start address of the block given as a hexadecimal number, e.g., `0x8000002c`.

7.1.2.3 Control-Flow Edges

In the basic block graph of a routine, the basic blocks are connected by edges representing the control flow in the routine. These control-flow edges are depicted as arrows. The colors of these arrows denote different kinds of control flow:

- Black edges are *normal edges*. Normal edges usually represent a control flow caused by textual succession in the assembly program.
- Red edges are *"false" edges*. "False" edges represent the control flow from a conditional jump instruction to its textual successor, i.e., the direction of control flow if the condition evaluates to false.

  The compiler sometimes negates conditions so that a "false" edge may lead to the assembly code corresponding to the "true" successor of the original C condition.
- Green edges are *"true" edges*. "True" edges usually represent the control flow from a conditional jump instruction to its target, i.e., the direction of control flow if the condition evaluates to true. Hence, red and green edges often occur in pairs. An example is block `0x80000032` in Figure 7.2. From this block, a green edge points to `0x8000003a` and a red edge to `0x80000034`.

  A "true" edge may also represent the control flow from an unconditional jump instruction to its target (the condition is always true). Hence, a block ended by an unconditional jump has a single outgoing edge, which is green.

  In addition, "true" edges point from blocks that end in a return instruction to the exit block of the routine. An example for this is the edge from `0x80000058` to the exit block `end` in Figure 7.2. Furthermore, "true" edges point from blocks ending in a call instruction to the block after the call, e.g., from `0x8000002c` to `0x80000032`.

  Finally, "true" edges represent the control flow from a computed jump to its possible targets. The compiler often translates the head of a switch statement into a block ended by a `ji` instruction with a register operand. In the visual representation, a bunch of green edges will point from this block to the various cases of the switch statement.
- Dark blue edges, or *call edges*, are mentioned for the sake of completeness. Call edges represent the control flow from a call instruction to the entry of the routine called. They usually do not occur in the basic block graph of a single routine, but rather connect routines in the call graph. Call edges resulting from recursive calls in a routine *R* form an exception: They are part of the basic block graph of *R* and lead from the block containing the recursive call to the entry block of *R*. This holds in particular for the virtual recursion introduced by loop transformation; the edge from `loop call rec prime.L1` to `0x80000040` in Figure 7.2 provides an example.

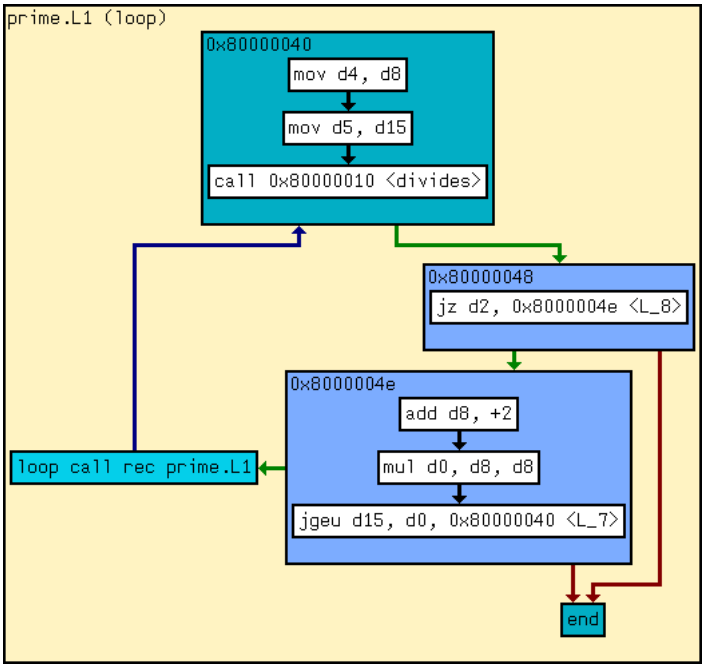


Figure 7.3: Some basic blocks with instruction sequences

7.1.3 Instruction Sequences

In the same way as routines contain basic block graphs that are initially hidden, basic blocks also contain instruction sequences that are initially hidden. They can be exposed by the same commands that can be used to expose basic block graphs (see section 8.2).

7.1.3.1 Instructions

Instruction sequences are visualized as graphs. In these graphs, each node represents an assembly instruction from the visualized program. Such a node is visualized as a white box labeled by the mnemonic form of the assembly instruction together with its operands (see Figure 7.3).

If the instructions in a basic block are shorter than the block label, the latter is truncated.

7.1.3.2 Info Fields at Instructions

The nodes of the graphs may carry up to three info fields, which are normally hidden, but can be popped up by special commands of the visualization view (see section 8.1.7).

The *first and third info fields* of instructions are empty. The *second info field* contains the address of the instruction given as a hexadecimal number, e.g., `0x8000002c`. Such addresses are byte addresses, as all addresses displayed by a^3.

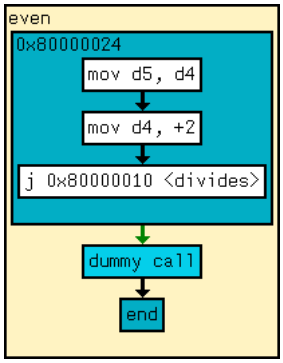


Figure 7.4: Routine ended by a jump to another routine

7.1.3.3 Edges Between Instructions

The instructions in a single basic block are executed sequentially and therefore connected by black normal edges. If a block ends with a branch to its own beginning, the corresponding "true" edge is also part of the instruction graph of the block. Furthermore, "true", "false" or even call edges may occur between instructions if the instruction graphs of several basic blocks have been merged using special commands (see section 8.2).

7.1.4 Tail Calls

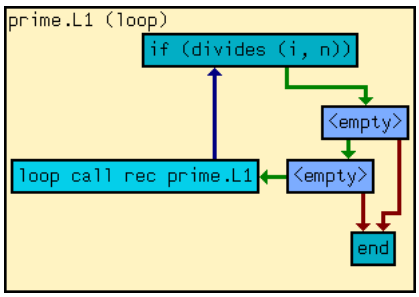
Sometimes, a compiler or an assembler writer performs *tail call optimization:* if the last instruction before the return instruction of routine R is a call to R', it may be replaced by a jump to the beginning of R'. The return address on the stack then is still the return address of R. Hence, R' immediately returns to the call site of R when it performs its return instruction. Thus, the return instruction of R is not reachable and can be omitted. In the graph picture, the call point indicating the call from R to R' is directly connected to the end block of R (see Figure 7.4).

7.1.5 Displaying Source Code in Basic Block Graphs

Up to now we have considered the case where all flags in the `Visualization` view are disabled, except for `Manhattan edges` (see section 3.4.6). If the visualization option `Display source code` is enabled *and this feature is included in your license*, basic blocks are labeled by fragments of source code instead of addresses such as `0x8000002c`, provided that references to source code lines have been compiled into the executable and the source code files are found by a^3. The visualization option `Normal layout` causes the basic blocks in basic block graphs to be ordered as in the executable. Figure 7.5 shows the source code and the basic block graph of routine `prime` and the extracted loop routine `prime.L1` when `Display source code` and `Normal layout` are enabled, where `bool` is `unsigned char`, and `uint` is `unsigned int`. The vertical ordering of basic blocks reflects the textual order of the basic blocks in the executable.

The names of the source files are extracted from the debug information in the executable. Section 4.5 describes how these files are found in the file system. a^3 uses the line information in the executable to associate source line numbers with code addresses.

```
bool prime (uint n) {
  uint i;
  if (even (n))
      return (n == 2);
  for (i = 3; i * i <= n; i += 2) {
      if (divides (i, n))
          return 0;
  }
  return (n > 1);
}
```



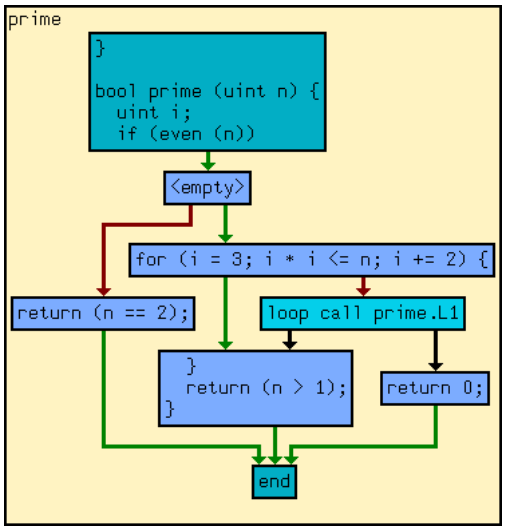


Figure 7.5: Basic block graph with source code

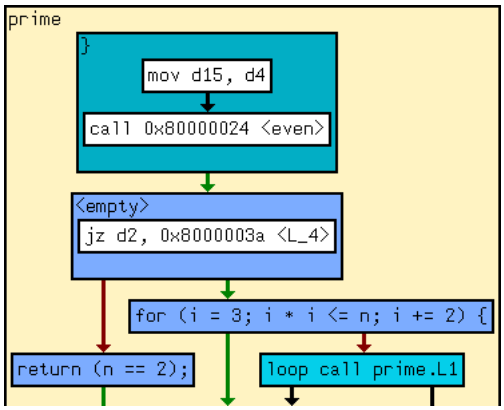


Figure 7.6: Look at the instruction level

The debug information in the executable usually suffers from several deficiencies. The association of source lines to basic blocks is not always as one might expect. In particular, there are basic blocks without any source code information, which are still labeled by their address.

Figure 7.6 shows the assembly instructions in the first two blocks of `prime`. Note that block labels are truncated if the enclosed instruction sequences are exposed: only some initial part of the first line of the block label remains. The first block, whose label contains `if (even (n))`, ends in the `call` instruction that calls `even`. The test of the result of `even` and the corresponding branch instruction are contained in the second block, which is labeled by its address since it has no associated source code. The line with the closing brace } at the beginning of the first block is the last line of the previous routine.

A look at the instruction level also helps to explain the colors of the edges leaving the second block of `prime`. The source says

```
if (even (n))
    return (n == 2);
```

but the edge leading to the node labeled by `return (n == 2);` is a red "false" edge. The reason for this is that the edge colors in a^3 are defined by the assembly instructions, and the compiler has chosen to test the result of `even` by testing the negated condition `even(n) == 0`.

7.2 Value Analysis

As explained in section 4.4.2, value analysis tries to statically determine the contents of registers and memory cells at every program point for each context. The obtained information can be employed to inspect which parts of the program access which memory areas and to verify that the program does not perform illegal accesses.

A value analysis is performed as part of WCET analysis, but can also be invoked as a separate operation without subsequent determination of WCETs. When a start operation (`Analysis` → `Start` or `F7` or ▶) or an "interactive" operation (`Analysis` → `Interactive` or ▶) is initiated in a value-analysis view, a^3 performs a value analysis and then displays a combined call graph and control-flow graph, which is similar to graphs without analysis results as described in section 7.1.

Results obtained from value analysis are twofold:

- Information for individual instructions about register and memory contents and read or written memory areas can be explored inside the a^3 GUI. This is only possible if value analysis has been started by the "interactive" operation.
- Information about assertions that have been specified in the AIS file (see, e.g., sections 5.12 and 5.17) and whether they could be refuted by the analysis, as well as the output that is produced by certain AIS directives (see section 5.23) can be found in the report file (see section 3.4.1 on how to declare a file name for the report).

In the following we explain how the former information can be interactively accessed.

The examples in this section are generic, i.e. not specific to the TriCore architecture.

The control flow graph and the information in the report file are always generated; however, in order to interactively access value analysis results for individual instructions, an interactive value analysis has to be started. Inside of a value-analysis view, this can be done by clicking on the ▶ button in the tool bar or by selecting `Analysis` → `Interactive` from the menu. In contrast to WCET or stack analysis, the generated control flow graph will not be annotated with analysis results. Instead, the value information at an instruction can be accessed by first selecting the node which belongs to the instruction and then pressing **v** or selecting `Show value analysis results` from the context menu. After that, a new view will appear in the section `Results view` in the list of available views. The title of the new view is the address of the instruction considered.

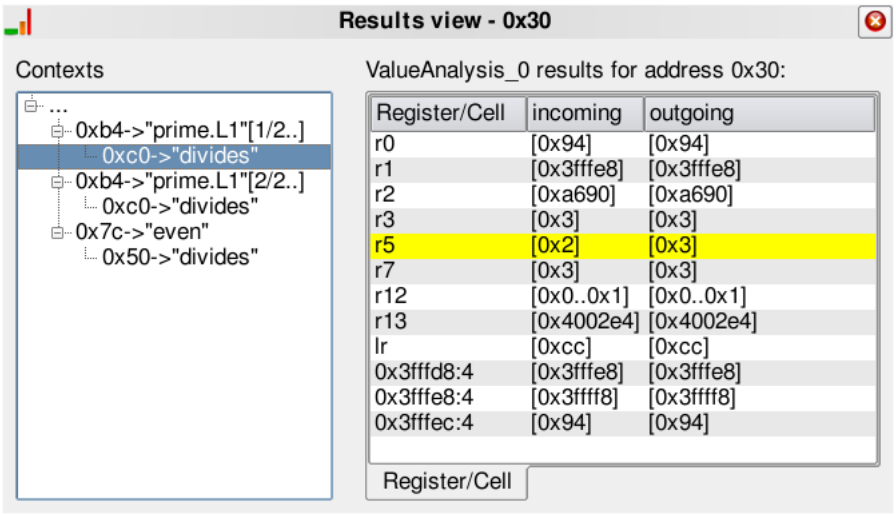


Figure 7.7: Results view for an instruction not accessing memory (`addi r5, r3, +0`)

7.2.1 The Tree of Contexts

The results view lists the contexts of the considered instruction on the left in a tree-like fashion (contexts have been introduced in section 4.3). Figure 7.7 shows the results view for the instruction at address `0x30`. This instruction is located in routine `divides` called from a loop in `prime` and also from `even`. Since the context parameter `max-unroll` (see section 5.4) is 2 in this example, there are two loop contexts and thus three contexts altogether:

```
0xb4->"prime.L1"[1/2..], 0xc0->"divides"
0xb4->"prime.L1"[2/2..], 0xc0->"divides"
0x7c->"even", 0x50->"divides"
```

These contexts are displayed in the context panel in a tree-like form (see Figure 7.7). When a leaf context is selected in this panel, the corresponding value analysis results are displayed in the `Register/Cell` tab on the right (see section 7.2.2).

The context panel features a context menu for generating AIS annotations from the value analysis results (see section 7.2.3).

7.2.2 Registers and Memory Cells

Depending on whether the instruction under consideration accesses memory or not, the `Register/Cell` tab will look differently. If the instruction does not access memory as in Figure 7.7, there is a single table showing the contents of registers and memory cells. If the instruction accesses memory as in Figure 7.8 there is a second table on the top listing the accessed memory locations.

Information about register contents and memory cells is shown in at least three columns. The first column lists the registers and memory cells for which value information is available. Registers are identified by name and memory cells by their start address and width, separated by a colon,

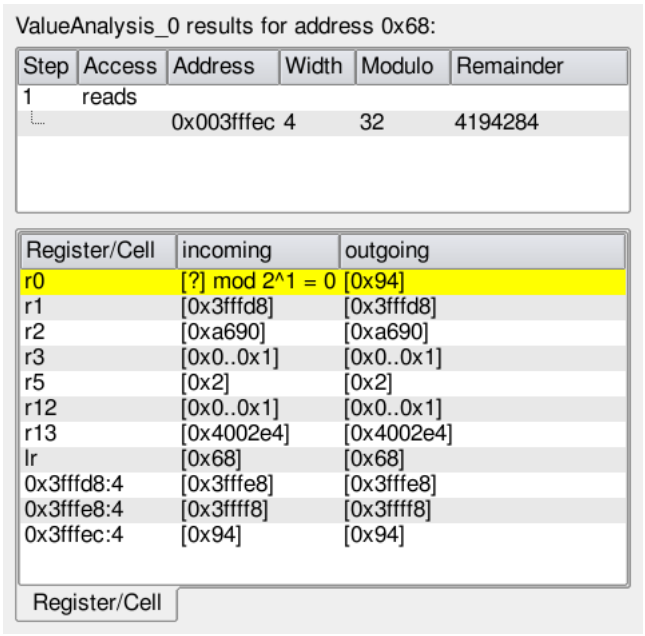


Figure 7.8: Results view for an instruction reading from memory (`lwz r0, +20(r1)`)

e.g., `0x3fffd8:4`. The `incoming` column displays the content information right before the instruction is executed; information about the contents valid after execution of the instruction is contained in the `outgoing` columns. If the instruction has only a single successor, there is only one `outgoing` column; otherwise, for multiple successors, for example, if the instruction of interest is a conditional branch, there is one column for every feasible path. If a register or a memory cell is changed by the instruction, the corresponding row will be highlighted in yellow (see Figure 7.7); the instruction considered is `addi r5, r3, +0`, which copies `r3` into `r5`. The actual content information can be given in several ways:

- `[`A`]` means that the value is known to be exactly A.
- `[`A_1`..`A_2`]` means that there is more than one possible value; the set of possible values is given by an *interval* consisting of a lower bound A_1 (the smallest possible value) and an upper bound A_2 (the largest possible value).
- `[?]` means that the analysis was unable to find out anything.
- `[?] mod 2^`k = m means that the analysis did not find any lower or upper bound, but knows that the lowest k bits of the value form the number m. For instance, `[?] mod 2^1 = 1` means that the value is odd.
- `[`A_1`..`A_2`] mod 2^`k = m means that the analysis found a lower bound A_1 and an upper bound A_2, and in addition knows that the lowest k bits of the value form the number m.

Information about memory accesses is described in six columns (see Figure 7.8). The `Step` column is only of relevance if the architecture contains instructions performing several memory accesses; in this case there will be a separate row for every access of the instruction. The `Access`

column distinguishes read and write accesses. The `Width` column specifies the width of the access. The remaining three columns, entitled by `Address`, `Modulo`, and `Remainder`, describe the set of possible addresses of the access. The information in the `Address` column can be a single address `[`A`]`, an interval `[`A_1`..`A_2`]`, or `[?]` meaning that no bounds have been found. If the `Modulo` column contains k and the `Remainder` column contains m, this means that the lowest k bits of the possible addresses form the number m. In contrast to the table of register/cell values, this information is also displayed for exact addresses. For instance, Figure 7.8 shows an exact `Address 0x003fffec`, `Modulo 32`, and `Remainder 4194284`, which is the decimal representation of `0x003fffec`.

7.2.3 Creating AIS Annotations

The value analysis results shown in the `Register/Cell` tab (section 7.2.2) can be transformed into AIS annotations. These annotations correspond to the `incoming` information; the `outgoing` information cannot be transformed. The AIS annotations for register values are described in section 5.10 while those for the values of memory cells are described in section 5.12.5. These annotations contain only intervals, i.e. lower and upper bounds; the modulo information is not expressed in the generated AIS annotations.

The transformation into AIS can be started via a context menu in the context panel. If there is more than one context in the context panel, the context menu features two different commands:

- **`Create AIS annotations`**
  creates AIS annotations valid for all contexts. These annotations do not contain the values for a specific context, but values resulting from joining over all contexts.
- **`Create context-sensitive AIS annotations`**
  creates AIS annotations from the values in a specific context. Since the context cannot be coded in AIS annotations, these annotations are incorrect when applied to other contexts or to the general situation with all contexts.

If the context panel contains a single context, then the context menu features only the command `Create AIS annotations`.

The generated AIS annotations are on the clipboard. They can be put into arbitrary AIS files by the usual paste operation. This works in the AIS browser/editor views of a^3 and in external editors.

Example: These are the AIS annotations generated with `Create context-sensitive AIS annotations` for the specific context shown in Figure 7.9:

```
instruction 0x30 is entered with "r0" = 0x94..0x94;
instruction 0x30 is entered with "r1" = 0x3fffe8..0x3fffe8;
instruction 0x30 is entered with "r2" = 0xa690..0xa690;
instruction 0x30 is entered with "r3" = 0x3..0x3;
instruction 0x30 is entered with "r5" = 0x2..0x2;
instruction 0x30 is entered with "r7" = 0x3..0x3;
instruction 0x30 is entered with "r12" = 0x0..0x1;
```

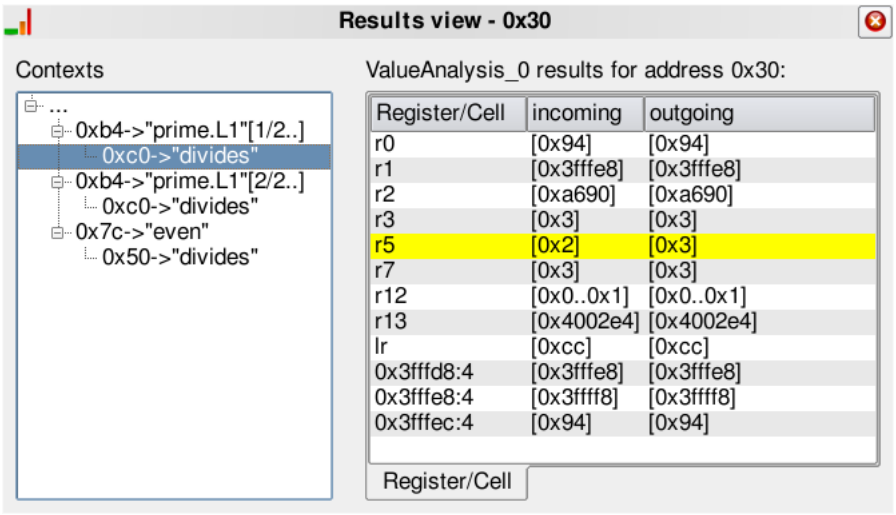


Figure 7.9: Results view for an instruction not accessing memory (`addi r5, r3, +0`)

```
instruction 0x30 is entered with "r13" = 0x4002e4..0x4002e4;
instruction 0x30 is entered with "lr" = 0xcc..0xcc;
instruction 0x30 area 0x3fffd8..0x3fffdb
                              contains 0x3fffe8..0x3fffe8;
instruction 0x30 area 0x3fffe8..0x3fffeb
                              contains 0x3ffff8..0x3ffff8;
instruction 0x30 area 0x3fffec..0x3fffef contains 0x94..0x94;
```

These are the AIS annotations generated with `Create AIS annotations` at the same instruction:

```
instruction 0x30 is entered with "r1" = 0x3fffd8..0x3fffe8;
instruction 0x30 is entered with "r2" = 0xa690..0xa690;
instruction 0x30 is entered with "r13" = 0x4002e4..0x4002e4;
instruction 0x30 is entered with "lr" = 0x68..0xcc;
instruction 0x30 area 0x3fffd8..0x3fffdb
                              contains 0x3fffe8..0x3fffe8;
instruction 0x30 area 0x3fffe8..0x3fffeb
                              contains 0x3ffff8..0x3ffff8;
instruction 0x30 area 0x3fffec..0x3fffef contains 0x94..0x94;
```

Note that some of the intervals are larger now since these intervals are the joins of the intervals for the specific contexts (the join of a list of intervals is the least interval containing all of them). Some annotations have disappeared altogether since the corresponding registers have unknown values in some other contexts.

7.3 Stack Analysis

This section describes the stack analysis performed by a^3 and the visualization of the stack analysis results induced by a start operation (`Analysis` → `Start` or `F7` or ▶) in a stack-analysis view. The visualization consists of the combined call graph and control-flow graph as described in section 7.1 annotated with stack analysis results.

TriCore documentation describes a stack for storing parameters, local variables, and temporary results. The stack pointer for this stack is register `A10`, also called `SP`. This stack is called *user stack* in a^3's output even if it is used as global interrupt stack.

A special feature of TriCore processors is their automatic context management. When a function call, interrupt, or trap occurs, the upper context (about half of the registers) is stored in the *context-save area* CSA, from which it is restored at the corresponding return from function or return from interrupt. Hence, the CSA is used like a stack on which the upper context is pushed. Therefore, the CSA is called *system stack* in a^3's parlance, and it is part of the task of stack analysis to find the maximum system-stack usage, i.e. the maximum amount of storage occupied by the saved contexts in the CSA. Note that the stack usage is always measured in *bytes* (not the number of saved contexts) so that the system stack grows and shrinks in portions of 64 byte, which is the size of a context.

a^3 for TriCore computes stack-usage information for two stacks, called *user stack* and *system stack*. The user stack is the stack with stack pointer `A10`, no matter whether it is used as user stack in the sense of the TriCore documentation or as global interrupt stack. The system stack is the context-save area CSA, which is used like a stack, but not called stack in the TriCore documentation.

Stack analysis calculates *stack levels*. The stack level for a stack is the amount of bytes stored on this stack. The stack level relative to a program point P_0 is the current stack level minus the stack level at P_0. For each routine, a^3 computes for both stacks the stack level at each instruction relative to the routine entry. These routine-local results are then combined to global stack levels for entire routines, measured relative to the start point of the analysis as specified in the analysis view (see section 3.5.2). Stack analysis finally yields upper bounds for these global stack levels, one for the user stack and one for the system stack. These upper bounds are still measured relative to the start point of the analysis, i.e. the real stack size may be larger because of some stack space allocated before control has reached the start point. Nevertheless, the results of a^3 are useful for verifying that the user and system stacks fit in the memory areas reserved for them during all runs of the program. They can also provide hints on how to reduce the sizes of these areas as much as possible, or to identify the parts of the program that consume too much stack space.

Sections 7.3.1–7.3.3 illustrate the basic stack analysis behavior using some examples. These examples are based on certain assumptions:

- The start point of the analysis is `main`.
- `exit` is declared as not-analyzed in the AIS file (see section 5.15).
- An AIS annotation exists that specifies that the not-analyzed routine `exit` does not violate

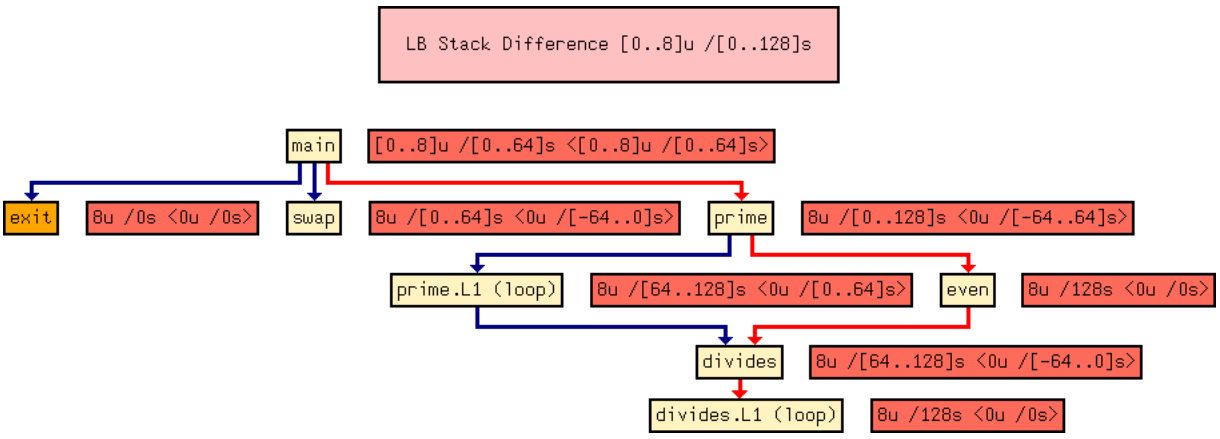


Figure 7.10: Control-flow graph with stack analysis results

the calling conventions and does not need stack space. (The latter is unrealistic, but adopted here for simplicity.)

The AIS annotations realizing these assumptions are as follows:

```
snippet "exit" is not analyzed
    and does not violate calling conventions
    and uses exactly 0 bytes of user stack
    and uses exactly 0 bytes of system stack;
```

Such annotations have been introduced in section 5.15. Examples for their effects are provided in section 7.3.4. Section 7.3.5 discusses stack analysis of recursive programs.

7.3.1 Stack Analysis Results at Routines

Figure 7.10 shows a call graph with stack analysis results, which are displayed in pink or red boxes located to the right of the nodes of the graph. Each result box carries two sets of results: a *global result*, coming first, and a *local result*, following in angular brackets. For instance, routine `prime` has global result `8u/[0..128]s` and local result `0u/[−64..64]s`. Each result consists of two parts: the first part, labeled by `u`, refers to the user stack, while the second part, labeled by `s`, refers to the system stack. Each part is an interval of possible stack levels. Intervals of the form `[`n`..`n`]` are abbreviated to n.

The local result at a routine R indicates the stack usage in R considered on its own: It consists of two intervals showing the possible range of stack levels within the routine, assuming value 0 for both stacks at routine entry.

The global result for routine R indicates the stack usage of R in the context of the whole application. It is a pair of intervals providing bounds for the stack levels while the processor is executing instructions of R, for all call paths from the start point to R. Thus, the global result at routine R does not include the stack usage of the routines called by R, nor the stack space allocated before the start point of the analysis was reached.

For instance, routine `prime` has a global result of `8u/[0..128]s` because the two calls of `prime` in `main` have stack level `8u/64s`, and `prime` itself has local stack usage `0u/[−64..64]s`. Routine `divides` has a global result of `8u/[64..128]s` because its local stack usage is `0u/[−64..0]s` and it is called in `prime.L1` at the accumulated stack level `8u/64s + 0u/0s + 0u/64s = 8u/128s` and in `even` at the accumulated stack level `8u/64s + 0u/64s + 0u/0s = 8u/128s`. These two accumulated stack levels have to be added to the local stack level `0u/[−64..0]s`, which gives the two results `8u/[64..128]s` and `8u/[64..128]s`. The join of these two results gives the global result `8u/[64..128]s` for `divides`, where the join of two intervals is the least interval containing both.

Information on the derivation of global results is contained in the first info fields of the result boxes, accessible by selecting the result box(es) and pressing **`i`** (see section 8.1.7). For each routine, the info field documents the chains of calls from the start point that lead to the maximum stack usage (the upper end of the global result intervals). Here are the contents of the first info fields of the result boxes of some routines from the example of Figure 7.10:

Routine `main`:

```
User/System Stack History:
      ->     0u/    0s <[   0..   8]u/[   0..  64]s> 'main'
      => [0..8]u/[0..64]s bytes
```

Routine `prime`:

```
User/System Stack History:
      ->     0u/    0s <   8u/  64s> 'main'
      ->     8u/   64s <   0u/[ -64..  64]s> 'prime'
      => 8u/[0..128]s bytes
```

Routine `even`:

```
User/System Stack History:
      ->     0u/    0s <   8u/  64s> 'main'
      ->     8u/   64s <   0u/  64s> 'prime'
      ->     8u/  128s <   0u/   0s> 'even'
      => 8u/128s bytes
```

Routine `divides`:

```
User/System Stack History:
      ->     0u/    0s <   8u/  64s> 'main'
      ->     8u/   64s <   0u/  64s> 'prime'
      ->     8u/  128s <   0u/   0s> 'even'
      ->     8u/  128s <   0u/[ -64..   0]s> 'divides'
      => 8u/[64..128]s bytes
```

In all lines marked by `->` but the last such line, the information in angular brackets `<...>` is the local stack level at the call of the next routine. In the last line marked by `->`, the information in angular brackets is the maximum local stack usage in this routine (there is no next routine to call). In any case, the information at the beginning of the line plus the one in angular brackets yields the information at the beginning of the next line, which thus provides the accumulated stack level till this point.

If there are several different call chains leading to the same maximum stack usage, a^3 presents only one of them. All call chains start with the call of the start point (`main` in this example) at the initial stack level `0u/0s`.

a^3 does not document how it obtains the information about the minimum stack usage (the lower (left) end of the global result interval). If minimum and maximum stack usage are obtained on different call chains, the lower end of the interval derived in the stack history is different from the left end of the global result interval.

For the routines shown above, the maximum levels of user stack and system stack are reached on the same call chain. For other examples, the situation may be more complicated since maximum user stack level and maximum system stack level are reached on two different call chains, which are both included in the info field.

The stack analysis results for the entire program (relative to the given start point) are summarized in a special pink box at the top (see Figure 7.10). For each stack, the interval for the entire program is the join of the global intervals at the routines, i.e. the smallest interval containing all global intervals.

The result box for routine *R* is displayed in *red* if for one of the two stacks, the upper end of the routine's global result interval equals the upper end of the interval for the entire program. Such results are marked by the phrase `** hits the global maximum **` in the report file (see section 7.3.2). All other result boxes at routines are displayed in *pink*. In Figure 7.10, the result boxes for all routines are red since for all routines, the maximum global user stack level is 8, the same as the maximum level of the entire program.

a^3 selects one routine hitting the global maximum and marks some path from the start point to that routine by red edges in the call graph. In Figure 7.10, the selected routine is `divides.L1`. If the user and system stack maxima are attained at different routines, a^3 will draw two red paths, one for the user and one for the system stack. In programs containing recursion, a red path may follow the recursion and thus contain loops.

7.3.2 The Report File

If a report file name is specified in the `Files` view, all results – local and global results at routines and the call chains in the info fields – are written into the report file specified. Global results visualized in red because the upper end of the interval for one of the two stacks equals the upper end of the corresponding interval for the entire program are marked by the phrase `** hits the global maximum **` in the report file:

```
routine 'divides.L1'
  => maximum usage is 8u/128s
  => ** hits the global maximum **
  => maximum local usage is 0u/0s
  => call-stack:
  User/System Stack History:
     ->      0u/    0s <    8u/   64s> 'main'
     ->      8u/   64s <    0u/   64s> 'prime'
```

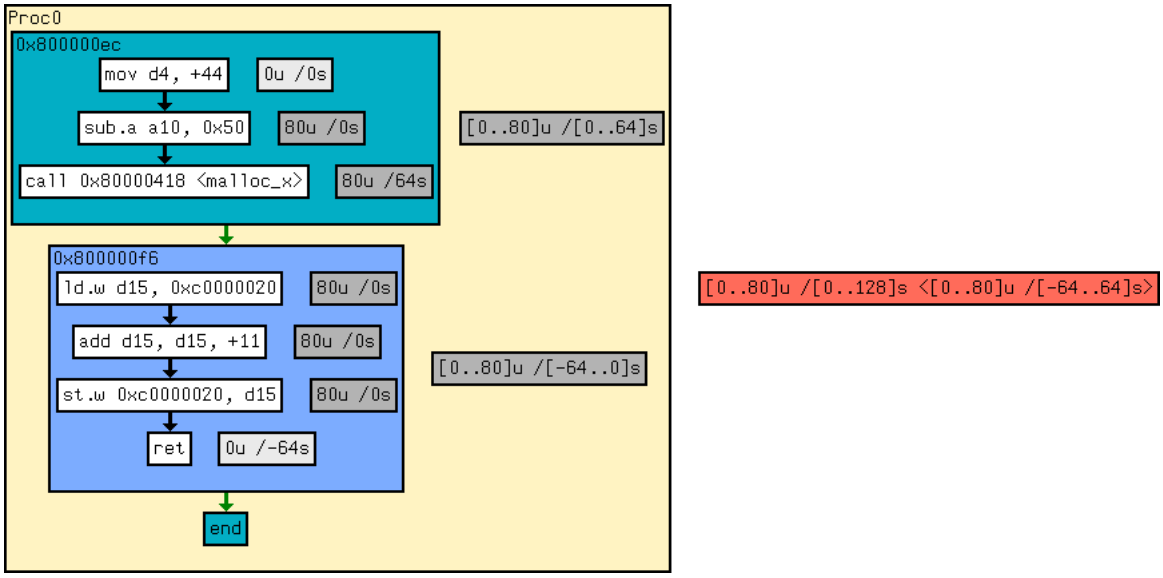


Figure 7.11: Some instructions with stack analysis results

```
->      8u/  128s <   0u/   0s> 'even'
->      8u/  128s <   0u/   0s> 'divides'
->      8u/  128s <   0u/   0s> 'divides.L1'
=> 8u/128s bytes
```

The report file is overwritten when a new timing, value, or stack analysis is started.

7.3.3 Stack Analysis Results at Basic Blocks and Instructions

Basic blocks and instructions are annotated with local stack analysis results. The results at instructions are determined by assuming a stack level of `0u/0s` at routine entry, and then taking the effects of each instruction into account. The result at a basic block is the join of the results for its instructions. Similarly, the local result at a routine is the join of the results for its basic blocks and the value `0u/0s` assumed at routine entry. Here, the join of a collection of intervals and numbers is the smallest interval containing all these intervals and numbers. The calculations for user stack and system stack are performed in parallel and independent from each other.

The result box at a block in a routine *R* is *dark gray* if the upper end of the result interval for one of the two stacks equals the upper end of the corresponding interval of the local result of routine *R*. Similarly, the result box at an instruction in a block *B* is *dark gray* if the upper end of the result interval for one of the two stacks equals the upper end of the corresponding interval at *B*. All other result boxes at blocks and instructions are *light gray*.

Figure 7.11 shows the instructions of a routine called `Proc0` with result boxes at instructions and blocks. The result at an instruction box describes the situation immediately after executing the instruction. Therefore the second instruction `sub.a a10, 0x50` is annotated with `80u/0s` because it changes the user stack pointer `a10` by a decrement of 80 (which means 80 bytes). Since the user stack of the TriCore grows downward, a decrement of the stack pointer by 80 actually

means the allocation of 80 bytes of stack space so that the user stack level (the amount of bytes on the user stack) increases by 80.

For a call instruction, "immediately after executing the instruction" refers to the situation after the call instruction, but before the first instruction of the routine called (*not* after returning from the call). The `call` instruction in Figure 7.11 saves the upper context in the context save area, called system stack by a^3. Therefore, the `call` instruction pushes 64 bytes on the system stack, changing the system stack level from 0 to 64. The corresponding return instruction in the called routine restores the upper context and thus removes these 64 bytes again so that the next instruction `ld.w d15, 0xc0000020` is executed with stack level `80u/0s`.

The final `ret` instruction of `Proc0` restores the upper context and thus pops the 64 bytes from the system stack that have been pushed by the `call` instruction calling `Proc0`. Since the user stack pointer `a10` is part of the upper context, it is restored to the original value 0 by the `ret` instruction. Therefore the stack level at the very end is `0u/−64s` relative to the level at routine entry. This is typical for routines adhering to the calling conventions.

7.3.4 External and Not-Analyzed Routines

When a^3 analyzes the stack usage of a program, it does not determine the stack usage of external or not-analyzed routines. External routines are called but not defined in the executable so that there is no routine code that could be analyzed. Not-analyzed routines are excluded from analysis by means of AIS specifications (see section 5.15). The stack usage and stack effect of all external and not-analyzed routines reachable from the start points must be specified in the AIS file as described in section 5.15.

Up to now, all the examples in this manual were created in presence of the AIS annotation

```
snippet "exit" is not analyzed
    and does not violate calling conventions
    and uses exactly 0 bytes of user stack
    and uses exactly 0 bytes of system stack;
```

This annotation specifies that `exit` should not be analyzed, that it does not use stack space (which is not realistic), and that it satisfies the calling conventions, i.e. behaves like a routine normally does. Figure 7.12 shows the analysis results for this standard AIS annotation. Although this annotation says that `exit` does not allocate stack space, the global analysis results for `exit` are `8u/0s`. This is because `main` allocates 8 bytes on the user stack before it calls `exit`, and the call of `exit` in `main` is a tail call implemented by a jump that does not allocate space on the system stack. Similarly, the global analysis result for `exit` is `28u/0s` (see Figure 7.13) if the stack behavior of `exit` is declared as

```
snippet "exit" is not analyzed
    and does not violate calling conventions
    and uses exactly 20 bytes of user stack
    and uses exactly  0 bytes of system stack;
```

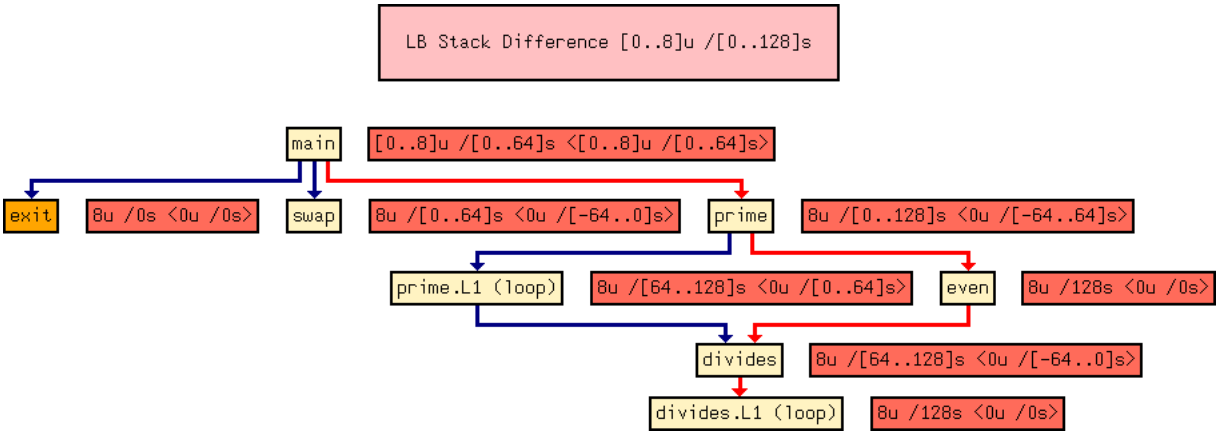


Figure 7.12: Results for the standard AIS annotation

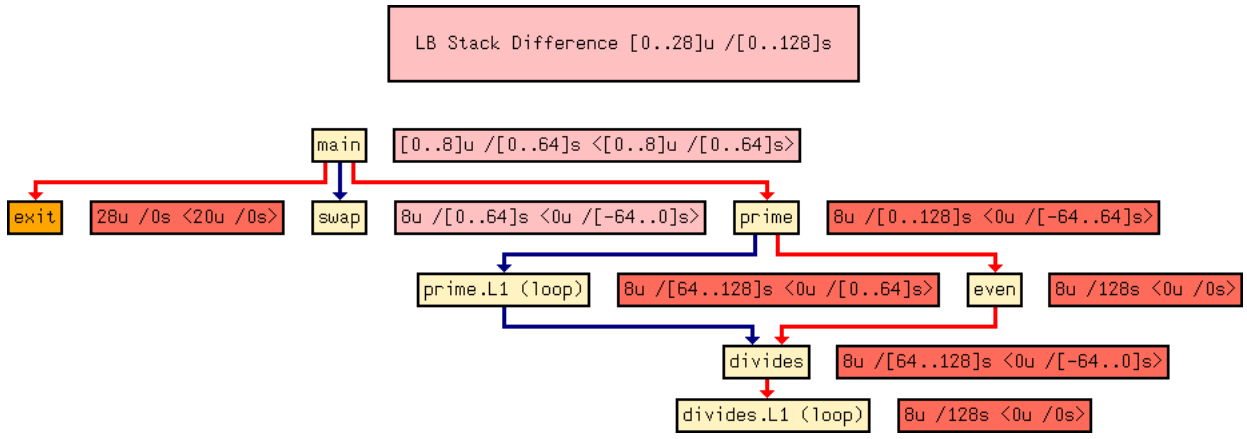


Figure 7.13: Results if `exit` uses 20 bytes of user stack

Note that the red path has now split into two since the routine with the greatest user stack level (`exit`) is now different from the routines with greatest system stack level.

If the AIS file does not contain a stack usage declaration for `exit`, the stack level of `exit` is marked as unknown (see Figure 7.14). An unknown stack usage is considered as higher than any known usage. This is the reason why now `exit` is the routine with the highest user and system stack level. The unknown stack level of `exit` leads to an unknown stack level for the whole program. The result box for the whole program therefore shows a stack level of **?** for both stacks.

While the stack usage assigned to `exit` has a visible effect, this is not true for the stack effect. It does not matter whether `exit` violates the calling conditions or not, and in case of violation, the stack effect assigned to `exit` does not matter either. The reason is that the executable reader **`exec2crl`** knows that `exit` never returns so that the stack effect of `exit` never takes effect.

The situation is different if `even` is declared as not analyzed. Any stack effect assigned to `even` by means of an AIS specification is observable since `even` does return and so there is some code after the call of `even`. To begin with, Figure 7.15 shows the effect of the specification

```
snippet "even" is not analyzed
    and uses exactly 0 bytes of user stack
```

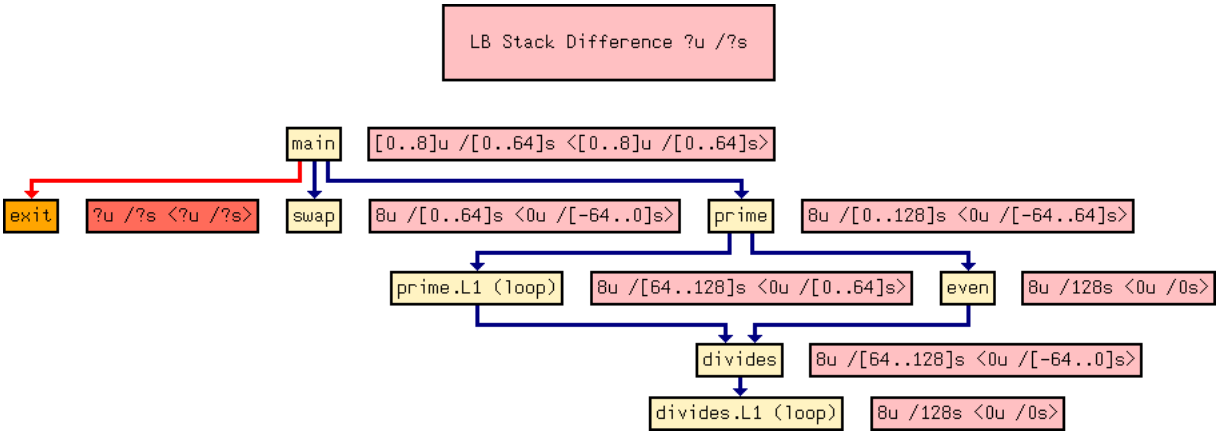


Figure 7.14: Results if the stack usage of `exit` is unspecified

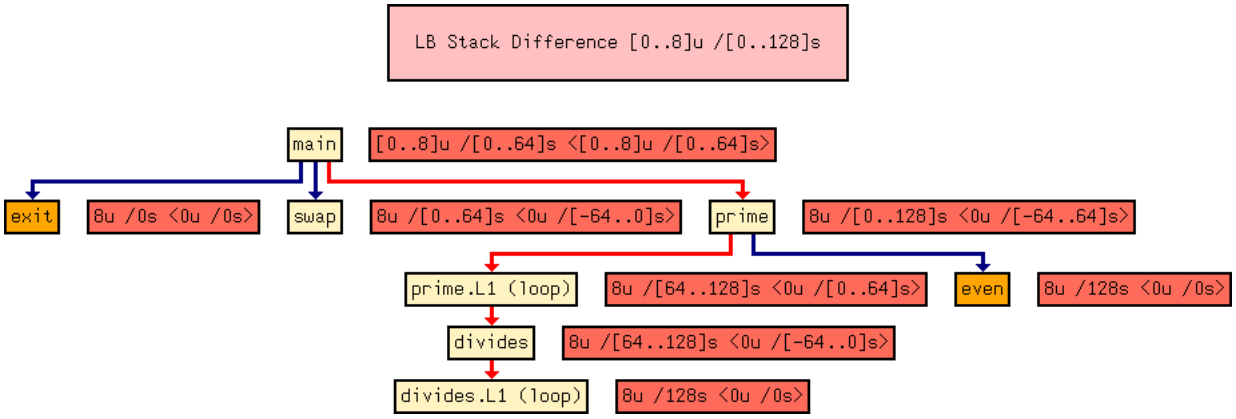


Figure 7.15: Result if `even` is not analyzed and does not violate calling conventions

```
    and uses exactly 0 bytes of system stack
    and does not violate calling conventions;
```

which asserts that `even` satisfies the calling conventions, which means that it removes 64 bytes from the system stack. The corresponding declaration with explicit stack effect is

```
snippet "even" is not analyzed
    and uses exactly 0 bytes of user stack
    and uses exactly 0 bytes of system stack
    and leaves behind exactly 0 bytes of user stack
    and removes exactly 64 bytes of system stack;
```

which yields the same result. Figure 7.16 shows the call site of `even` in `prime`, showing that the declared stack effect `0u/−64s` of `even` changes the stack level from `0u/64s` at the call to `even` to `0u/0s` at the subsequent `jz` instruction, which is the normal behavior for a routine. The same would happen if `even` were included in the analysis and there were no AIS specification about its properties. The end of `prime` shows that it again exhibits this standard stack effect of `0u/−64s`.

The following declaration specifies a non-standard stack effect:

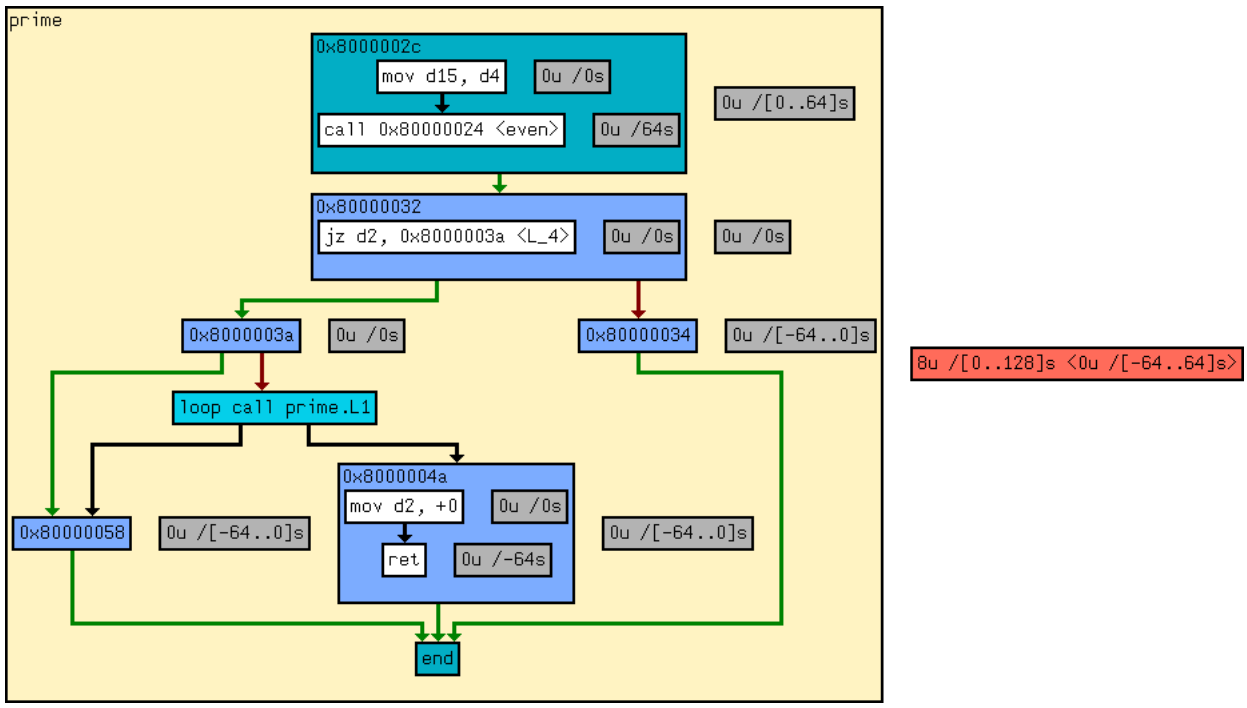


Figure 7.16: Effect of `even` on `prime` without violation of calling conventions

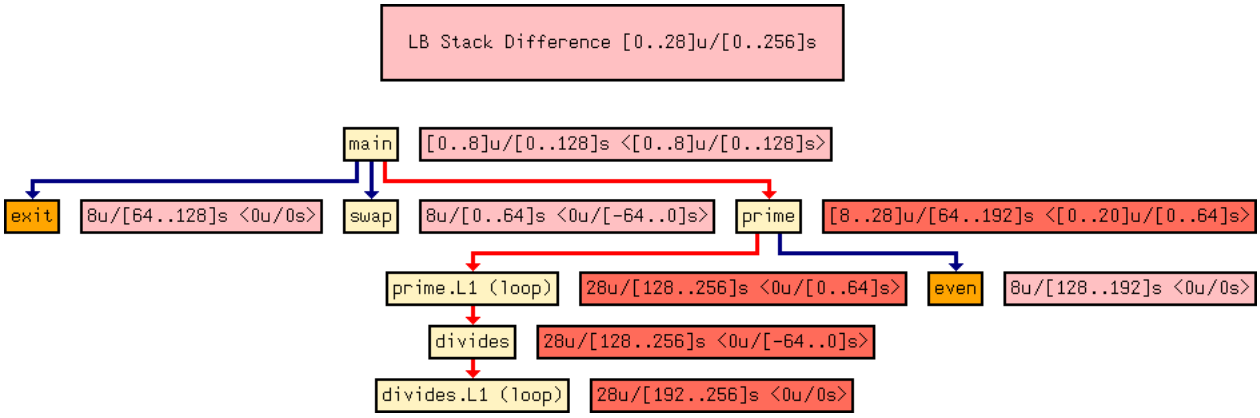


Figure 7.17: Result if `even` has a non-standard stack effect

```
snippet "even" is not analyzed
    and uses exactly 0 bytes of user stack
    and uses exactly 0 bytes of system stack
    and leaves behind exactly 20 bytes of user stack
    and removes exactly 0 bytes of system stack;
```

The resulting call graph is depicted in Figure 7.17, and the call site of `even` in `prime` is shown in Figure 7.18. The declared stack effect `20u/0s` of `even` now changes the stack level from `0u/64s` at the call to `even` to `20u/64s` at the subsequent `jz` instruction, The return instruction `ret` of `prime` restores the user stack pointer from the upper context. Hence, the final user stack level is 0, the same as if `even` respects the calling conventions (Figure 7.16), i.e. the user stack effect of

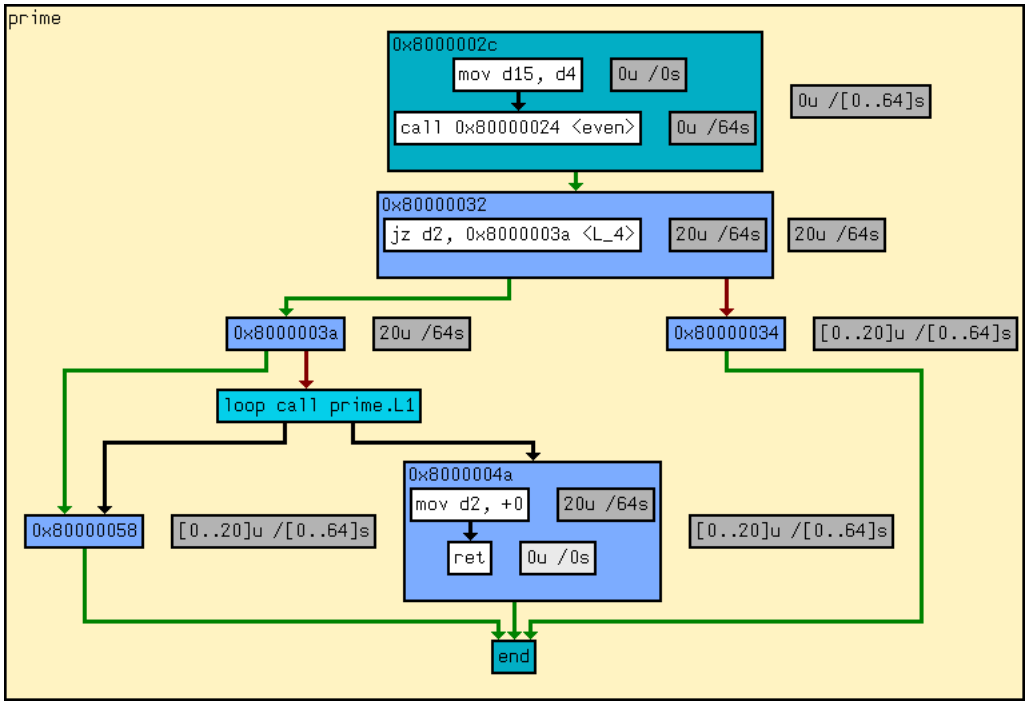


Figure 7.18: Non-standard stack effect of `even` in `prime`

`even` is confined within `prime` and has no effect outside of `prime`. In contrast, the system stack level at the end of `prime` is 0 in Figure 7.18 compared to -64 in Figure 7.16. Thus, the system stack effect is not confined within `prime` and influences the global results of all routines (compare Figure 7.17 with Figure 7.15; there are two calls of `prime` in `main`, and the first call influences the second).

7.3.5 Recursion

a^3 does not know the maximal call depth of recursive calling loops unless this information is specified in the AIS file. If the analyzed program contains recursion and the AIS file does not contain a call depth bound for all recursive loops, a^3 does not complete stack analysis and reports about the reason in the messages view.

The possible call depth specifications are described in section 5.19.1. There are basically two forms:

```
ROUTINE R INCARNATES Bounds;
ROUTINE R1 CALLS TO R2 INCARNATES Bounds;
```

The first form provides bounds for the number of occurrences of routine R on the call stack, while the second form specifies bounds for the number of occurrences of the call from R_1 to R_2.

Figure 7.19 shows the stack analysis results for

```
routine "fac" incarnates max 1;
```

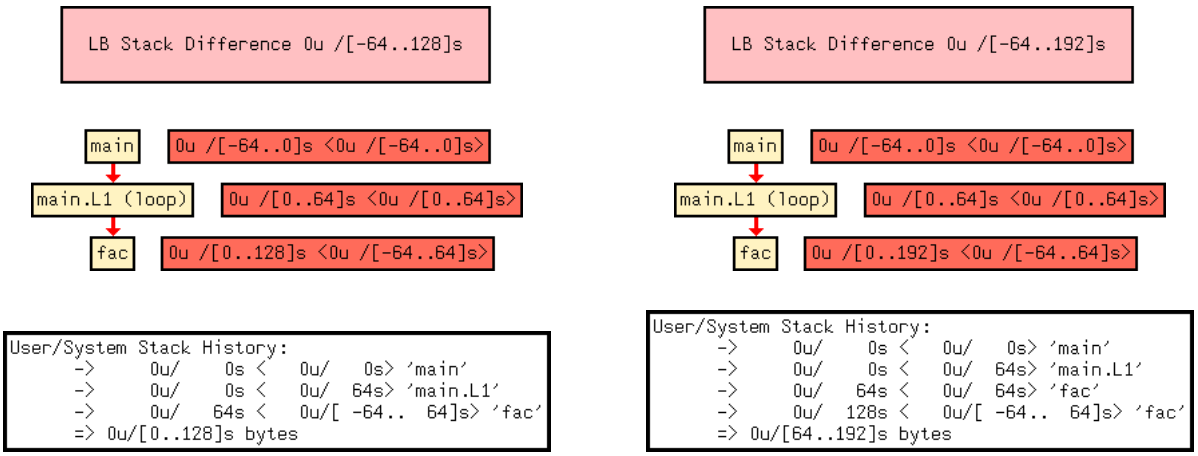


Figure 7.19: Recursive function with call depths 1 (left) and 2 (right)

on the left and

```
routine "fac" incarnates max 2;
```

on the right. The lower part of the figure shows the contents of the first info field of the result box next to routine `fac`. With bound 1 for `fac`, a^3 only considers call chains with at most 1 occurrence of `fac`. Since `fac` is called in `main` via `main.L1` at stack level `0u/64s` and has local stack result `0u/[−64..64]s`, the call chain `main` → `main.L1` → `fac` yields a global result of `0u/[0..128]s` for `fac`. With bound 2 for `fac`, a^3 considers call chains with at most 2 occurrences of `fac`. The call chain `main` → `main.L1` → `fac` → `fac` yields a global result of `0u/[64..192]s` shown in the info field. This result is joined with the result `0u/[0..128]s` for the call chain `main` → `main.L1` → `fac`, leading to the final result `0u/[0..192]s` shown in the annotated call graph. The same result is obtained with

```
routine "fac" calls to "fac" incarnates max 1;
```

which limits the number of occurrences of `fac` → `fac` to 1.

In general, an incarnation bound l for a routine R restricts the call chains considered by a^3 to those with at most l occurrences of R. If for instance A calls B and C, B calls A, and C calls A, unrestricted recursion can be avoided by a bound on A, or by bounds on B and on C.

Note that a^3 has no possibility to check whether a given bound is a true property of the application. If the user provides a bound that is not supported by specific measures in the code, the results of a^3 can only be used as hint what would be the stack usage if the bound were respected by the application. A bound `bound` > 0 for a directly recursive routine R can for instance be enforced by a static counter in R:

```
... R (...) {
  static int counter = 0;
  counter += 1;
  if (counter < bound) { ...;  R(...); ...; }
  counter -= 1;
}
```

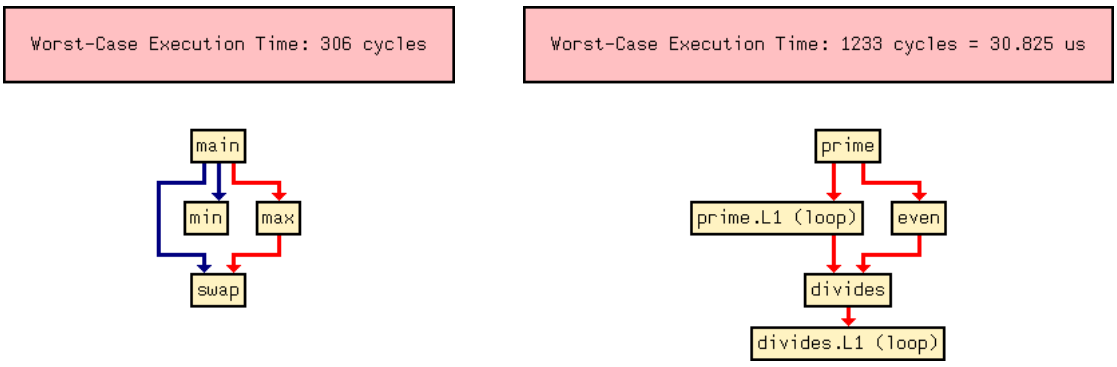


Figure 7.20: Graphs resulting from `Analyze`

7.4 Timing Analysis

When a start operation (`Analysis` → `Start` or `F7` or ▶) is performed in a timing-analysis view, a^3 performs a WCET analysis and then displays a combined call graph and control-flow graph annotated with the results of the analysis. Apart from the annotations, this graph looks like the graphs without analysis results described in section 7.1. In this section, we describe only the differences between the graphs with WCET annotations and the basic graphs of section 7.1.

7.4.1 Overall Worst-Case Execution Time

Graphs resulting from WCET analysis contain a special pink box at the top. This box tells the computed overall worst-case execution time of the analyzed program part in processor cycles, and in real time provided there is a clock rate specification – see section 5.2. Figure 7.20 shows two examples. The left example has no clock rate specification, while the right example includes a specification

```
clock exactly 40 MHz;
```

Call edges belonging to the WCET path computed by a^3 are red instead of dark blue (see section 7.4.5 for more details about the WCET path). For instance, routine `main` in the left example of Figure 7.20 contains a conditional that calls `min` in the then-part and `max` in the else-part. Since the else-part including `max` takes longer than the then-part including `min`, the call of `max` is on the WCET path, while the call of `min` is not. Routine `swap` is called several times; some calls are on the WCET path and some are not.

7.4.2 WCET Contributions and Contexts of Routines

Graphs with WCET results carry information on WCET contributions and contexts in the second info fields of routine nodes (see section 8.1.7 for accessing this information in **aiSee**). Examples are provided after the following description.

There are four kinds of information in these info fields:

- A `Predicted WCET Contribution` (in processor cycles, and in real time provided there is a clock rate specification – see section 5.2). This is not the WCET for one invocation of the routine, but the sum of the estimated WCETs of all routine invocations along the WCET path. It covers only the instructions of the routine itself, not those of the extracted loop routines or those of the routines called. The estimated WCET of the entire analyzed code as given in the pink box at the top is the sum of the predicted WCET contributions of all routines.

  Routines that are not on the WCET path (such as `min` in Figure 7.20, left) contribute nothing to the WCET; the `Predicted WCET Contribution` in their second info field is 0. These routines are only reachable via dark blue call edges. Routines on the WCET path are those that can be reached from the start point via a red call path.

- A `Cumulative` WCET contribution (in processor cycles, and in real time provided there is a clock rate specification – see section 5.2). Like the `Predicted WCET Contribution`, this is not the WCET for one invocation of the routine, but the sum of the estimated WCETs of all routine invocations along the WCET path. In contrast to the `Predicted WCET Contribution`, it covers not only the instructions of the routine itself, but also those of the extracted loop routines and those of the routines called.

  The calculation of cumulative WCETs is a license feature. Cumulative WCETs are not shown if they are not included in your license.

- The predicted WCET contribution is broken down according to the various routine contexts. In the resulting list, the contexts are identified by their number. A description of the contexts is provided in the last part of the info field (see below). For each context, the part of the WCET contribution belonging to that context is listed in processor cycles. Note that the contribution may be 0 for some contexts if some calls of the routine are not on the WCET path.

  The WCET contribution only refers to the routine itself, not to the extracted loop routines or to the routines called by that routine. Therefore, a *cumulative contribution* is added in parentheses that adds up the contributions of the routine itself and all routines called by it.

  If the call string length is restricted by the `max-length` parameter in the AIS file (see section 5.4) so that some call strings are pruned, the cumulative WCET contribution becomes unknown (written as `?`) for some routines.

- A list of *calling contexts* (see section 4.3 for a general introduction of this notion). The contexts depend on the context parameters specified in the AIS file (see section 5.4). For each context, the list displays the context number under the heading `Contexts`, an internal number under the heading `Position` (omitted in the examples below), and the call string of the context as introduced in section 4.3.

Example 1. The first example is the one whose call graph is depicted in Figure 7.20, left. Routine `main` calls `min`, `max`, and `swap`, and routine `max` contains two calls of `swap`. The example was run with `max-length=inf`, which means unrestricted call string length in contexts. There is no clock rate specification so that only cycle numbers are displayed.

Second info field of routine `main`:

```
Predicted WCET Contribution: 102 cycles
                  Cumulative: 306 cycles
Predicted WCET Contribution per Context (cumulative):
    Context   0: 102 cycles (306 cycles)
Contexts  Call String
       0  no-history
```

Note that the cumulative WCET of `main` equals the overall WCET of this example. It results as the sum of the WCET contribution of `main` itself (102 cycles) and of the cumulative contribution of `max` (204 cycles – see below).

Second info field of routine `min`:

```
Predicted WCET Contribution: 0
                  Cumulative: 0
Predicted WCET Contribution per Context (cumulative):
    Context   0: 0 cycles (0 cycles)
Contexts  Call String
       0  0xd400015d->"min"
```

This routine is not on the WCET path and thus has a WCET contribution of 0 cycles.

Second info field of routine `max`:

```
Predicted WCET Contribution: 92 cycles
                  Cumulative: 204 cycles
Predicted WCET Contribution per Context (cumulative):
    Context   0: 92 cycles (204 cycles)
Contexts  Call String
       0  0xd400016d->"max"
```

The cumulative WCET of `max` is its own WCET (92 cycles), plus the WCETs of the two calls of `swap` in `max` (56 + 56 cycles, see below).

Second info field of routine `swap`:

```
Predicted WCET Contribution: 112 cycles
                  Cumulative: 112 cycles
Predicted WCET Contribution per Context (cumulative):
    Context   0: 0 cycles (0 cycles)
    Context   1: 56 cycles (56 cycles)
    Context   2: 56 cycles (56 cycles)
Contexts  Call String
       0  0xd400014a->"swap"
       1  0xd400016d->"max", 0xd400011d->"swap"
       2  0xd400016d->"max", 0xd400012f->"swap"
```

The call of `swap` in `main` is not on the WCET path. Therefore the WCET contribution of `swap` in the corresponding context (context 0) is 0. This is also the reason for the blue color of the edge from `main` to `swap` in Figure 7.20, left. Since `swap` does not call any other routine, its cumulative WCET contribution equals its own WCET contribution in every context. The overall WCET-contribution of `swap` (112) is the sum of the contributions per context (56 + 56).

Example 2. The second example is the one whose call graph is depicted in Figure 7.20, right. Routine `prime` calls `even`, which calls `divides`. Routine `prime` contains a loop that is extracted as `prime.L1` by the loop transformation (see section 4.1). The loop contains a call of `divides`, which contains a compiler-generated loop extracted as `divides.L1`. Again, the example was run with `max-length=inf`, which means unrestricted call string length in contexts. The loop bound is 21, but the number of loop contexts was restricted by `max-unroll=3`, so that at most 3 loop contexts are distinguished for `prime.L1`: a context `[1/3..]` for the first iteration, a context `[2/3..]` for the second iteration, and a context `[3/3..]` for the remaining 19 iterations. In contrast to the first example, there is a clock rate specification of 40 MHz so that the WCETs are displayed in cycles and in real time units.

Second info field of routine `prime`:

```
Predicted WCET Contribution: 44 cycles = 1.1 us
                  Cumulative: 1100 cycles = 27.5 us
Predicted WCET Contribution per Context (cumulative):
    Context   0: 44 cycles (1100 cycles)
Contexts  Call String
       0  no-history
```

The cumulative WCET 1100 of `prime`, which equals the overall WCET of this example, is the sum of the non-cumulative WCET of `prime` (44), the cumulative WCET of even (46), and the cumulative WCET of the loop (1010, see below).

Second info field of routine `even`:

```
Predicted WCET Contribution: 3 cycles = 0.075 us
                  Cumulative: 46 cycles = 1.15 us
Predicted WCET Contribution per Context (cumulative):
    Context   0: 3 cycles (46 cycles)
Contexts  Call String
       0  0x8000002c->"even"
```

The cumulative WCET 46 of `even` is the sum of its non-cumulative WCET (3) and the cumulative WCET of `divides` in context 0 (43), which is the context corresponding to the call of `divides` from `even` (see below).

Second info field of routine `prime.L1`:

```
Predicted WCET Contribution: 317 cycles = 7.925 us
                  Cumulative: 1010 cycles = 25.25 us
Predicted WCET Contribution per Context (cumulative):
```

```
    Context   0: 10 cycles (1010 cycles)
    Context   1: 22 cycles (967 cycles)
    Context   2: 285 cycles (912 cycles)
Contexts  Call String
       0  0x8000003a->"prime.L1"[1/3..]
       1  0x8000003a->"prime.L1"[2/3..]
       2  0x8000003a->"prime.L1"[3/3..]
```

As explained above, the loop has 3 contexts, of which contexts 0 and 1 correspond to the first 2 iterations, while context 2 corresponds to the remaining 19 iterations. It turned out that the first iteration takes 10 cycles, the second iteration takes 22 cycles, while all other ones take 15 cycles. This leads to the non-cumulative WCET $19 \cdot 15 = 285$ in context 2. Since loops are analyzed as recursive routines (see section 4.1), the cumulative WCET of each iteration includes the WCETs of the later iterations and the cumulative WCETs of the corresponding calls of `divides`. Thus, the cumulative WCET 912 in context 2 results from the non-cumulative WCET 285 and the cumulative WCET of `divides` in the corresponding context 3, which is 627 (see below). The cumulative WCET 967 in context 1 results from adding the non-cumulative WCET (22) and the cumulative context-2-WCET of `divides` (33) to the cumulative WCET of context 2 (912). The overall non-cumulative WCET (predicted WCET contribution) of the loop (317) results from adding the non-cumulative WCETs in the various contexts ($10 + 22 + 285$).

Second info field of routine `divides`:

```
Predicted WCET Contribution: 384 cycles = 9.6 us
                 Cumulative: 736 cycles = 18.4 us
Predicted WCET Contribution per Context (cumulative):
    Context   0: 27 cycles (43 cycles)
    Context   1: 17 cycles (33 cycles)
    Context   2: 17 cycles (33 cycles)
    Context   3: 323 cycles (627 cycles)
Contexts  Call String
       0  0x8000002c->"even", 0x80000024->"divides"
       1  0x8000003a->"prime.L1"[1/3..], 0x80000040->"divides"
       2  0x8000003a->"prime.L1"[2/3..], 0x80000040->"divides"
       3  0x8000003a->"prime.L1"[3/3..], 0x80000040->"divides"
```

This routine has 4 contexts. Contexts 1–3 correspond to contexts 0–2 of the loop, while the remaining context 0 corresponds to the call of `divides` in `even`. It turned out that that call, which is the first call of `divides`, takes 27 cycles, while all other calls take only 17 cycles. The WCET 323 in context 3 adds up the times of 19 calls of `divides` ($19 \cdot 17 = 323$) since context `[3/3..]` of the loop corresponds to 19 loop iterations. The cumulative WCETs of `divides` equal the non-cumulative ones plus the WCETs of `divides.L1`. The overall non-cumulative WCET (predicted WCET contribution) of `divides` (384) results from adding the non-cumulative WCETs in the various contexts ($27 + 17 + 17 + 323$).

The distribution of WCETs among the various loop iterations is not always as regular as in the examples presented above. As explained in section 7.4.3, the WCET contribution for one

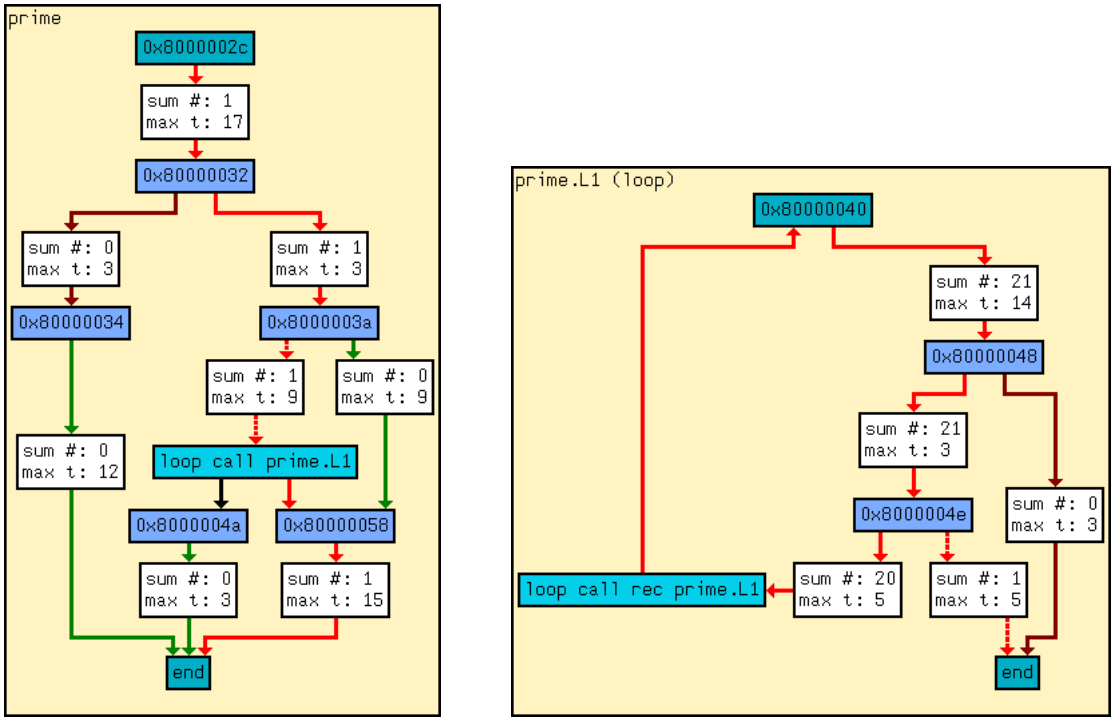


Figure 7.21: Basic block graphs resulting from `Analyze`

iteration may be 0 because it is added to another iteration. This seemingly strange behavior does not affect the overall WCET, but only its distribution among the various contexts. It may also happen that only part of the WCET contribution of one iteration is accounted in another iteration.

7.4.3 WCET Information for Basic Blocks

Figure 7.21 shows the basic block graphs of an ordinary routine (left) and of a loop routine (right). Most edges are annotated by a white box displaying two numbers. The meaning of these numbers becomes clearer when looking at the second info fields of these white boxes. For instance, the second info field of the white box at the outgoing edge of the loop-entry node `0x80000040` contains under the heading **`Details`** the lines

```
0: cycles=2, count=1
1: cycles=14, count=1
2: cycles=7, count=19
```

The numbers in these lines refer to the three contexts of the loop routine, namely

```
Context 0: 0x8000003a->"prime.L1"[1/3..]
Context 1: 0x8000003a->"prime.L1"[2/3..]
Context 2: 0x8000003a->"prime.L1"[3/3..]
```

This example was run with `max-unroll = 3` so that three loop contexts are distinguished: contexts 0 and 1 for the first two iterations and context 2 for the remaining 19 iterations. In general, the second info field of a white box at an edge e displays two numbers for each context c:

- the worst-case traversal `count` $C(e,c)$ for edge e and context c as determined by path analysis (see section 4.4.4),
- and the estimated worst-case execution time in `cycles` for edge e and context c as determined by the combined cache and pipeline analysis (see section 4.4.3). This is the estimated WCET of the instructions in the basic block at the beginning of e provided that the block is left via e. Loop call blocks like `loop call prime.L1` and `loop call rec prime.L1` do not contain instructions by themselves and therefore have no white boxes at their outgoing edges.

In the second info fields of the white boxes, contexts are identified by their number. A description of the contexts can be seen in the second info field of the surrounding routine.

The information in a white box itself is derived from the information in its second info field as follows:

- `sum #` indicates the sum of the `count` values, taken over all contexts.
- `max t` indicates the maximum of the `cycles` values, taken over all contexts.

Consider again the white box below block `0x80000040` in Figure 7.21. Part of the info field of this box has been presented above. It shows that the execution count in contexts 0 and 1 is 1, while the count in context 2 is 19. Thus, the `sum #` entry in the white box is the sum of 1, 1, and 19, which is 21. The execution time is 2 in context 0, 14 in context 1, and 7 in context 2. Thus, the `max t` entry in the white box is the maximum of 2, 14, and 7, which is 14.

Since the first two contexts in the example above correspond to the first two loop iterations and the third context to the remaining 19 iterations, it seems very natural that the execution counts in the three contexts are 1, 1, 19. Thus it is a bit surprising that the execution counts assigned to the three contexts may also be 1, 0, 20, or even 0, 0, 21. The reason is that the counts are determined by path analysis using integer linear programming (see section 4.4.4). If the execution times in contexts 1 and 2 were identical, the count distributions 1, 1, 19 and 1, 0, 20 would yield the same WCET, and path analysis would choose some arbitrary distribution.

Figure 7.22 shows another example. This program involves a loop

```
for (i = 1; i <= 10; i++)
```

The AIS file for this example contains the correct, but quite imprecise specification

```
loop "main" + 1 loop min 0 max 11;
```

The example was run with context specification

```
interproc flexible, max-length = inf;
```

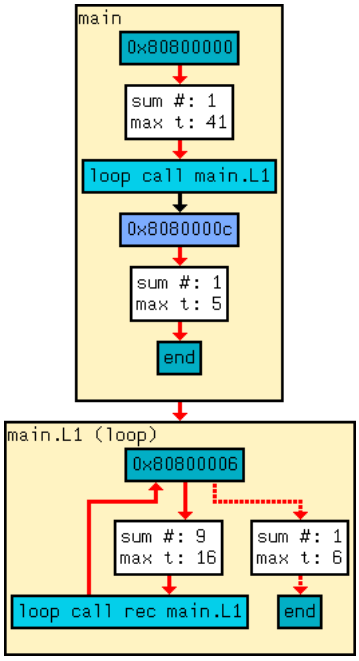


Figure 7.22: Example with infeasible paths

(see section 5.4) so that the number of loop contexts is derived from the specification in the AIS file. Since the position of the loop head is not specified there, a^3 assumes the worst case, which is test at the beginning. Therefore, the specification in the AIS file implies that the loop test is executed at most 12 times, and thus, 12 loop contexts should be distinguished, but a^3 finds out that the last two contexts are infeasible, i.e. do not correspond to any loop iteration. These contexts are suppressed so that there are only 10 contexts, which are numbered 0–9:

```
Context   0: 0x80800000->"main.L1"[1/12]
Context   1: 0x80800000->"main.L1"[2/12]
Context   2: 0x80800000->"main.L1"[3/12]
Context   3: 0x80800000->"main.L1"[4/12]
Context   4: 0x80800000->"main.L1"[5/12]
Context   5: 0x80800000->"main.L1"[6/12]
Context   6: 0x80800000->"main.L1"[7/12]
Context   7: 0x80800000->"main.L1"[8/12]
Context   8: 0x80800000->"main.L1"[9/12]
Context   9: 0x80800000->"main.L1"[10/12]
```

In reality, the loop test is at the end, namely at the end of the first and only block of the loop.

Value analysis is able to determine the value of `i` in each iteration since each iteration is performed in a new context. Hence it can predict that block `0x80800006` (the loop test) is left via the left edge leading to the next iteration in the first 9 executions of the test (contexts 0–8), but is left via the right edge leading to the block after the loop in the 10th and last execution (context 9). The left white box in `main.L1` therefore lists the following **`Details`** in its second info field:

```
0: cycles=13, count=1
```

```
        1: cycles=16, count=1
        2: cycles=14, count=1
3-      8: cycles=6, count=1
        9: infeasible
```

The right white box lists the following:

```
0-      8: infeasible
        9: cycles=6, count=1
```

Note that adjacent lines with the same information are collapsed into a single line tagged by a range of contexts.

Besides the **Details** information as presented above, the second info field of the white boxes contains some summary information under the heading **Statistics**. The **Statistics** information of the second info fields of the two white boxes in routine `main.L1` looks as follows (with some parts replaced by "`...`"):

```
Statistics:                              | Statistics:
  Number of contexts:                 10 |   ...:   10
  Number of infeasible contexts:       1 |   ...:    9
  Common infeasible contexts:          0 |   ...:    0
  Additional infeasible contexts:      1 |   ...:    9
  Total execution count:               9 |   ...:    1
  Total contribution (cycles):        79 |   ...:    6
  Maximum count over contexts:         1 |   ...:    1
  Maximum execution over contexts:    16 |   ...:    6
```

The **Statistics** part lists the following information:

- **Number of contexts** indicates the number of contexts of the routine in which the white box occurs. In the example, the number is 10 for both white boxes, which are in the same routine `main.L1`.
- **Number of infeasible contexts** indicates the number of contexts for which value analysis has found that the edge through the white box is infeasible, i.e. never taken as a control path. In the example, the number is 1 for the left box that is infeasible in context 9, and 9 for the right box that is infeasible in contexts 0–8.
- **Common infeasible contexts** indicates the number of contexts in which both edges are infeasible. In the example, this number is 0 since the contexts in which both boxes are infeasible have been suppressed.

  This information appears only if several different edges emanate from a single basic block *B* (block `0x80800006` in this example). The displayed number is then the number of contexts in which all the edges emanating from *B* are infeasible, i.e. the number of contexts in which *B* itself is infeasible.

- **`Additional infeasible contexts`** is the difference between the `Number of infeasible contexts` and the number of `Common infeasible contexts`. Like the latter, it appears only if several different edges emanate from a single basic block.
- **`Total execution count`** is the sum of the execution counts over all contexts. This is the same as the number `sum #` displayed in the info field itself. In the left box, this sum is 9 since the `count` is 1 in contexts 0–8.
- **`Total contribution (cycles)`** is the sum of the product of `count` and `cycles` over all contexts. In the left box, this sum is $13+16+14+6\cdot 6=79$. This number can be understood as the contribution of the edge through the white box to the overall WCET. (In contrast, the number `max t` displayed in the info field itself is the *maximum* of the number of cycles in the various contexts, without considering the execution counts.)
- **`Maximum count over contexts`** is the maximum of the execution counts over all contexts. In the example, this is 1 for both boxes.
- **`Maximum execution over contexts`** is the maximum of `cycles` over all contexts. This is the same as the number `max t` displayed in the info field itself. In the left box, this is the maximum of 13, 16, 14, and 6, which is 16.

7.4.4 Loop Count Information

The results of automatic loop bound detection and the user-specified loop bounds appear in the first info fields of loop call nodes of the form `loop call` *R*`.L`*n*. These are the loop call nodes that mark the place where the loop has been extracted by the loop transformation. See section 8.1.7 for accessing info fields in **aiSee**.

Example 1: If the first info field of `loop call prime.L1` in Figure 7.21 were opened, you would see

```
User specified bounds:
 iteration-min: 1
 iteration-max: 21

Analyzed bounds:
 [?] in context 0: no-history
```

Automatic loop bound detection did not obtain any useful result in this example.

For this example, the specification in the AIS file has been

```
loop "prime" + 1 loop min 0 max 20 begin;
```

claiming that the loop test is at the beginning of the loop. Nevertheless, the info field gives bounds of 1 and 21. The reason is that the numbers in the AIS file refer to executions of the loop body, while the numbers in the info field refer to executions of the loop test. If the loop test is at the beginning, it is executed once more than the loop body. If the AIS file had contained the specification

```
loop "prime" + 1 loop min 0 max 20 end;
```

claiming that the loop test is at the end of the loop, the info field would have contained bounds of 0 and 20.

Example 2: The first info field of `loop call main.L1` in Figure 7.22 contains:

```
User specified bounds:
 iteration-min: 0
 iteration-max: 12

Analyzed bounds:
 [10] in all contexts
```

Here the user specified

```
loop "main" + 1 loop min 0 max 11;
```

which applies to the loop body. Since the user did not specify whether the loop test is at the beginning or the end, the iteration bounds 0–11 for the loop body lead to bounds 0–12 for the number of executions of the loop test. Automatic loop bound detection was successful for this example. It found that the loop test is executed exactly 10 times. This result is valid for all contexts of `main` (in fact, there is only one in this example). In general, automatic loop bound detection may find one result for each context of the routine containing the loop.

Note that user annotations supersede the results of loop bound detection unless they are default annotations. Therefore, the analyzed bound of 10 in the example above gets used only if the imprecise user annotation is deleted or tagged with `by default`.

7.4.5 The Worst-Case Execution Path

Path analysis determines a worst-case traversal count for almost every edge in each context (see section 4.4.4). These traversal counts are contained in the second info fields of the white boxes in the basic block graphs (see section 7.4.3). The white boxes themselves contain the sum of the execution counts taken over all contexts, labeled by `sum #` (see Figures 7.21 and 7.22).

All edges with a non-zero worst-case traversal count are colored light red. Those which correspond to “false” edges are dotted to distinguish them from the “true” and normal edges. Often, the light red edges form a worst-case path through the basic block graph. This path need not be the *real* worst-case path because it might be infeasible without being recognized as such by a^3. Consider for instance code such as

```
if (x > 0) b1;  b2;  if (x < 0) b3;
```

where `x` may hold any positive or negative value depending on the inputs and is not modified in any of the three blocks. Then the path `b1` → `b2` → `b3` that includes both `b1` and `b3` is infeasible because `b1` and `b3` cannot be executed together in the same run through this code snippet. Yet

each of the three blocks is feasible since there are runs that execute `b1` and other runs that execute `b3`. Path analysis does not track conditions and will display the infeasible path `b1` → `b2` → `b3` as worst-case execution path. Despite this behavior, the WCET computed by a^3 is a safe approximation of the real WCET since the infeasible path `b1` → `b2` → `b3` needs more time than the feasible paths `b1` → `b2` and `b2` → `b3`. Thus, the computed WCET is still an upper bound of the real WCET. Taking into account infeasible paths only reduces the precision of the analysis, i.e. increases the difference between computed WCET and real WCET.

In the discussion above, we assumed that the light red edges form a path. Yet in reality, they may fail to do so. As explained in section 4.3, each routine is analyzed several times, once for each context. Sometimes, path analysis assigns the same execution counts in all contexts, but it may as well happen that execution counts are 0 in one context and non-zero in another, i.e. the computed worst-case paths depend on the context. An edge is colored light red if its execution count is non-zero in at least one context. Hence, the collection of light red edges does not represent a single worst-case path in general, but the union of the worst-case paths of the various contexts. This applies in particular to loop routines where usually almost every edge is light red. In loops, you may also consider the light red edges as a single path that loops around so that nearly every edge belongs to it. In ordinary routines, the worst-case path is usually more visually recognizable, e.g., in Figure 7.21, left. Call edges are colored light red if at least one of the calls they represent has a non-zero worst-case traversal count, i.e. lies on the worst-case path. The remaining call edges are still dark blue (see for instance Figure 7.20, left).

7.5 Graphs with Cache and Pipeline States

When an "interactive" operation (`Analysis` → `Interactive` or) is issued in a timing-analysis view, a^3 performs a WCET analysis and then displays a combined call graph and control-flow graph that looks like the graphs with WCET results described in section 7.4, but additionally provides access to the cache and pipeline states created by the analysis (and to the values in registers and memory cells by pressing `v`, see section 7.2).

When you select a basic block with the left mouse button and press the **`p`** key, a^3 opens a context selection box displaying all contexts for the basic block selected. When a context is selected and the **`Ok`** button of the context selection box is clicked, a^3 opens a new graph-browser view showing the cache and pipeline states of the selected basic block for the selected context (this may take some time).

The collection of possible cache and pipeline states for a basic block and a context is displayed as a graph. Figure 7.23 shows such a state graph, in low magnification. The large gray rectangles correspond to the instructions of the basic block the state graph belongs to. The smaller rectangles with a colored frame are individual states. A red frame indicates start states (relative to an instruction), a green frame indicates end states, states with a blue frame are states in-between, and states in a white frame do not belong to the control-flow considered and must be deleted. The end states of an instruction correspond to the start states of the next instruction.

The cyclewise evolution of the states is indicated by the arrows connecting them. Each layer in the picture corresponds to one CPU cycle. Branches in the trees are caused by case distinctions that cannot be statically evaluated. On the other hand, two states fall together when details they differ

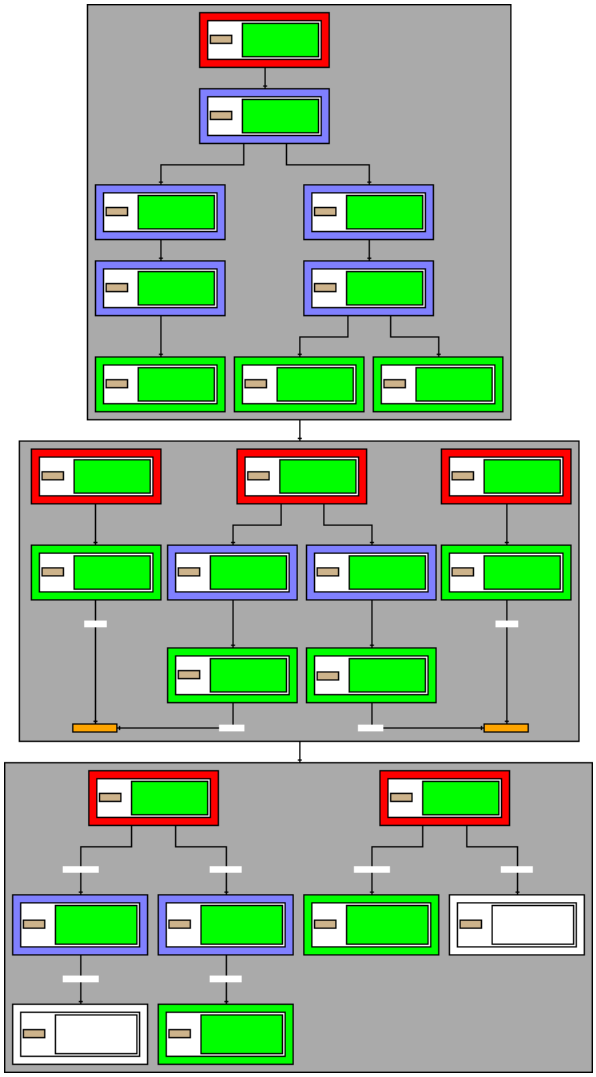


Figure 7.23: Analyzer states

in leave the pipeline. This happened for instance at the end of the second instruction. Two pairs of end states are identical and fall together, which is indicated by the arrows to the orange boxes that stand for the results of the identification. Consequently, the third instruction has only two start states, which again split into two successor states each.

Figure 7.24 shows the lower left part of the second gray box from Figure 7.23 in greater magnification. The two states with the green frame are – except for the cycle count – identical with each other. These two states fall together into one, which is indicated by the arrows labeled by `subsume` and `subsume2` leading to an orange box representing the result of the identification. This orange box is labeled by the maximum of the cycle counts of all states identified in it.

Because of the identification, the last gray box starts with two states (those with a red frame in Figure 7.23). In the control flow graph, the corresponding instruction node has two successors, a "true" successor and a "false" successor. Accordingly, each state splits now into two, one indicating what happens if the "false" edge is followed, and one for the "true" edge. Yet in each case, only one of the successors is compatible with the prefetch decision taken previously. The incompatible

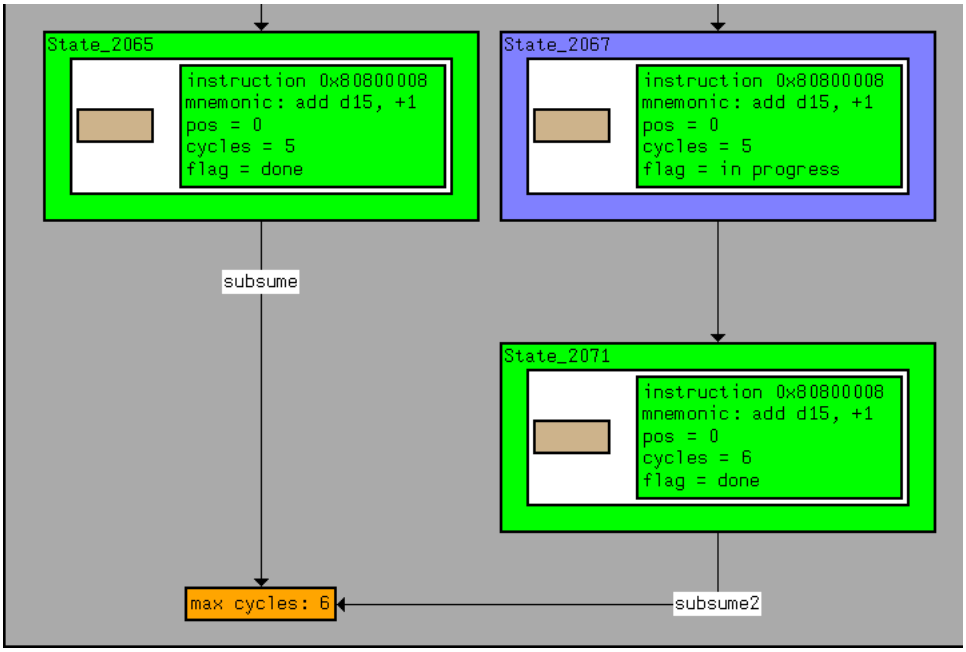


Figure 7.24: State subsumption

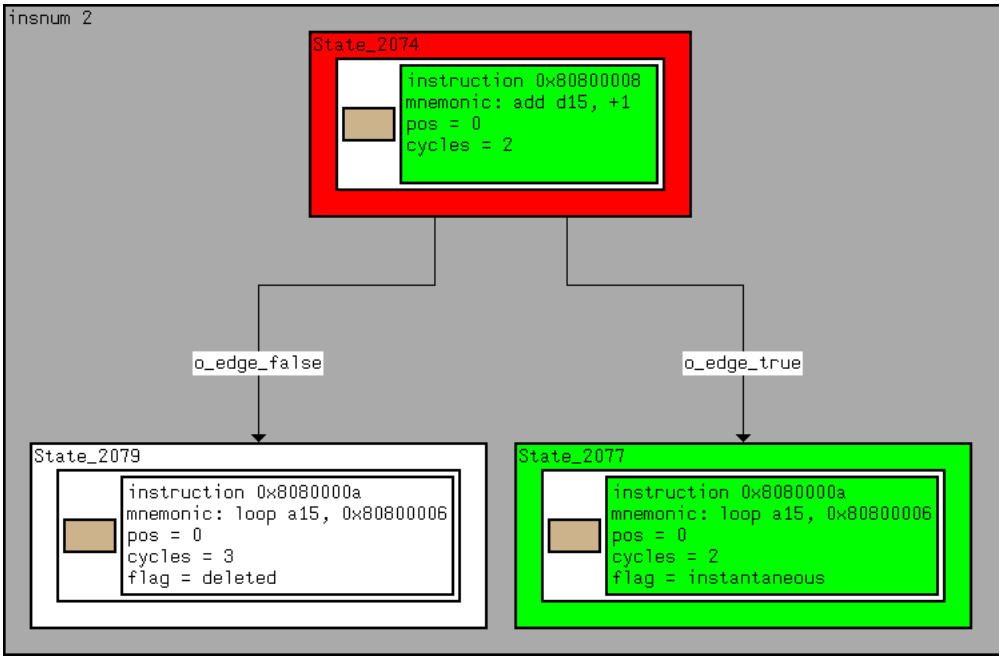


Figure 7.25: Analyzer states in larger magnification

ones are deleted (white), while the compatible ones are end states (green frame) to be propagated to the next basic block (the “true” successor and the “false” successor respectively).

Figure 7.25 shows such a situation in larger magnification so that the text panels included in the states become readable. Such text panels can also be seen in Figure 7.24. These text panels contain the following information:

- **`instruction`** *address* indicates the address of the current instruction, i.e. the instruction that is being retired at the end of the surrounding gray box. Initial states form an exception: they usually still display the address of the previous instruction.

- **`mnemomic:`** *code* shows the disassembly of the instruction at the address displayed one line before.

- **`cycles =`** *number* counts the cycles from the beginning of the basic block.

- **`flag =`** ... presents the kind of the state (as also indicated by the frame color). Initial states (red frame) do not display the flag, intermediate states (blue frame) are labeled `in progress`, incompatible states (white) are labeled `deleted` because they are removed from the set of states without a successor state, and final states (green frame) are usually labeled as `done`.

  There is another possible label for final states. Since the processor may retire several instructions at once, it may happen that an initial state is already final since the current instruction has already been retired together with the previous one. In this case, a^3 still draws two different states ⚠, but the final state is marked by the flag `instantaneous`, and the `cycles` counter is not incremented, but has the same value in the two states (see Figure 7.25).

Chapter 8

The aiSee Graph Viewer

aiSee is **AbsInt**'s graph viewer. It reads a graph description written in GDL (Graph Description Language), computes a layout for this graph, and displays this layout on the screen.

In a^3 **aiSee** graphs are embedded as views. Whenever a control flow graph or the results of a successful analysis are visualized, a new view is appended to the **`Graphs`** section in the list of available views. By selecting one of the available **`Graphs`** views, the graph is displayed in the right part of the a^3 window; menu and tool bar are extended by several entries for accessing **aiSee** functions.

When the detach button in the right upper corner of a graph view is clicked, the graph view is detached into a separate window. When such a detached window is closed, the graph displayed in it returns into the main a^3 window as a graph view. Closing a graph view completely is only possible in its non-detached state by clicking the close button .

The following sections contain an introduction to the operations of **aiSee**: how to navigate through a graph and how to access initially hidden information. The description is biased towards the browsing of graphs generated by a^3. For detailed information and more advanced features, consult the separate **aiSee** manual.

If you are used to the old version **2** of **aiSee**, keep in mind that this chapter solely addresses the new version **3**.

The examples in this chapter are generic, i.e. not specific to the TriCore architecture.

8.1 aiSee Basics

8.1.1 Command Invocation

Most **aiSee** commands can be invoked in three ways: by hitting a key on the keyboard, by clicking a tool bar button, or by invoking a command from a menu. Furthermore, some commands are accessible from the context menu.

8.1.2 Operation Modes

aiSee has several operation modes for collecting operands or additional information: in *Selection Mode* you can select a node, several nodes, or a graph area; in *Zoom Mode* you can zoom in and out; finally, in *Pan Mode* you can drag the graph.

You can enter different modes either by using the tool bar or the `Mode` menu. For details refer to sections 8.1.3.2 and 8.1.4 below.

8.1.3 Scrolling and Scaling

Most graphs are too large to be displayed completely (unless the nodes are displayed extremely small). Therefore, **aiSee** usually shows only some part of the graph. This part can be scrolled, and scaled up or down by zoom operations.

8.1.3.1 Scrolling the Graph

A graph can be scrolled by hitting the *arrow keys* ↑, ↓, ←, and →. For faster vertical scrolling you can also use the **`PgUp`** and **`PgDn`** keys. Vertical scrolling can also be achieved by turning the mouse wheel (if available), and arbitrary scrolling by moving the mouse while holding down the middle mouse button (if available). When the letter **`o`** is hit, the graph is scrolled so that its "origin" (a fixed point somewhere near the upper left corner of the graph) sits in the upper left corner of the graph window. All these operations work in any mode.

If there is no middle mouse button available, you can still scroll the graph with the mouse by entering *Pan Mode*: click on the button in the tool bar or select **`Mode`** → **`Pan`** in the menu. Then you can scroll by moving the mouse while the left mouse button is pressed.

8.1.3.2 Scaling the Graph

For scaling the graph you can use the following commands:

- **`+`** or magnifies (scales up) by a factor of 1.41.
- **`-`** or reduces (scales down) by a factor of 0.7.
- **`0`** (zero) or scales to some fixed medium magnification.
- **`m`** or scales and scrolls so that the whole graph fits in the window (approximately).
- **`h`** or scales and scrolls so that the whole graph vertically fits in the window (approximately).
- **`w`** or scales and scrolls so that the whole graph horizontally fits in the window (approximately).

The commands **+**, **-**, and **0** (zero) scale the distances from a point near the center of the graph window. All those commands can also be invoked from the `View` menu.

While all the previous commands can be used in any mode, you can also enter the *Zoom Mode*: either by clicking on the button in the tool bar or by choosing **Mode** → **Zoom** in the menu; then click with the left mouse button anywhere in the graph to scale up or hold additionally the **Shift** key pressed to scale down. If you move the mouse while its left button is pressed, you can select a rectangular region of the graph. When the left mouse button is released, the graph is scrolled and magnified so that the selected area extends through the entire **aiSee** window.

8.1.4 Selection of Graph Elements

In order to select certain elements from the graph, you first need to enter one of the selection modes. Depending on whether you want to select only nodes, only graphs, or both, choose the corresponding entry in the **Mode** → **Select** menu or click on the corresponding button in the family in the tool bar. Elements can then be selected by *left-clicking* on them. Usually, selected elements are *highlighted*, that is, are shown in a kind of inverse video mode.

By holding the **Shift** key down while clicking on nodes, you can select or deselect several elements. When **Shift** is held down, each left-click on a node toggles between selection and non-selection. Another way to select several nodes is to move the mouse while the left mouse button is held down. This draws a rectangle and selects all nodes met by the rectangle. The key combination **Ctrl+A** selects *all* graph elements.

All selections are undone at once by pressing **Esc**.

8.1.5 Show Panner

With the `Show panner` function you can create an overview of the whole graph. By pressing **s**, clicking on , or selecting **View** → **Show panner** in the menu the graph is magnified in such a way that it is completely visible and a rectangle can be moved with the mouse to focus a certain area of the graph; in the window this region is lighted. By clicking the left mouse button, the corresponding part of the graph is magnified such that it fills the entire window. To exit this function without modifying the current viewing perspective press **Esc**.

8.1.6 Resetting aiSee

Sometimes it is desired to read the input file again and return to the initial viewing perspective of the graph. This can be done by clicking on the button or by selecting **Edit** → **Reload** in the menu.

8.1.7 Accessing Info Fields

Nodes in GDL graphs can be associated with up to three information fields. To access them, first select the nodes whose fields you want to show. Then press **i**, click on the button in the tool

bar, select **View** → **Show info** in the menu, or choose **Show info** from the context menu. A subwindow will pop up; in its left upper corner, there are numbered buttons to switch between the three information fields. Buttons leading to empty information fields are inactive. The texts in the info fields are shown in normal size no matter how the graph is scaled.

By proceeding as above you can open several information windows. When you move the mouse over one of them, the node to which the current information belongs will be highlighted.

When several information fields are opened, a huge part of the graph might be covered. To make the whole graph visible again, you can switch on / off the display of the information subwindows. To this end press **Ctrl+i**, select **View** → **Info fields visible**, or click on the button in the tool bar. Repeatedly doing this toggles between showing and hiding the opened information boxes.

8.2 Exposing and Hiding Subgraphs

8.2.1 The Hierarchy Levels of Control-flow Graphs

The control-flow graphs generated by a^3 are structured in three hierarchical levels:

1. The nodes of the call graph represent routines. All routines except for externally defined ones and those declared not-analyzed contain basic block graphs, which are initially hidden, but can be exposed by **aiSee** commands.

2. The nodes of the basic block graphs represent basic blocks. All basic blocks except for the exit blocks and the loop-call blocks introduced by the loop transformation contain instruction sequences, which are initially hidden, but can be exposed by **aiSee** commands.

3. The nodes in the instruction sequences represent assembly instructions.

From the point of view of **aiSee**, the relationship between routines and basic block graphs is the same as the relationship between basic blocks and instruction sequences, and the same **aiSee** commands can be used at both levels. Put in more general terms, this means that **aiSee** supports the concept of graphs whose nodes may contain *subgraphs*, whose nodes in turn may contain subgraphs, etc.

8.2.2 Visualizing Subgraphs

There are several different ways to visualize a subgraph:

- **Folded**
  The subgraph is shown as a single node called a *summary node*. The nodes and edges of the subgraph are not visible.

  In the call graph that is initially shown by **aiSee**, all subgraphs are folded as in Figure 8.1 (unless the `Boxed routines` flag in the `Visualization` view is activated – see sec-

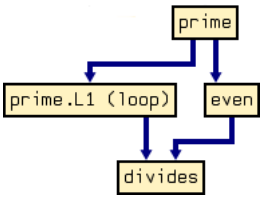


Figure 8.1: Graph with folded subgraphs

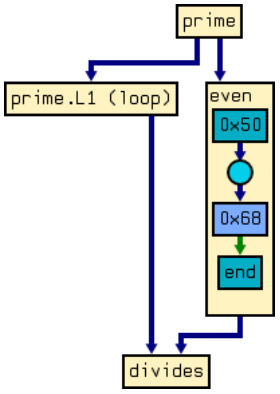


Figure 8.2: Call graph with boxed routine `even`

tion 3.4.6). The yellow nodes representing analyzed routines are summary nodes that stand for basic block graphs whose inner structure is hidden.

- **Boxed**
  The nodes and edges of the subgraph are visible. They are enclosed in a box which represents the subgraph as a whole. Figure 8.2 shows an example. In the boxed state, all edges that connect nodes of the subgraph with its environment begin or end at the enclosing box in the visualization. This applies in particular to the call edges (blue arrows) so that it is not apparent which basic blocks contain which calls if the basic block graph is in the boxed representation.

- **Clustered**
  As in the case of boxed representation, the structure of the subgraph is visualized in an enclosing box, but now edges connecting a node of the subgraph with its environment begin or end at the subgraph node and cross the border of the enclosing box (see Figure 8.3). In the case of a basic block graph for instance, the clustered representation reveals which basic blocks call which routines.

  ⚠ The clustered representation is an *experimental* feature. The layout of clustered subgraphs is quite challenging. **aiSee** includes an experimental implementation of clustered layouts which is not supported or maintained. Nevertheless, many **aiSee** users find this feature quite useful.

- **Unfolded**
  In this representation, there is no enclosing box (see Figure 8.4). The subgraph is not only freely embedded in its environment, worse yet, the nodes of the subgraph are not necessarily

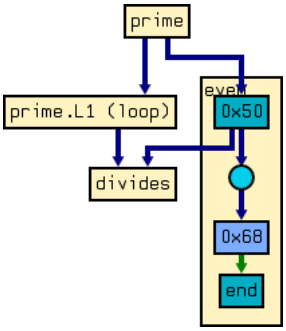


Figure 8.3: Call graph with clustered routine `even`

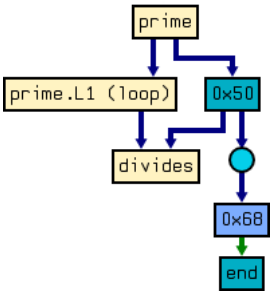


Figure 8.4: Call graph with unfolded routine `even`

close to each other; there may be other nodes in-between. If several different subgraphs are shown unfolded, it is no longer apparent which nodes belong to which subgraph.

8.2.3 Switching between the Visual Representations of Subgraphs

When **aiSee** starts, all subgraphs are folded (unless the `Boxed routines` flag in the `Visualization` view is activated; refer to section 3.4.6). To expose a subgraph, first switch to *Selection Mode* and select its summary node. Then invoke any of the following commands:

- `Unfold into box:` press **b** or click in the tool bar.
- `Unfold into cluster:` press **c** or click in the tool bar.
- `Unfold:` press **u** or click in the tool bar.

A subgraph can be folded again by first selecting any node belonging to it and then pressing **f** or clicking in the tool bar.

Notice that if you invoke the `Unfold into box` command on a routine, only basic blocks of the routine will be exposed in a box; yet, all basic blocks of the routine are folded (unless the `Boxed basic blocks` flag in the `Visualization` view is activated; see section 3.4.6). To also unfold all basic blocks of the routine in one step, you can use the `Unfold into box recursive` function by pressing **Shift+B** or selecting from the tool bar. Analogous recursive versions are also available for the other commands.

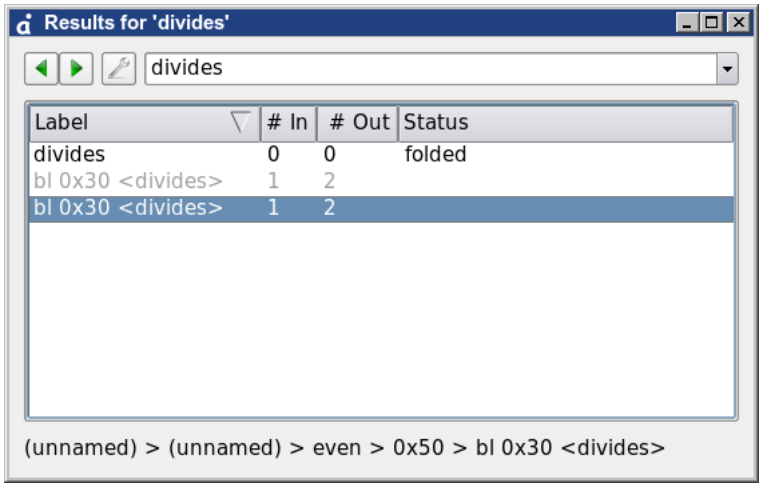


Figure 8.5: Window for searching nodes

All the above functions can also be accessed by selecting the corresponding entry in the `Folding` menu or `Folding` context menu.

If more than one node or subgraph in the graph is selected, any fold or unfold operation refers to all subgraphs which contain one of the selected nodes or to all nodes which are contained in one of the selected subgraphs.

In principle, it should be possible to switch from any visual representation to any other using the command belonging to the target representation. However, this is not always possible since sometimes there is no way to achieve the required selection.

- To put a subgraph into a boxed, clustered or unfolded state, the corresponding summary node must be selected. If the subgraph is currently folded, this is easily done by left-clicking on the summary node. (This means move the mouse pointer into the node and then press the left mouse button.) If the subgraph is currently boxed or clustered, it is selected by left-clicking into the surrounding box outside any node of the subgraph. If the subgraph is currently unfolded, there is no way to select it for boxing or clustering. You have to fold it first to get a summary node that can be selected.

- To fold a subgraph into a summary node, select any of its nodes and then invoke the `Fold` command as described above.

8.3 Find Nodes

Control-flow graphs may be huge. Therefore, **aiSee** provides a `Find Node` function. It can be invoked by pressing **`Ctrl+f`**, clicking in the tool bar, or choosing **`Edit`** → **`Find node...`** from the menu. After that, an auxiliary window for node search will pop up (see Figure 8.5). In this window you can supply a search string in the upper input field. Figure 8.5 shows the result of a search for the string `divides`. The result is given as a list of nodes. For every retrieved node, the list shows its label, the number of ingoing and outgoing edges, and its current status (for summary

nodes *boxed*, *clustered*, or *folded*). Labels of nodes that are already visible in the graph will be shown in black, others in gray. In the example, all routines are folded so that only routine nodes are visible.

The search performed on the graph is incremental, that is, while you type, the node list will dynamically change in such a way that it contains the labels of all those nodes that match against the current search string (see below).

When the find box is opened for the first time, it does not show anything. A list of nodes appears only after the first character is written into the input field.

Note that depending on the graph and the search string the list of nodes can be huge since it contains all nodes from all hierarchy levels (routines, basic blocks including exit blocks with their label `end`, and instructions). If the graph is annotated with stack analysis results, the node list even contains the auxiliary result nodes.

If a node is selected in the list of search results, its position in the hierarchy is shown in the baseline of the find box. In Figure 8.5 for instance, the selected node `bl 0x30 <divides>` is part of block `0x50`, which in turn is part of routine `even`. (`(unnamed)` refers to invisible nodes without labels at the top of the hierarchy.) The node selected in the result list is also selected in the graph picture provided that it is exposed in the graph. If it is hidden because of folding, the summary node of the subgraph containing it is selected instead. The graph picture is scrolled so that the selected node is displayed in the center of the graph window. The search-result list offers a context menu accessible by clicking with the right mouse button on a node. The context menu includes the following operations:

- `Expose and box node:` The selected node in the graph picture is unfolded into a box. This operation is applied once, so maybe the selected node from the result list is still not visible. The same effect can be achieved by pressing the **b** key in the graph view.
- `Expose and recursive box node:` Repeats unfolding all the subgraphs that contain the selected node from the result list until that node is completely visible. The same effect can be achieved by pressing **Shift+B** in the graph view.

By double-clicking on a node n in the result list, the result list is replaced by the list of nodes adjacent to n. This show-neighbors feature is described in detail in section 8.4.

If you have invoked the `Find node` function several times, all successful searches are saved in a history; to browse through them, use the and buttons.

To close the search window press **Esc** or use the standard close button of the window.

By default, the search string is matched not only against the node labels, but also against the node titles and infofields.

Since the result list only displays the label, it may happen that the list contains nodes that seemingly do not match the search string (since it is not contained in their label, but in one of their infofields).

Figure 8.6: Configuration window for finding nodes

This default behavior can be changed by clicking on the button. A new subwindow will appear in which you can configure the search criterion (see Figure 8.6). Furthermore, you can set up whether the search string should be interpreted as regular expression, whether the search should be case-sensitive, whether the search should be restricted to nodes which have been selected in the graph before, and whether only nodes that are currently visible in the graph should be considered for search results. The regular expressions for finding nodes have the same structure as those described in section 3.5.5.

8.4 Show Neighbors

In large control-flow graphs, visually following long bent edges may prove difficult. Furthermore, call edges are not connected to call instructions in the boxed representation so that finding the routine called by looking at the call edge is impossible without changing the representation. For these reasons, **aiSee** provides a method for finding the neighbors of a node. Here, *neighbor* means any node connected by an edge, regardless of the direction of the edge. This includes *invisible edges*, which are used to attach stack analysis results to routines, basic blocks, and instructions.

Note that the notion of neighbor relies on the specification of the edges in the GDL file that is being visualized. The nodes declared as source and target of an edge in the GDL file are not necessarily the nodes connected by the edge in the visualization. For instance, the target of a call edge to an analyzed routine is *not* the corresponding routine node, but the first instruction of the routine called. In the visualization however, the call edge leads to the routine node if the control-flow graph of the routine called is folded or boxed. In fact, analyzed routine nodes do not have any neighbors except for the stack analysis result boxes attached to them by invisible edges if stack

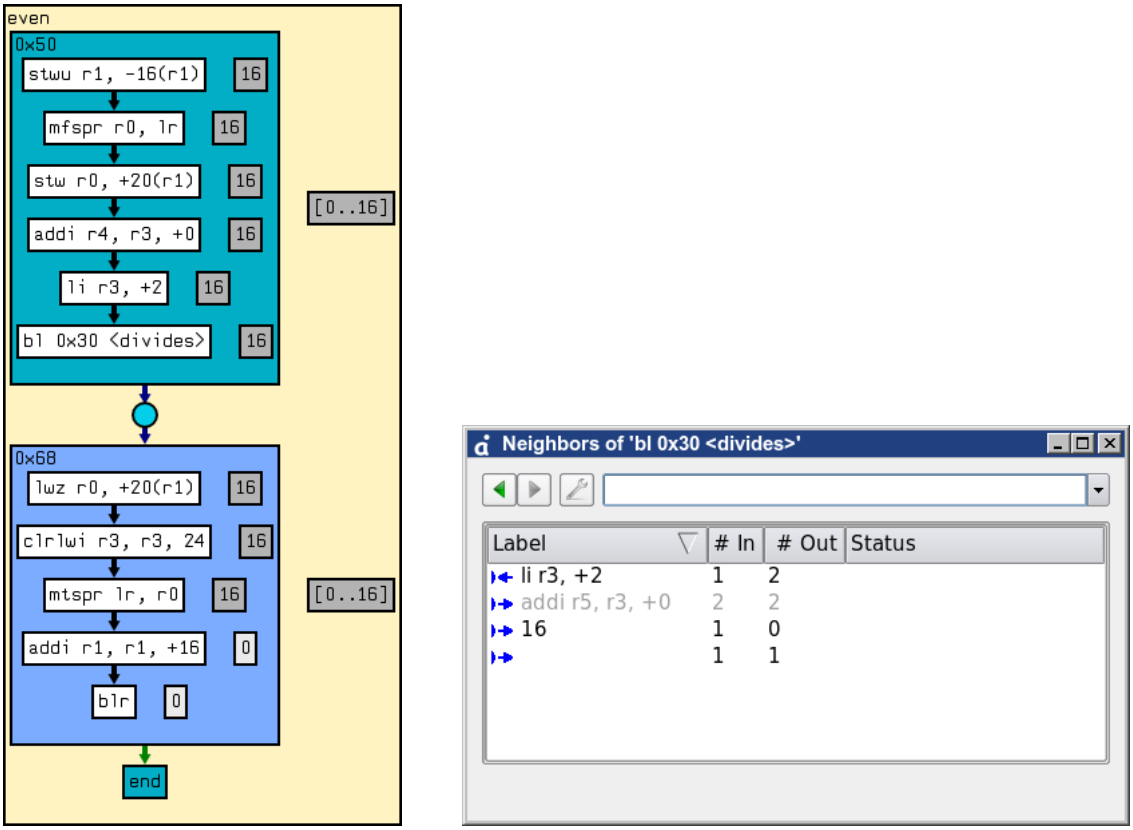


Figure 8.7: Window with list of neighbors

analysis is enabled. The same is true for blocks containing instructions.

When you select a node *n* and hit **Ctrl+n** or select **Show Neighbors** from the context menu, an auxiliary window listing the neighbors of *n* appears. Alternatively, you can also access this function via the button in the tool bar or via **Window→Show neighbors** in the menu. When you double-click on a node in the list of neighbors, the list is replaced by the list of neighbors of the double-clicked node. By repeating this procedure, you can walk through the entire graph (if it is connected).

Figure 8.7 shows the instructions of routine `even` with stack analysis results on the left and the neighbors list of the call instruction `bl 0x30 <divides>` on the right. Each entry in the list of neighbors starts with the symbol ↦ or ↤. Here, ↦ means that the edge connecting the selected node to the neighbor points from the selected node to the neighbor, while ↤ indicates an edge in the opposite direction. After this symbol, the label of the neighbor, its number of ingoing and outgoing edges, and its status follow.

The dialog box is actually identical to the `Find Node` dialog box (see Figure 8.5 and section 8.3). Thus it also has the functionality that nodes selected in the list are highlighted and centered in the graph picture. Note that if you (accidentally) supply a search string in the text input field, the search will refer to all nodes of the graph (that is, not only to the nodes available in the list of neighbors); in this case you can press to move back to the list of neighbors.

In the example of Figure 8.7, one of the neighbors (with label `16`) is the stack result node attached to the node originally selected, and `addi r5, r3, +0` is the first instruction of routine `divides`,

whose node is not visible in the graph on the left, and thus colored in gray in the node list. The other two neighbors are the predecessor (marked by ⇤) and the successor (marked by ⇥) of the call instruction in the control-flow graph of `even`. The last node is the light green circle indicating the call point. The neighbors of such a call point are the nodes belonging to the call instruction and the instruction immediately following this instruction.

Chapter 9

Typical Working Patterns

This section lists some working patterns for obtaining useful information or reacting to reported problems. All mentioned **aiSee** operations refer to the new **aiSee 3** as described in chapter 8.

9.1 Finding a Routine in an aiSee Picture

This section describes a method to find a routine in an **aiSee** picture by its name.

1. It is recommended, but not necessary, to first bring the picture in a state where only routine nodes are visible. This can be done using the `Reload` function by clicking on in the toolbar, which reads the graph again from the file where it has been stored and shows its initial layout (see section 8.1.6). Usually this means that only routines and their calling relationships are displayed.

   Remarks:

   - If the `Boxed Routines` flag in the `Visualization View` is activated (see section 3.4.6), the initial layout also shows basic block graphs.
   - If the file containing the displayed graph has meanwhile been deleted, the `Reload` operation cannot be executed.

   In both cases, you should not worry and continue with step 2.

2. Zoom using the `+` key until individual routines can be clearly distinguished (it does not matter if most of them are situated off screen).

3. Use `Ctrl+f` to start a node search (see section 8.3).

4. Write the routine name *N* in the input field and select the entry *N* in the node list.

5. The previous step causes the **aiSee** picture to be scrolled so that the desired routine is situated close to the center of the window and selected.

6. When you press `b`, its basic block graph becomes visible (but not the individual instructions). When you press `Shift+B` instead, its basic block graph with all instructions becomes visible. Details about exposing and hiding subgraphs by these and other operations can be found in section 8.2.

9.2 Finding an Instruction in an aiSee Picture

This section describes a method to find an instruction in an **aiSee** picture by its address.

1. Press `Ctrl+f` to open the `Find Node` subwindow (see section 8.3).
2. Press the button to change the search preferences.
3. Within the `Search Preferences` subwindow (see Figure 8.6) solely select `Info 2` in the `Search for` section and press `OK`.
4. Back in the `Find Node` subwindow provide the instruction address in the input field.
5. Ideally, there is now a single node in the node list. Select it, and in case the corresponding node is hidden by folding, that is, its entry is depicted in gray, click with the right mouse key on it and select `Expose and recursive box node` from the context menu. This causes the routine and basic block containing the instruction to be put into the boxed state so that the desired instruction becomes visible in the **aiSee** picture. Clicking on it again in the find-node dialog causes the picture to scroll so that the desired instruction is situated near the center of the window.

9.3 Finding the Place of Maximum Stack Usage

If you want to find the place of maximum stack usage, you should perform a stack analysis with a report file. Reporting is activated by specifying a file name in the `Report file` field in the `Files` view in the `Configuration` section (see section 3.4.1). After a successful stack analysis with reporting enabled, the report file will contain analysis results for all routines. Routines where the global maximum of the system or user stack usage is attained are marked by the phrase `** hits the global maximum **`. Thus they can be found by searching for this phrase in the report file. Note that there may be several different such routines, some for the system and some for the user stack.

The report file does not only contain the global and local stack analysis result for each routine, but also the call chain leading to the global result, e.g.,

```
routine 'divides.L1'
  => maximum usage is 8u/128s
  => ** hits the global maximum **
  => maximum local usage is 0u/0s
  => call-stack:
```

```
User/System Stack History:
   ->      0u/    0s <    8u/   64s> 'main'
   ->      8u/   64s <    0u/   64s> 'prime'
   ->      8u/  128s <    0u/    0s> 'even'
   ->      8u/  128s <    0u/    0s> 'divides'
   ->      8u/  128s <    0u/    0s> 'divides.L1'
   => 8u/128s bytes
```

Section 7.3.1 provides details about the meaning of global and local stack results at routines and the interpretation of the call chain.

If you would like to see a visualization of the place of maximum stack usage, search for the routine with maximum stack usage in the **aiSee** picture of the application with stack analysis results as described in section 9.1. Once you have found the routine, the stack analysis result box attached to it provides the same information as the report file: global and local stack analysis results form the box label, and the call chain is displayed when you select the first information field (see section 8.1.7).

9.4 Avoiding State Explosion

If the number of combined cache and pipeline states excessively increases in a basic block, a^3 warns about this "state explosion":

```
tricorepipe: Info: In "dry2_1.c", line 15:
   At address 0x80000494 in routine 'strcpy_x.L1':
     Possible state explosion from 266 entry states
       to 30736 final states.
     Please use further annotations
```

The given address is the address of an instruction, say I, in a basic block, say B. In this example, the analysis of block B starts with 266 states, which evolve into 30736 states after instruction I.

Often, a state explosion results from a sequence of memory accesses to unknown addresses. If there are n memory areas with different properties, every such memory access splits each state into n successor states, one for every area that may be accessed. Thus you should watch out for unknown accesses near the address specified. This can be done in the report file, maybe together with the graph picture. (See section 9.2 for how to find an instruction by its address in the picture).

The state explosion can often be avoided if you add some information about the addresses of memory accesses near the point where the explosion occurs (of course, this is only possible if you possess such information). Two kinds of annotations are useful for this purpose: First, you may provide direct address information by means of an access annotation

INSTRUCTION *ProgramPoint* ACCESSES $Range_1$, ..., $Range_n$;

as described in section 5.9. Such address information is valid for a single access only. Second, if addresses are formed by adding a known offset to an unknown register value, the address is known once you specify that register value (provided you know it). This can be done in the form

```
INSTRUCTION ProgramPoint IS ENTERED WITH
   Register1 = Expr1, ..., Registern = Exprn;
```

as described in section 5.10. Such information may be more economic than an ACCESSES specification since it may affect several accesses via the same base register.

Appendix: Document History

This appendix describes the revision history of this manual. Dates are written in the format YYYY-MM-DD.

A *manual release* is a release version of the manual. The manual release number occurs on the title page after the heading "Manual Release". Note that manual releases are not necessarily correlated with product releases. The installation package of a product release contains the most recent manual release available when the package was created.

The manual release number is incremented when there is a new version of some manual. In contrast, the build number and the date, also appearing on the title page, are incremented on a daily basis.

The document history given in the sequel documents all changes in this manual. Therefore, it includes the dates when certain product features were first described in the manual. Yet note that these dates are usually different from the dates when the product features were added to the product. Usually, it takes some time to describe a new product feature. Sometimes, it even happened that a product feature was not yet documented in this manual when the product with the new feature was released.

2008

2008-11-27 Initial version of manual describing a^3 for TriCore.
Manual Release 00037

2008-12-05 Some details added to description of graph browser **aiSee** (chapter 8).
Manual Release 00039

2008-12-11 Clarification: Timing analysis does not support accesses spanning multiple memory areas (6.3).
Manual Release 00040

2008-12-12 New check box `Show MD5 sums` in the `Files` view (section 3.4.1).
Description of `TriCore` view (section 3.4.2).
New target TriCore 1797 supported.
Manual Release 00041

2008-12-16 Update of description of `Analyses` view (3.4.3) and `Registers` view (3.4.4).
Computed calls are indicated by a red border (no longer dotted; 7.1.1.1).
Manual Release 00042

2009

2009-01-16 Clarification: Instruction fetches from Data Flash, Boot Rom, and LMI are not modeled in case of TC1766 and TC1796 (6.3).
Manual Release 00045

2009-01-22 Sizes of caches and local memory can be specified in the TriCore view (3.4.2.4).
AIS annotations for assumed alignment (5.9.5).
More exact description of assumed alignment (3.4.3.4, 5.9.5).
Manual Release 00046

2009-02-02 Update of `Visualization` view (3.4.6):
– Preview and `Color scheme` added;
– `Fast animation` removed;
– `Trace-like pipeline state visualization` added.
Renaming of chapter 7 and its sections.
Manual Release 00047

2009-02-19 New: `Show per context info in XML report` in `Analyses` View (3.4.3.2).
The font for labels in graphs can be selected in the `Visualization` view (3.4.6).
Register values may refer to other registers (section 5.10).
Manual Release 00048

2009-02-26 Shortcut for `Analysis` → `Start` is now `F7` (formerly `F6`).
Shortcut for `Project` → `Start all analyses` is now `F6` (formerly `F5`).
Addition of value analysis (chapter 5, in particular 5.12, 5.17.3, 5.23)
and viewing of value analysis results (7.2).
Manual Release 00050

2009-02-27 Legend for color scheme replaced by hint to legend built in a^3 (section 7.1).
Manual Release 00051

2009-03-19 Description of configuration files added (new section 2.2).
Manual Release 00052

2009-03-31 Infeasible routines are shown in gray
(they previously have been omitted in graphs with analysis results).
Paragraphs with stack-analysis examples promoted to subsections (7.3.1 – 7.3.3).
Manual Release 00053

2009-06-04 New: `Output worst-case execution path in XML report`
in `Analyses` View (3.4.3.2).
Description of `Results` view (now called `Overview` view) added (3.5.4).
Description of disassembly views added (3.5.6).
Description of source code and debug views added (3.6).
Manual Release 00055

2009-07-30 Additional execution time can be product or selection (5.13).
Description of user registers added (section 5.10).
Register specifications may involve general AIS expressions (5.10).
Infeasibility can be conditional (5.17, in particular 5.17.2).
Condition values can be conditional (5.18, in particular 5.18.2).

Description of parametric loop bounds added (5.20, in particular 5.20.1 and 5.20.2).
Description of AIS expressions added (5.28).
Mirrored memory areas are handled like different non-mirrored areas (6.1).
Manual Release 00056

2009-08-21 Navigation through view list and view history (beginning of chapter 3).
New field `XML result file` in the `Files` view (section 3.4.1).
New `Source files` view uniting the former `Includes` and `Replacements` views (3.4.5).
Clarification: Areas containing data values are constant, not read-only (5.12.5).
Manual Release 00058

2009-09-02 The initial stack pointer value may be important for stack analysis (section 3.4.2).
Manual Release 00059

2009-09-15 New section on avoiding state explosions (section 9.4).
Manual Release 00060

2009-09-30 New field `XML stylesheet` in the `Files` view (section 3.4.1).
Update of analysis management: `Overview` and `Results` view transformed into `Create` and `Overview` view (section 3.5).
WCET path and infeasible code marked in `Disassembly` view (3.5.6).
New view showing the section table of the executable (3.6.5).
Manual Release 00061

2009-11-18 Machine settings can be specified via hardware control registers (3.4.2.3).
Manual Release 00062

2009-11-23 Remark on unresolved source positions added (section 5.31.1).
New expressions `value` (5.28.14), `restrict`, `defined`, `bounded`, `try` (5.28.6).
Examples for the new expressions (5.20.2).
Memory cells can be referenced in expressions (5.28.5).
Manual Release 00063

2009-12-01 More command line options (section 2.1).
Difference analysis is mentioned (section 3.5.1).
Path analysis with the ILP method based on prediction files does no longer require full unrolling of loops and recursive procedures (3.4.3.7, 5.4).
New annotation `IS A TAIL CALL` (section 5.8.1).
Information on the WCET contribution of non-routine snippets can be requested (5.14).
Manual Release 00065

2010

2010-01-13 New command line option `-B` for “interactive” batch mode (section 2.1).
New command line option `-i` to start specific analyses in batch mode (2.1).
Manual Release 00066

2010-01-15 Updates in `Help` menu (3.2) and `Project` menu (3.3).
Unified `Preferences` dialog box (3.3 and 3.5.6).
No more global XML report file (3.4.1).

New textual and XML report files local to analyses (3.5.2).
Messages local to analyses in `Overview` view (3.5.4).
Reorganization of global `Information` views (3.6).
Manual Release 00067

2010-01-18 Labels defined in the AIS file (5.25) can be used as analysis start (3.5.2).
Such AIS labels can refer to areas (previously only single program points) (5.25).
AIS labels can be forced to replace routine names in graphs and report files (5.25).
Manual Release 00068

2010-01-23 The keyword `EXACTLY` is optional in loop bounds (section 5.20.1).
`Additionally takes` now admits general expressions (5.13).
Therefore, certain old annotations have to be modified (5.13.1).
New AIS expression `undef` giving an undefined value (5.28.3).
New `switch` expressions in AIS (5.28.13).
Space and time units such as `bytes` and `cycles` in expressions (5.28.16).
New logical operator `always` in AIS expressions (5.28.12).
AIS assertions, possibly about values of registers or memory cells (5.11).
Manual Release 00069

2010-01-27 Additional start points for timing analysis (section 5.5).
For these, instructions accessing some memory area can be counted (5.24.1).
Manual Release 00070

2010-02-03 Not accessing instructions, but accesses to some memory area are counted (5.24.1).
Manual Release 00071

2010-02-05 Directives to disable cache and pipeline flushes at excluded snippets (5.15.6).
Manual Release 00072

2010-02-19 New option `Reduce number of identical messages` (3.4.3.4).
Manual Release 00073

2010-02-26 Frequency of external clock at pin BFCLKO can be specified (3.4.2.5).
The sections in the AIS overview are marked with the analyses that they affect (5.1).
Manual Release 00074

2010-03-09 The `max-unroll` parameter can be set for individual loops (5.4).
Manual Release 00075

2010-03-23 Description of target check and how to switch it off (section 5.8.4).
Access counting now works for the main start point, too (relocated section 5.24.1).
Manual Release 00076

2010-04-01 Export feature in `Project` menu for copying input files (section 3.3).
GUI support for XTC interface to T1 tool (3.3).
Maximum memory usage in percent (section 3.4.3.8).
Flag `Enable widening` instead of widening threshold (3.4.3.8).
New AIS expression `inf` making `undef` obsolete (5.28.3).
Manual Release 00078

2010-04-09 Descriptions of calls or branches via tables may involve indirections (5.8.3).
Arrays may be null-terminated, and the access may be restricted to special indices.

New syntax for requests for the WCET contribution of non-routine snippets (5.14).
Therefore, existing requests have to be modified (5.14.1).
Operands of unary operators should be parenthesized in AIS expressions (5.28).
The results of value analysis can be converted to AIS annotations (7.2.3).
Manual Release 00079

2010-06-21 Special value *nan* is replaced by $(-inf..inf)$ (5.28, in particular 5.28.2, 5.28.8).
The predicate `defined` that tested for *nan* has been removed
(use `bounded` instead, see section 5.28.6).
The former `round` operation is now called `int` (5.28.9).
Manual Release 00081

2010-07-13 New `WCET computation mode Local best-case` (3.4.3.6).
Manual Release 00082

2010-07-27 New operator `exact` in AIS expressions to test if a value is exact (5.28.6).
Document structure: Introduction of proper chapters and four hierarchy levels.
Reorganization of chapter 3 to match the structure in the GUI.
Preferences dialog box: `Code view` part renamed into `Disassembly` (3.5.6).
There can be more than one `interproc` specification (5.4).
Manual Release 00085

2010-07-30 Description of timing specifications for loops (5.21).
Clarification about positions of the form `"`*A*`" + 0 bytes` (5.9.1).
Description of the *`Position`* `.. +` *n* `bytes` abbreviation (5.9.1 and 5.12.1).
Description of `align` operation in program-point specifications (5.29).
Manual Release 00086

2010-08-03 New annotation `might contain data` for memory areas (5.12.4).
AIS expressions can produce and process modulo information
(5.28, in particular 5.28.2 and 5.28.10).
New operators `uint` and `sint` for casting arbitrary values
to unsigned and signed integers of given bit width (5.28.10).
New operator `exactly ==` for comparing two sets of possible values (5.28.11).
Manual Release 00087

2010-08-04 Execution times of code snippets excluded from analysis may be expressions (5.15.3).
The option `Disable visualization` in the `Visualization` view
now only takes effect in batch-mode runs of a^3 (3.4.6).
Manual Release 00088

2010-08-13 Simplified syntax for referencing memory contents in AIS expressions (5.28.5)
(not fully backward compatible if units or `lbf`/`hbf` have been used).
Manual Release 00089

Index

Symbols

⋆ in title bar, 17
+ (**aiSee**), 222, 223
− (**aiSee**), 222, 223
⊥ (stack analysis result), 180

Numbers

0 [zero] (**aiSee**), 222, 223

A

About... (a^3 Help menu), 19
AIS file (Files view), 22
AIS files, 22
AIS format, 66
AIS Quick Reference (a^3 Help menu), 19
aiSee, 221
aiSee Manual (a^3 Help menu), 19
aligned data accesses, 30
alloc, 178
alloca, 178
analysis phases, 60
:Anon_*A*, 40, 55, 183
anonymous routines, 55, 183
any-history (context), 57
arrays of function pointers, 78
arrow keys (**aiSee**), 222
assembly instruction, 187, 224
assembly name of routine, 54
assertion (AIS language), 13, 66
Assume aligned data accesses, 30
Auto (TriCore view), 24
availability of a^3, 12

B

Shift+B (**aiSee**), 226
b (**aiSee**), 226
basic block, 185, 224
basic block graph, 184, 224
batch mode, 15
black arrow, 186, 188
blue arrow, 186, 225
Boxed basic blocks (Visualization option), 38
Boxed routines (Visualization option), 37
boxed subgraph, 225

C

c (**aiSee**), 226
C source code, 38
cache analysis, 62
Cache size (TriCore view), 26
call edge, 183, 186, 225
call graph, 182
calling conventions, 178
calloc, 178
CLOCK (AIS file), 68
clustered subgraph, 225
Color scheme (Visualization option), 38
Comma as decimal sign (Visualization option), 37
COMPILER (AIS file declaration), 69
computed call, 76, 183
  via array of function pointers, 78
computed jump, 76, 186
conditional jump, 186
configuration file, 15
context specification, 69
contexts, 55
Control-Flow Graph (a^3 Analysis menu), 42
Create AIS annotations, 193
create view, 39
Ctrl+f (**aiSee**), 227
Ctrl+i (**aiSee**), 224

Ctrl+n (**aiSee**), 230
Cumulative WCET contribution, 207

D

dark gray stack analysis result boxes, 199
dark red arrow, 186
Data cache mode, 30
declaration (AIS language), 13
declarations (AIS language), 66
default-unroll, 70
defining analyses, 39
details (WCET info field), 213
Disable data accesses to PMI (TriCore view), 25
Disable visualization for batch mode (Visualization option), 39
Disable writes via LMI (TriCore view), 24
Disassembly (a^3 Analysis menu), 42
disassembly view, 45
Display analysis messages (a^3 Analysis menu), 42
Display analysis results (a^3 Analysis menu), 42
Display source code (Visualization option), 38, 184, 188
Display WCET profiles (Visualization option), 38
DMA accesses, 104
Do not analyze values of memory cells, 35
dummy call, 55, 185
Dump AIS files to report, 29
dynamic data structures, 178

E

Enable widening for cache states, 34
END (AIS file), 73
end (in displayed graphs), 185
ENTRY (AIS file), 73
entry block, 185
environment variables, 13
Esc (**aiSee**), 223, 228
Exceptions never occur, 30
Executable text field (Files view), 22
exit block, 185, 186
Export (a^3 Project menu), 20
Export of machine settings, 25
external routines, 180, 200
extract annotations from source files, 29, 64, 174

F

Ctrl+f (**aiSee**), 227
f (**aiSee**), 226
"false" edge, 186
Files view, 22, 191, 192
find node (**aiSee**), 227
Find node... (**aiSee** Edit menu), 227
finding nodes (**aiSee**), 227, 229
First free CSA (TriCore view), 24
folded subgraph, 224
Font (Visualization option), 38
function pointers
  arrays of, 78

G

Generate pipeline basic block statistics, 33
global results (for routines), 196
graph selection (**aiSee**), 223
gray routine, 183
gray stack analysis result boxes
  dark gray, 199
  light gray, 199
green arrow, 186

H

h (**aiSee**), 222
Help menu (a^3), 19
hierarchy levels in graphs, 224

I

Ctrl+i (**aiSee**), 224
i (**aiSee**), 223
ILP solver, 33
Import of machine settings, 25
INCLUDE (AIS file), 144
include directories, 65
inf, 147

infeasible problem, 64, 75, 120, 135
Info fields visible (**aiSee** View menu), 224
info fields
  access, 223
  at basic blocks, 185
  at instructions, 187
  at routines, 183, 206
  first info field, 197, 215
  second info field, 186, 187, 206, 211, 216
  third info field, 183, 186
Inline loop routines (Visualization option), 38
Instruction cache mode, 30
instruction sequence, 187, 224
interactive batch mode, 15
interproc (context specification in AIS), 69
invocation, 14
irreducible loops, 54, 128, 129, 136

J

jump
  computed, 186
  conditional, 186
  unconditional, 186

K

keep temporary files, 23

L

LABEL (AIS file), 142
left mouse button (**aiSee**), 223, 227
License... (a^3 Help menu), 19
license key, 12
light gray stack analysis result boxes, 199
light red arrow, 216
line information, 174
Loading machine settings, 25
local results (for routines), 196
longjmp, 178
loop bound analysis, 61
loop bound missing, 129
loop bounds, 128
loop transformation, 50
loop without exit, 75

M

m (**aiSee**), 222
malloc, 178
Manhattan edges (Visualization option), 37
Manual (a^3 Help menu), 19
max t, 212
max-length, 57, 60, 69
max-unroll, 59, 70
Maximum memory usage, 34
MD5 sums, 23
Memory size (TriCore view), 26
Mode menu (**aiSee**), 222
mouse dragging (**aiSee**), 222, 223
multi-entry loops, 54, 128, 136

N

Ctrl+n (**aiSee**), 230
names of routines, 54, 183
NEVER EXECUTED, 120
New (a^3 Project menu), 39
New project (a^3 Project menu), 19
no-history (context), 57
node selection (**aiSee**), 223, 227, 229
nonterminating loop, 64, 75
normal edge, 186, 188
Normal layout (Visualization option), 37, 188
not-analyzed routines, 109, 180, 183
Number of CSAs (TriCore view), 24

O

o (**aiSee**), 222
Open file (a^3 Project menu), 20
Open project... (a^3 Project menu), 19
operation modes (**aiSee**), 222
orange routine, 183
Output worst-case execution path in XML report, 29
overview view, 42

P

Pan (**aiSee** Mode menu), 222

parametric loop bounds, 128
path analysis, 63
path analysis options, 32
Path analysis variant, 32
path replacement, 65
pathan2, 64
PgDn (**aiSee** command), 222
PgUp (**aiSee** command), 222
phases of the analysis, 60
pink stack analysis result boxes, 198
pipeline analysis, 62
Predicted WCET Contribution, 207
Preferences... (a^3 Project menu), 21
Project menu (a^3), 19
Project messages verbosity level, 29
project tool bar, 18, 19

Q

Quit (a^3 Project menu), 21

R

Recent Projects (a^3 Project menu), 20
recursion, 183, 186, 204
red arrow
    dark red, 186
    light red, 216
red border, 75, 183, 185
red stack analysis result boxes, 198
Reduce number of identical messages, 30
Registers view, 35
Reload (**aiSee** Edit menu), 223
Report file (Files view), 22, 61, 128, 198
report file, 198
report file verbosity level, 29
resetting (**aiSee**), 223
return instruction, 76
routine names, 54, 183
routines, 54, 183

S

s (**aiSee**), 223
safe patterns, 30, 61
Save project (a^3 Project menu), 20
Save project as... (a^3 Project menu), 20
scaling (**aiSee**), 222
scrolling (**aiSee**), 222
SDA base (TriCore view), 24
SDA2 base (TriCore view), 24
Search Preferences window (**aiSee**), 229
Select (**aiSee** Mode menu), 223
selection
    of graphs, 223
    of nodes, 223, 227, 229
    of project files, 19
    of subgraphs, 227
self-loop, 64
Set as default button (Visualization option), 39
setjmp, 178
View → Show info (**aiSee** View menu), 224
show MD5 sums, 23
Show neighbors (**aiSee** Window menu), 230
show neighbors (**aiSee**), 229
Show panner (**aiSee** View menu), 223
show panner (**aiSee**), 223
Show per context info in XML report, 29
skip target check, 82
snippets, 108
source code, 38, 188
source code annotations, 174
Source files view, 35
Stack address (TriCore view), 23
stack analysis, 195
Interactive (a^3 Analysis menu), 42
Start (a^3 Analysis menu), 41
Start all analyses (a^3 Project menu), 21, 42
state explosion, 234
statistics (WCET info field), 214
Stop analysis (a^3 Project menu), 21, 42
strip compilation path, 30, 65
stripped executables, 55
subgraph, 224
    boxed, 225
    clustered, 225

folded, 224
selection, 227
unfolded, 225
subgraph selection (**aiSee**), 227
`sum #`, 212
summary node, 224
switch statement, 186

T

T1 (by Gliwa), 20
`Target` (`TriCore` view), 26
target check, 82
temporary files, 12, 23
termination (a^3), 21
termination (analysis), 21, 42
textual successor, 186
timing specifications for loops, 135
title bar (of a^3), 17
`Trace-like pipeline state visualization` (Visualization option), 38
`TriCore` view, 23
"true" edge, 186
`TRY` (AIS files), 143

U

`u` (**aiSee**), 226
Unable to resolve ..., 143
unbounded problem, 64, 129
unconditional jump, 186
unfolded subgraph, 225
unreachable code, 180
unresolved computed call, 75
unrestricted recursion, 180
Use only safe patterns, 30, 61

V

`v` (**aiSee** command; value analysis), 190
value analysis, 62
verbosity level
project messages, 29
report file, 29
`Version...` (a^3 `Help` menu), 19
`View` menu (**aiSee**), 223
Visualization view, 37, 182
volatile memory area, 98

W

`w` (**aiSee**), 222
WCET computation mode, 31
`Welcome` (a^3 `Help` menu), 19
`Welcome` view (a^3), 18
widening, 34

X

`XML result file` (`Files` view), 23
`XML stylesheet` (`Files` view), 22
`XTC` (a^3 `Project` menu), 20

Y

yellow routine, 183

Z

`Zoom` (**aiSee** `Mode` menu), 223