Bound-T time and stack analyzer

Bound-T Assertion Language

TR-UM-003

Issue 6.5
2013-11-30

Tidorum Ltd.

Tidorum Ltd
www.tidorum.fi
Tiirasaarentie 32
FI-00200 Helsinki
Finland

This document, then a part of the Bound-T User Manual, was written at Space Systems Finland Ltd by Niklas Holsti, Thomas Långbacka and Sami Saarinen.
The document is currently maintained at Tidorum Ltd by Niklas Holsti.

Copyright 2005 – 2013 Tidorum Ltd.
This document can be copied and distributed freely, in any format or medium, provided that it is kept entire, with no deletions, insertions or changes, and that this copyright notice is included, prominently displayed, and made applicable to all copies.

Document reference:	TR-UM-003
Document issue:	6.5
Document issue date:	2013-11-30
Bound-T version:	4
Last change included:	BT-CH-0258
Web location:	*http://www.bound-t.com/manuals/assertion-lang.pdf*

Trademarks:
Bound-T is a trademark of Tidorum Ltd.

Credits:
This document was created with the free OpenOffice.org software, *http://www.openoffice.org*.

The information in this document is believed to be complete and accurate when the document is issued. However, Tidorum Ltd. reserves the right to make future changes in the technical specifications of the product Bound-T described here. For the most recent version of this document, please refer to the web-site *http://www.bound-t.com/*.

If you have comments or questions on this document or the product, they are welcome via electronic mail to the address *info@tidorum.fi* or via telephone or ordinary mail to the address given below.

Please note that our office is located in the time-zone GMT + 2 hours, and office hours are 9:00 - 16:00 local time. In summer daylight savings time makes the local time equal GMT + 3 hours.

Cordially,

Tidorum Ltd.

Telephone: +358 (0) 40 563 9186
Fax: +358 (0) 42 563 9186
Web: *http://www.tidorum.fi/*
E-mail: *info@tidorum.fi*

Mail: Tiirasaarentie 32
FI-00200 Helsinki
Finland

Credits

The Bound-T tool was first developed by Space Systems Finland Ltd. (*http://www.ssf.fi/*) with support from the European Space Agency (ESA/ESTEC). Free software has played an important role; we are grateful to Ada Core Technology for the Gnat compiler, to William Pugh and his group at the University of Maryland for the *Omega* system, to Michel Berkelaar for the *lp-solve* program, to Mats Weber and EPFL-DI-LGL for Ada component libraries, and to Ted Dennison for the *OpenToken* package. Call-graphs and flow-graphs from Bound-T are displayed with the *dot* tool from AT&T Bell Laboratories. Some versions of Bound-T emit XML data with the *XML_EZ_Out* package written by Marc Criley at McKae Technologies.

Contents

1 INTRODUCTION 9

1.1 What Bound-T is 9
1.2 Overview of this document 11
1.3 Other Bound-T documentation 12

2 WRITING ASSERTIONS 14

2.1 What assertions are 14
2.2 Assertion = context + fact 15
2.3 Assertions on the repetition of loops 18
2.4 Assertions on the number of loop starts 20
2.5 Assertions on the execution count of calls 23
2.6 Assertions on the execution count of an instruction 26
2.7 Assertions on the execution time of a subprogram 27
2.8 Assertions on the execution time of a call 28
2.9 Assertions on the stack usage of a subprogram 29
2.10 Assertions on the final stack height of a subprogram 31
2.11 Assertions on the callees of a dynamic call 32
2.12 Assertions on variable values 33
2.13 Assertions on variable invariance 37
2.14 Assertions on volatility 40
2.15 Assertions on target-specific properties 42
2.16 Assertions on instruction roles 42
2.17 Special assertions on subprograms 44

3 IDENTIFYING PROGRAM PARTS 49

3.1 Why and how: the different ways 49
3.2 Names, scopes, and qualified names 50
3.3 Source-code positions 51
3.4 Identifying subprograms 55
3.5 Identifying variables 57
3.6 Identifying loops 58
3.7 Identifying calls 64
3.8 Identifying instructions 68

4 ETERNAL LOOPS AND RECURSION 70

4.1 Handling eternal loops 70
4.2 Handling recursion 72

5 ASSERTION LANGUAGE SYNTAX AND MEANING 76

5.1 Introduction 76
5.2 Assertion syntax basics 76
5.3 Overall assertion structure 80
5.4 Scopes 81
5.5 Global bounds 81

5.6 Subprograms....82
5.7 Loops....84
5.8 Calls....87
5.9 Instructions....89
5.10 Clauses and facts....90
5.11 Execution-time bounds....92
5.12 Stack bounds....92
5.13 Repetition bounds....93
5.14 Start bounds....99
5.15 Variable bounds....99
5.16 Variable invariance....101
5.17 Volatility Marks....102
5.18 Property bounds....103
5.19 Callee bounds....104
5.20 Role bounds....104
5.21 Combining assertions....105

6 TROUBLESHOOTING 108

6.1 Warning messages....108
6.2 Error messages....109

Tables

Table 1: Allowed means of identification of program parts 49
Table 2: Line-number comparison fuzz 80
Table 3: Meaning of loop properties 86
Table 4: Meaning of call properties 89
Table 5: Fact and context combinations 91
Table 6: Meaning of execution time assertion 92
Table 7: Meaning of repetition count assertion 93
Table 8: Meaning of variable value assertion 99
Table 9: Meaning of variable invariance assertion 102
Table 10: Meaning of property value assertion 103
Table 11: Effect of multiple assertions on the same item 105
Table 12: Warning messages 108
Table 13: Assertion error messages 110

Figures

Figure 1: Inputs and outputs 11
Figure 2: An assertion file 17
Figure 3: Starting a loop, no unpeeling, no null check 22
Figure 4: Starting a loop with null check 22
Figure 5: Starting an unpeeled loop with null check 23
Figure 6: An irreducible flow-graph 47
Figure 7: Longest call path in recursion example 74
Figure 8: A loop in a flow-graph 95
Figure 9: A general kind of loop asserted to repeat 6 times 96
Figure 10: A middle-exit loop asserted to repeat 6 times 97
Figure 11: An exit-at-end loop asserted to repeat 6 times 97

Document change log

Issue	Section	Changes
6.4	-	Change log started.
	Various	Inserted "manuals/" in the URLs for Bound-T manuals.
	Section 1.1	Added note on stack-usage analysis.
	Section 2.9	Added consideration of assertions on and analysis of final stack height.
	Section 2.10	Added explanation of stable and unstable stacks.
	Section 2.15	Added section on instruction role assertions.
	Section 2.16	Added consideration of final stack height. Added subsection on **return to offset** assertions for subprograms.
	Section 3.4	Added subsection on identifying subprograms by offsets.
	Section 5.2	Added subsection on instruction-role syntax. Added keyword **returns** as a synonym for **return**.
	Section 5.6	Added syntax for **return to offset**. Added syntax for optional offset in *Sub_Name*.
	Section 5.9, 5.10	Added *Role_Bound* clauses for instruction blocks.
	Section 5.19	Added section on *Role_Bound* clauses.
	Section 5.20	Added row on instruction roles to table 11.
	Section 6.2	Updated error-message table.
6.5	All	Page numbering starts from 1 for the title page and continues sequentially from the front matter to the text, for easier PDF handling.
	Section 2.14	Added section on asserting volatility of variables.
	Section 3.6	Updated to explain that the **executes** property is not inherited from a nested, inner loop to the containing outer loops.
	Section 5.17	Added section on the syntax of **volatile** assertions.
	Chapter 6	Updated warning and error messages.

This page is almost blank on purpose.

1 INTRODUCTION

1.1 What Bound-T is

Bound-T is a tool for developing real-time software - computer programs that must run fast enough, without fail.

The main function of Bound-T is to compute an *upper bound* on the *worst-case execution time* of a program or subprogram.

The function, “bound time”, inspired the name “Bound-T” pronounced as “bounty” or “bound-tee”.

Bound-T can also compute an *upper bound* on the *stack usage*, thus making sure that the program cannot fail due to stack overflow.

Real-time deadlines

A major difficulty in real-time programming is to verify that the program meets its run-time timing constraints, for example the maximum time allowed for reacting to interrupts, or to finish some computation.

Bound-T helps to answer questions such as

- What is the maximum possible execution time of this interrupt handler? Is it less than the required response time?
- How long does it take to filter a block of input data? Will it be ready before the output buffer is drained?

To answer such questions, you can use Bound-T to compute an upper bound on the execution time of the subprogram concerned. If the subprogram cannot be interrupted by other computations, and this upper bound is less or equal to the time allowed for the subprogram, we know for sure that the subprogram will always finish in time.

When the program is concurrent (multi-threaded), with several threads or tasks interrupting one another, the execution time bounds for each thread can be combined to verify the timing (schedulability) of the program as a whole.

Static analysis – all cases covered

Timing constraints are traditionally addressed by measuring the execution time of a set of test cases. However, it is often hard to be sure that the case with the largest possible execution time is tested. In contrast, Bound-T analyses the program code *statically* and considers *all* possible cases or paths of execution. Bound-T bounds are sure to contain the worst case.

Static analysis – no hardware required

Since Bound-T analyses rather than executes the target program, target-processor hardware is not required. With the static analysis approach, timing constraints can be verified without complicated test harnesses, environment simulations or other tools that you would need for really running the target program.

Of course, thorough software-development processes should include testing, but with Bound-T the timing can be verified early, before the full test environment becomes available. In many embedded-system development projects the hardware is not available until late in the project, but Bound-T can be used as soon as some parts of the embedded target program are written.

It’s impossible, but we do it with assertions

The task Bound-T tries to solve is generally impossible to automate fully. Finding out how quickly the target program will finish is harder than finding out if it will *ever* finish – the famously unsolvable “halting problem”. For brevity and clarity, this manual generally omits to mention the possibility of unsolvable cases. So, when we say that Bound-T will do such and such, it is always with the implied assumption that the problem is analysable and solvable with the algorithms currently implemented in Bound-T.

For difficult target programs, the user can always control and support Bound-T's automatic analysis by giving *assertions*. An assertion is a statement about the target program that the user knows to be true and that bounds some crucial aspect of the program's behaviour, for example the maximum number of a times a certain loop is repeated.

Approximations

Also bear in mind that Bound-T produces an upper bound for the execution time, which may be different from the exact worst-case time. Various approximations in Bound-T's analysis algorithms may give over-estimated, too conservative bounds. However, the bounds can be sharpened by suitable assertions.

Context and place

The figure below illustrates the context in which Bound-T is used. The inputs are the compiled, linked executable target program and an optional file of assertions. Another optional input file shows the positions of marks in the source-code; assertions can use such marks to identify the program parts (subprograms, loops, ...) to which the assertions apply. The input from command-line arguments and options is not shown in the figure.

The outputs are the bounds on execution time and stack usage (optional), as well as control-flow graphs and call graphs (also optional).

The present document defines the language in which are written the assertions that, in the figure, enter the Bound-T tool from the left.

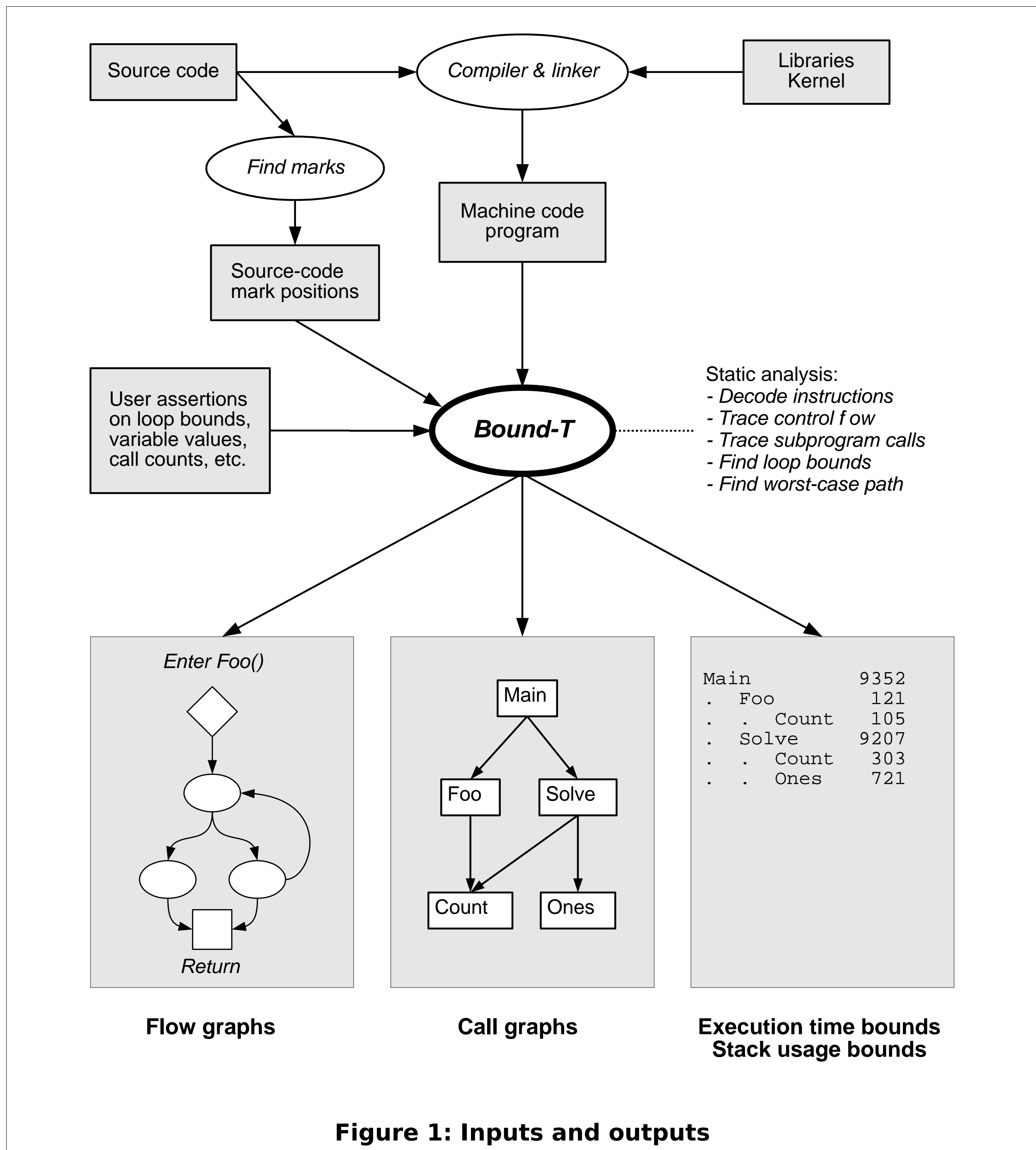

Figure 1: Inputs and outputs

1.2 Overview of this document

What the reader should know

This document defines and explains the Bound-T *assertion language* – the text that a user can give to Bound-T, in addition to the target program code, to support and constrain the analysis of the execution time and stack usage of the target program. The reader is assumed to be familiar with the general usage of Bound-T – for example from the Bound-T User Guide, at *http://www.bound-t.com/manuals/user-guide.pdf* – and to know how to program in some

common procedural (imperative) language, such as C or Ada. Familiarity with real-time and embedded systems is an advantage. Most examples in the manual are presented in C, but Bound-T is independent of the programming language, since it works on the executable machine code.

Target program, target processor

To use Bound-T effectively, the user must also know the structure of the *target program* – the program being analysed. In some cases, the user also needs to understand the architecture of the *target processor* that will run the target program.

Assertion language manual overview

This document is organised into chapters as follows:

- Chapter 2 shows in a tutorial and example-driven way how to write assertions to control and support Bound-T.
- Chapter 3 explains how to identify parts of the program – subprograms, loops, calls – for placing assertions on them.
- Chapter 4 shows how to use assertions to handle two special cases: eternal loops, and recursive subprograms.
- Chapter 5 defines the formal syntax and meaning of the assertion language.
- Chapter 6 lists all warning messages and error messages from the assertion parser, with explanations and advice on solving the problems.

1.3 Other Bound-T documentation

This manual is supplemented by other Bound-T documentation as follows.

User Guide

The Bound-T User Guide at *http://www.bound-t.com/manuals/user-guide.pdf* introduces Bound-T's features and usage in an informal, tutorial way. It gives several simple examples of assertions. Read the User Guide to get started, then return to this manual for more examples and for the full definition of the assertion language. The User Guide ends with a glossary of terms and concepts related to Bound-T that you may find helpful.

Reference Manual

The Bound-T Reference Manual at *http://www.bound-t.com/manuals/ref-manual.pdf* explains all the command-line options, including those that tell Bound-T to use certain assertions (assertion files) in the analysis. The Reference Manual also explains the outputs from Bound-T, including some warning and error messages that may be related to problems in the assertions. Still, for most assertion-related problems you should find the warning and error messages in the present manual, in chapter 6.

Target-Specific Application Notes

Bound-T is available for several target processors, with a specific version of Bound-T for each processor. Most features of the Bound-T assertion language are general and target-independent, but there are some details that can be target-specific:

- The form of target-program identifiers – the names of subprograms, variables, and labels – may depend on the target programming language and the cross-compiler, for example, through compiler-specific “name mangling”.
- The set of target-processor registers and their names is evidently target-specific, and affects assertions on the usage and values of registers.
- Some assertions may refer to numeric addresses of code or data in the target computer. The form of numeric addresses is target-specific.
- For each target processor Bound-T defines a set of “properties” that can be asserted to have certain values for certain parts of the target program. The names and meaning of such properties are target-specific.
- Assertions on execution time or stack usage use target-specific units (cycles, storage units). Their meaning in absolute units (seconds, bits, or octets) is target-specific.

Most target-specific aspects of assertions appear only in string literals: expressions of the form “_foo” that, for example, give the name of a subprogram as a string in quotes. The string may have to be different for different targets and cross-compilers – for example, some C cross-compilers put an underscore '_' in front of C identifiers, others do not – but the higher-level form and meaning of the assertion is the same.

Additional information for specific targets is given in separate Bound-T *Application Notes*. Please refer to *http://www.bound-t.com/app_notes* for a list of the currently supported target processors and the available Application Notes.

User Manual for find_marks

An assertion must identify the part(s) of the target program to which the assertion applies. One way to identify a program part is to insert a *mark* in the source code at that part. Tidorum provides a program called *find_marks* that scans the source-code files, finds the marks, and creates a mark-definition file for Bound-T. The User Manual for *find_marks* explains how to write marks and how to use or modify this program, which is provided under the GNU Public Licence (GPL). The manual also defines the format of the mark-definition files – necessary information should you decide to write your own mark-finder program. Please refer to *http://www.bound-t.com/manuals/find-marks-manual.pdf* for this manual.

Hard Real Time Programming Model

Bound-T contains special high-level support for target programs that follow the *Hard-Real-Time (HRT)* programming model, an architectural style for concurrent, real-time programs originally defined by the European Space Agency. There is a separate manual that explains how to use Bound-T in HRT mode. See *http://www.bound-t.com/manuals/hrt-manual.pdf*. However, the assertion language is entirely independent of whether Bound-T is used in basic mode or in HRT mode.

2 WRITING ASSERTIONS

2.1 What assertions are

When the looping structure of the target program is too complex for Bound-T to find good loop bounds automatically, the user can help with *user assertions* that fill in the gaps in the automatic analysis. These assertions can directly state loop-repetition bounds or other constraints on the execution paths. The assertions can also, or instead, state constraints on variable values or other items from which automatic analysis can derive loop bounds and other bounds on the execution path.

Assertions can also improve the *precision* of the automatic analysis by making the computed worst-case time-bounds closer to the real worst-case times. For example, an assertion can limit the number of times a computationally heavy branch in a conditional statement in a loop is chosen, giving a realistic mix of light and heavy executions of the statement.

Embedded control programs often have several “modes” of execution. For example, the attitude-control software on a spacecraft may have a safe mode, a coarse-pointing mode and a fine-pointing mode. The active software tasks, their activation frequencies and their execution paths can be quite different for different modes. Thus, the worst-case execution time analysis and schedulability analysis should be done separately for each mode. You can use assertions to select the mode-specific execution paths.

Finally, you can use assertions to analyse special cases such as cases where the target program has empty inputs or invalid inputs. Sometimes it is useful to know the execution time of such special cases even if it is much less than the execution time of normal cases.

The assertion file

You write assertions in a text file, using the text editor of your choice, and use the option *-assert filename* to tell Bound-T to use this assertion file in the analysis. You can repeat this option several times to combine assertions from several files in the same analysis.

The Bound-T User Guide shows some simple examples of assertions. The present chapter introduces the full assertion language by description and more examples. Chapter 3 goes into more detail on how to focus your assertions on certain subprograms or other parts of the program. Chapter 4 talks about the special case of eternal loops and of recursive subprograms. Finally, chapter 5 defines the formal syntax and meaning of the assertion language.

The assertion language is “free format” and treats line separators and comments as white-space. White-space can appear between any two lexical tokens (keywords, numbers, strings). You can thus lay out and indent the assertion text as you please. The examples in this chapter generally use indentation systematically but divide the text into lines less systematically, depending on the length and structure of the assertion text.

It is a good idea to motivate and describe you assertions in the assertion file. Comments can be written anywhere in the file where white-space can appear. A comment begins with a double hyphen "--" and extends to the end of the line.

Target-specific issues

The assertion language is designed to be generic and independent of the target processor. Nevertheless, the types of assertion that can be handled may depend on the target, in particular on the compiler and linker and on the form and content of the symbolic debugging information in the executable file.

The target-specific Application Notes explain such limitations, which may also depend on the target compiler options, such as the optimisation level.

While the assertion language is generic, the target processor and the target programming tools define how assertions should refer to subprograms and variables by name or by machine-level address. The target-specific Application Notes explain the naming rules. Likewise, the units of execution time (cycles) and stack usage (storage units) are target-specific and explained in the Application Notes.

Assertion pre-processing

Bound-T reads the assertions from one or more optional input text files named with the *-assert* option. It may be convenient to combine assertions from several files, for example if the program uses libraries for which assertion files already exist. However, for reusable libraries the assertions must often use different numbers, for example different loop bounds, depending on the application that uses the library. For such cases we recommend that a preprocessor such as *cpp* or *m4* be used to preprocess the assertion files. This will allow the use of macros (*#defines*) to parametrise the assertions, for example by the size of the input-data assumed in the worst-case scenario.

2.2 Assertion = context + fact

An assertion expresses some *fact* that holds in or for some *context*, within the target program under analysis.

Facts

The following sorts of facts can be asserted:

- variable value range (minimum, maximum or both),
- number of repetitions of a loop,
- number of executions of a call, loop entry, or other instruction,
- worst-case execution time of a subprogram or a call,
- worst-case stack usage of a subprogram,
- final stack height of a subprogram,
- the possible callees of a dynamic (indirect, computed) call,
- invariance (constancy) of a variable in a part of the program,
- volatility of a variable or a range of memory locations,
- value or range for some target-specific property in a part of the program,
- the target-specific execution role performed by an instruction.

Contexts

The following sorts of contexts can be used in assertions:

- the whole program (global context),
- a subprogram,
- a set of loops,
- a set of calls,
- a single instruction.

When the context is a single instruction, the instruction is identified by its machine address or by its offset, in machine-address units, from the start of the subprogram that contains the instruction.

When the context is a set of loops or calls, these loops or calls are identified by syntactic or semantic properties. Nesting the loop or call context within a subprogram context limits the set of loops or calls to those within this subprogram. Otherwise the set can contain loops or calls from any subprogram.

When the context is a subprogram, the subprogram is identified by its link-name which is usually the same as the source-code identifier or name of the subprogram, perhaps more or less altered by “mangling”. A subprogram can also be identified by its machine address.

There are actually two kinds of subprogram context: the subprogram *entry point*, where facts about the initial state (entry state) can be asserted, and the subprogram *body*, where facts that hold throughout the subprogram can be asserted. Figure 2 below shows an example assertion file and points out the different kinds of facts and contexts in the file. Note that the figure does not include all the possible kinds of assertions.

The rest of this chapter discusses and gives examples of most types of assertions. We will first focus on the facts that can be asserted and the allowed combinations of fact and context. Then we will show in more detail how to define contexts, in particular loop and call contexts. For simplicity we will assume execution time (WCET) analysis but many of the examples are valid also for stack usage analysis. Stack usage analysis usually requires fewer assertions because loops do not have to be bounded.

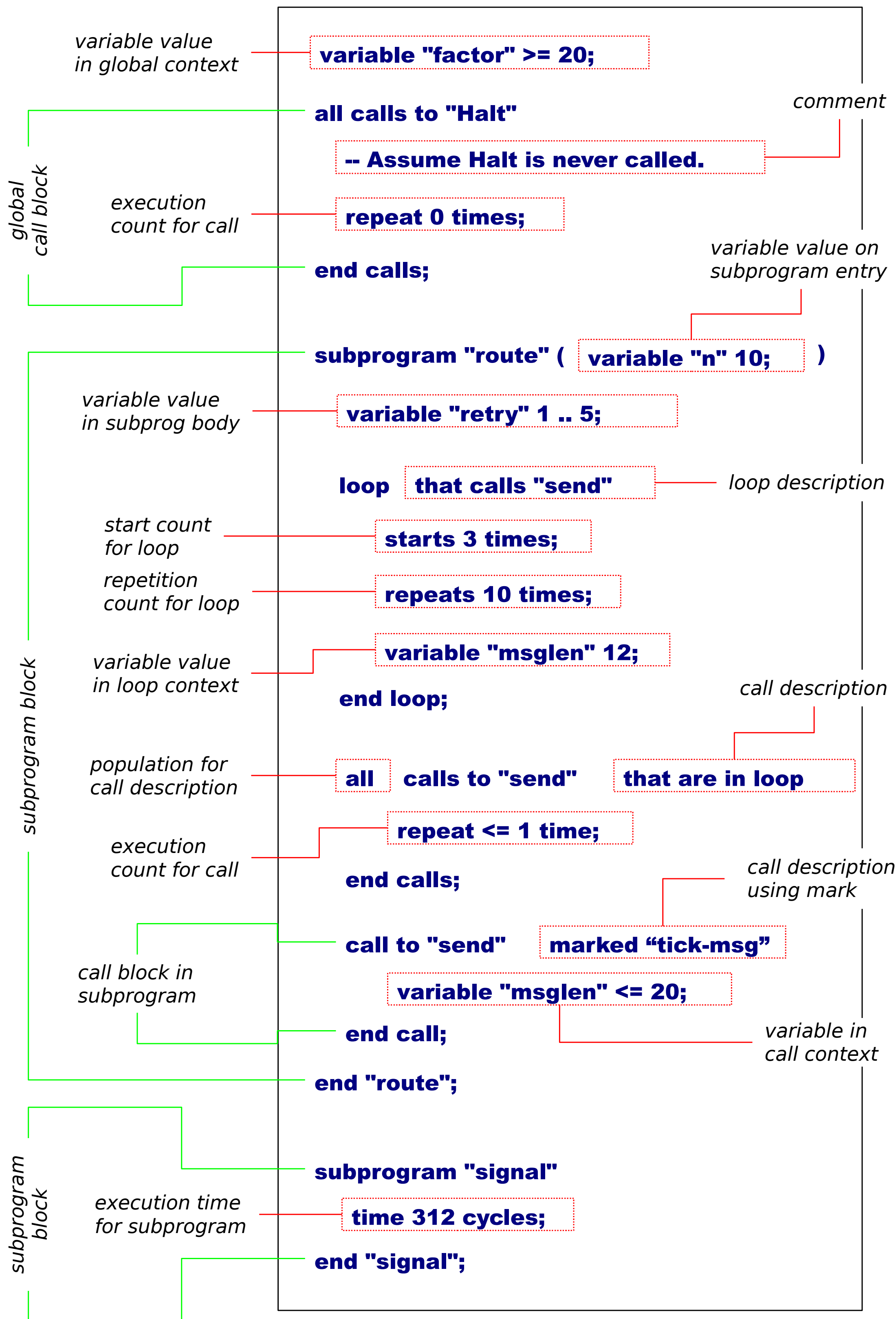

Figure 2: An assertion file

2.3 Assertions on the repetition of loops

Why?

A repetition assertion for a loop bounds the number of times the loop is repeated (iterated) each time execution reaches this loop. The form of the assertion is "repeats *N* times" where *N* is a number or a range of numbers. (There is a nice point about which parts of the loop are repeated this number of times; see section 5.13 for an exact definition.)

For analysing execution time, you *must* give assert a repetition bound for each loop that Bound-T does not bound automatically. Even for automatically bounded loops you may use a repetition assertion to set a smaller repetition bound if the automatically determined bound is unrealistically large.

Consider unrolling

Bound-T applies loop-repetition assertions to the machine-code form of the loop. There are several compile-time code optimizations that can alter the number of repetitions of the loop. For example, if the source-code loop copies 40 octets from one place to another, the compiler may decide to "unroll" the loop so that it instead copies 20 words of 16 bits or 10 words of 32 bits. The source-code loop-bound of 40 may correspond to a loop bound of 20 or 10 in the machine code. Other optimizations or code transformations may reduce the number of repetitions by one or change it in other ways.

Thus, to assert the correct repetition count you should look at the machine code and not only at the source code. See section 5.13 for a precise explanation of the meaning of a repetition-count assertion for a loop. Instead of loop-repetition assertions you could try to help the automatic loop-bound analysis by asserting bounds on variable values as explained in section 2.12 below.

Looping in a subprogram

The most common assertion bounds the number of repetitions of a loop in a subprogram. The assertion must identify the subprogram (usually by name), the loop (or loops) in question (usually by some properties of the loops) and state the number of repetitions. Thus, the assertion consists of a "subprogram block" that contains a "loop block" that contains an repetition-count fact.

Here is how to assert that in the subprogram *Reverse_List* the two loops that call *Swap_Links* repeat (each) exactly 100 times:

```
subprogram "Reverse_List"
   all 2 loops that call "Swap_Links"
      repeat 100 times;
   end loops;
end "Reverse_List";
```

The part "all 2" says that we expect this assertion to match exactly two loops in *Reverse_List*. If there are less than two or more than two loops that call *Swap_Links* Bound-T will report an error in the assertion-matching phase.

Looping in any subprogram

If loops with certain properties have the same repetition bound in all subprograms, the same loop-block can be made to apply in all subprograms by writing the loop-block alone, in the global context, without an enclosing subprogram block.

Here is how to assert that any loop in the whole program (or the part of the program we are now analysing) that uses (reads) the variable *Polling_Count* repeats at most 24 times:

```
all loops that use "Polling_Count"
    repeat <= 24 times;
end loops;
```

The keyword **all** (without a following number) means that any number of loops can match this assertion.

Nested loops

Bound-T analyses nested loops independently. It may find bounds for all, none or any levels of a loop nest, so you may need to help by asserting bounds for all, none or any levels. The level that an assertion addresses is identified by saying whether the loop contains or is contained in another loop.

For example, assuming that the subprogram *Add_Matrix* contains a two-level loop nest, that is an outer loop that contains an inner loop, here is how to assert that the outer loop repeats 10 times and the inner, 20 times:

```
subprogram "Add_Matrix"
    loop that contains loop  -- The outer loop.
        repeats 10 times;
    end loop;
    loop that is in loop     -- The inner loop.
        repeats 20 times;
    end loop;
end "Add_Matrix";
```

For deeper nesting, the descriptions “contains loop” and “is in loop” must be extended to describe the nesting of the inner or outer loop, too. For example, here is how to assert bounds on a three-level loop nest in the subprogram *Multiply_Matrix*:

```
subprogram "Multiply_Matrix"
    loop that contains (loop that contains loop)  -- Outermost loop.
        repeats 10 times;
    end loop;
    loop that contains loop and is in loop        -- Middle loop.
        repeats 15 times;
    end loop;
    loop that is in (loop that is in loop)        -- Innermost loop.
        repeats 20 times;
    end loop;
end "Multiply_Matrix";
```

Non-rectangular loops

In some nested loops, the number of repetitions of the inner loop is not constant but depends on the iteration number of the outer loop. For example, here is an Ada loop that traverses the "lower triangle" of a 100×100 matrix *M*:

```
for I in 1 .. 100 loop
   for J in 1 .. I loop  -- Note the upper bound!
      Traverse (M(I,J));
   end loop;
end loop;
```

The outer loop repeats 100 times. The inner loop repeats *I* times where *I* is the counter for the outer loop. On the first iteration of the outer loop ($I = 1$) the inner loop repeats once; on the last iteration of the outer loop ($I = 100$) the inner loop repeats 100 times. At present, it is not possible assert such a variable bound for the inner loop, nor can Bound-T deduce such a variable bound automatically.

As a work-around, you can assert an "average" bound on the inner loop such that the total number of repetitions of the inner loop is correct, or close. In this example, when the outer loop is finished the inner loop has been repeated a total of $100 \times (100 + 1) / 2 = 5050$ times. Thus, the average number of repetitions of the inner loop for each repetition of the outer loop is $5050 / 100 = 50.5$. The closest possible assertion is 51 repetitions:

```
loop that is in loop repeats 51 times; end loop;
```

This assertion corresponds to a total of $51 \times 100 = 5100$ repetitions of the inner loop, an overestimation of 50 repetitions compared to the true number of 5050 repetitions.

In this example you can remove this overestimation because the inner loop always calls the subprogram *Traverse* and you can assert the total number of times this call occurs as follows:

```
call to "Traverse" repeats 5050 times; end call;
```

This assertion makes Bound-T compute a WCET bound that corresponds to exactly 5050 repetitions of the inner loop. (However, you also have to make the loop assertion unless Bound-T bounded the loop automatically.) The next section shows more examples of execution-count assertions for calls.

2.4 Assertions on the number of loop starts

Why?

We say that a loop *starts* when execution reaches the loop head, after which the loop repeats zero or more times. For example, when the following code is executed the outer loop starts once and the inner loop starts three times (when the outer-loop index *I* has the values 2, 4, 6):

```
for I in 1 .. 7 loop
   Foo (I);
   if I mod 2 = 0 then
      for J in 1 .. 5 loop
         Bar (I, J);
      end loop;
   end if;
```

```
end loop;
```

A start-count assertion on a loop defines the number of times the loop starts in each execution of the subprogram that contains the loop. The form of the assertion is “starts *N* times” where *N* is a number or a range of numbers.

It is never *necessary* to assert the start count of loops, because Bound-T can determine a finite WCET bound without such assertions as long as the number of *repetitions* of all loops are bounded (automatically or by assertions). However, start-count assertions on loops can often *improve* (sharpen) the WCET bound. Without such assertions, the WCET bound may include an unrealistically (infeasibly) large number of executions of those loops, or even some loops that should not be included at all because they represent a scenario that you want to exclude from the analysis.

Start inner loop a given number of times

Consider the example above, where the inner loop (*for J...*) is entered only for even values of the index *I* of the outer loop, which happens three times as the outer-loop index runs from 1 to 7. However, at present Bound-T does not discover this on its own, so it will assume that the inner loop starts and runs on every iteration of the outer loop, which leads to an over-estimated execution-time bound. Assuming that these loops lie in the subprogram *Nested*, the following assertion text advises Bound-T that the inner loop starts only three times in each execution of *Nested*:

```
subprogram “Nested”
   loop that is in loop starts 3 times; end loop;
end “Nested”;
```

Don't enter that loop at all

You can use a start-count assertion to exclude a certain loop from the analysis by asserting that the loop starts zero times. For example, assume that the program under analysis, in some cases, must wait for some event to happen, and does so by polling the function *Event_Ready*. That is, here and there in the program we find loops of this form:

```
if <some condition> then
   loop
      exit when Event_Ready;
   end loop;
end if;
```

If you want to analyse the program under the assumption that this polling never happens, you can use the following assertion (with the global context) to exclude all these loops:

```
all loops that call “Event_Ready” start 0 times; end loops;
```

Careful with null checks and unpeeling

Start-count assertions can fail or have a distorted effect if the compiler changes the structure of the loop in a way that makes entry to parts of the loop conditional. Consider for example this simple C loop:

```
for (i = 0; i < m; i++) DoThat(i);
```

A simple compiler may generate code for this loop with the simple control-flow graph shown in Figure 3 below.

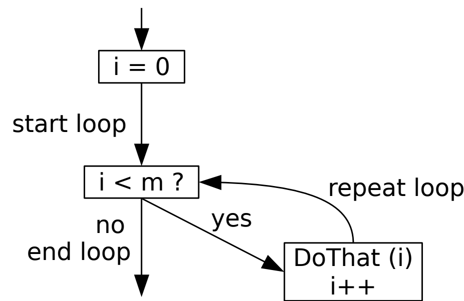

Figure 3: Starting a loop, no unpeeling, no null check

Here, a start-count assertion bounds the number of executions of the edge marked “start loop” in Figure 3. This does not depend on the value of *m*, which is what one would expect from the source-code form of the loop.

However, if the compiler prefers to generate “bottom-test” code the machine code corresponds to the following C form:

```
i = 0;
if (i < m) do {DoThat(i); i++} while (i < m);
```

The control-flow graph then has the shape shown in Figure 4 below.

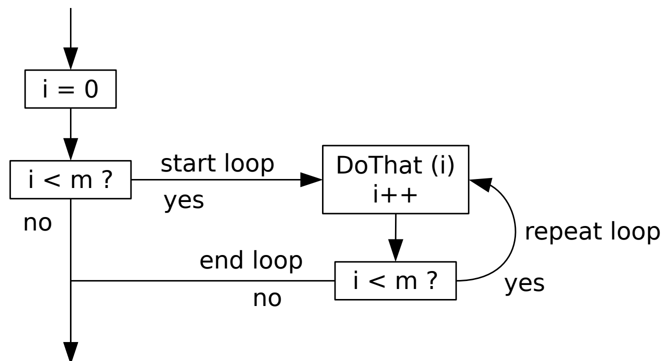

Figure 4: Starting a loop with null check

The “start loop” edge is now executed only when $m > 0$, which means that a start-count assertion on this loop does not bound the number of times this code is entered with $m < 1$.

Furthermore, the compiler may decide to unpeel the first iteration, and then has to add more checks on the value of *m*. If the compiler also prefers to generate “bottom-test” code the machine code corresponds to the following C form:

```
i = 0;
if (i < m) {
    DoThat(i);
    i++;
    if (i < m) do {DoThat(i); i++} while (i < m);
}
```

The control-flow graph then has the complex shape shown in Figure 5 below.

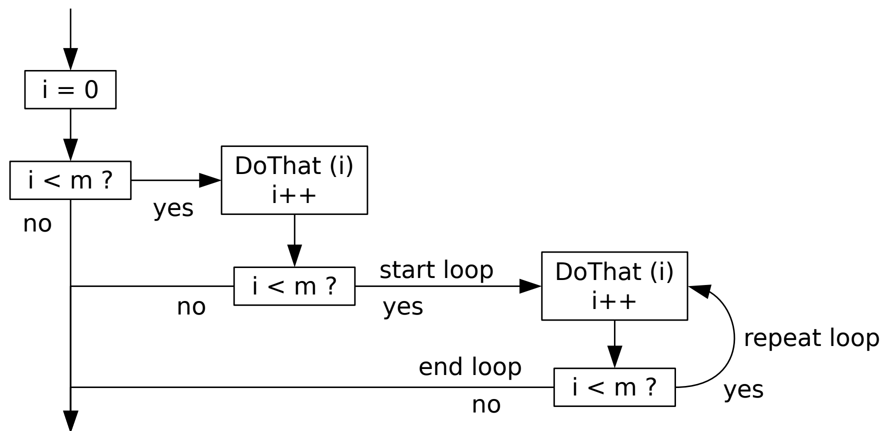

Figure 5: Starting an unpeeled loop with null check

Note how the first iteration (i = 0) of the loop is not actually within the loop in the control-flow graph, which contains only the second and later iterations (i > 0). The "start loop" edge is executed only when m is at least 2, which means that a start-count assertion on this loop covers does not bound the number of times this code is entered with m < 2.

2.5 Assertions on the execution count of calls

Why?

An execution-count assertion for a call defines the number of times the call is executed in each execution of the caller. The form of the assertion is "repeats *N* times" where *N* is a number or a range of numbers.

It is never *necessary* to assert the execution count of calls, because Bound-T can determine a finite WCET bound without such assertions as long as all loops are bounded (automatically or by assertions). However, execution-count assertions on calls can often *improve* (sharpen) the WCET bound. Without such assertions, the WCET bound may include an unrealistically (infeasibly) large number of some calls, or even some calls that should not be included at all because they represent a scenario that you want to exclude from the analysis.

You can use an execution-count assertion for a call to exclude certain execution paths completely, or to limit the number of times certain execution paths are taken within loops. However, sometimes a better way may be to help the automatic control-flow analysis by asserting bounds on variable values as explained in section 2.12 below.

Don't take that path in that subprogram

Perhaps the most common assertion of this type is to exclude a certain path in a subprogram by asserting that a call in that path is executed zero times. The usual reason for this is to exclude certain unusual scenarios from the worst-case analysis.

The assertion must identify the subprogram (usually by name), the calls in question (usually by the name of the callee) and state that the call is executed zero times. Thus, the assertion consists of a "subprogram block" that contains a "call block" that contains an execution-count assertion.

Assume that the subprogram *Invert_Matrix* calls the subprogram *Report_Singularity* if it detects an error. The following asserts that no such call is never executed, in other words, that the error case is excluded from the analysis:

```
subprogram "Invert_Matrix"
    all calls to "Report_Singularity"
        repeat 0 times;
    end calls;
end "Invert_Matrix";
```

The resulting WCET bound for *Invert_Matrix* will not include execution paths that involve calls to *Report_Singularity*.

Don't take that path in any subprogram

When all calls to a certain callee subprogram should be excluded everywhere (from all caller subprograms) the easiest way is to mark the callee subprogram *unused* as explained in section 2.17. That will also prevent the (useless) analysis of the callee subprogram itself.

If you want to exclude all calls to a subprogram from the analysis of the callers, but still want to analyse the callee subprogram itself, the call-block and its execution-count assertion can be made to apply in all caller subprograms by writing the call-block alone (in global context) without an enclosing subprogram block. Here is how to assert that the subprogram *Halt* is never called:

```
all calls to "Halt"
    repeats 0 times;
end calls;
```

Whenever Bound-T finds a call to *Halt* this assertion makes the execution path that leads to the call infeasible. However, Bound-T will still analyse the *Halt* subprogram itself, although this analysis is not needed for the analysis of callers. To prevent this useless analysis, also give an execution-time assertion for *Halt*, for example

```
subprogram "Halt" time 0 cycles; end "Halt";
```

or simply assert that *Halt* is unused:

```
subprogram "Halt" unused; end "Halt";
```

and then you can drop the assertion on calls to *Halt* as redundant.

Don't call every time

It is common for loop bodies to include call statements that are conditional, so they are not necessarily executed on every iteration of the loop. If there are no assertions to prevent it, Bound-T will compute a WCET bound that assumes that every loop iteration takes the longest path through the loop-body. If the longest path includes a conditional call that in reality is executed rarely, for example only once for every 100 loop iterations, the WCET bound may be strongly overestimated. To make the WCET bound more precise, you can assert a smaller execution count on the call.

Assume that the subprogram *Emit_Message* contains a loop that stores bytes in a buffer one by one and calls *Flush_Buffer* when the buffer becomes full, as in the following Ada-like pseudocode:

```
procedure Emit_Message is
begin
   for K in 1 .. Message_Length loop
      put message byte number K in Buffer;
      if Buffer is full then
         Flush_Buffer;
      end if;
   end loop;
   Flush_Buffer;
end Emit_Message;
```

(The purpose of the final call to *Flush_Buffer* is to emit the partially filled buffer.) Assume that *Message_Length* is at most 1000 and that the *Buffer* can hold up to 100 bytes. The longest path through the loop body includes the call of *Flush_Buffer*, so by default the WCET bound for the loop will include 1001 calls of *Flush_Buffer* (1000 in the loop plus the one at the end). However, at most 11 calls can occur in a real execution (10 in the loop plus the one at the end). The WCET bound will probably become much more accurate if we assert this:

```
subprogram "Emit_Message"
   call to "Flush_Buffer" that is in loop
      repeats <= 10 times;
   end call;
end "Emit_Message";
```

Note that this assertion does not apply to the last call of *Flush_Buffer* because it specifies the call property "is in loop". However, the effect would be the same without this restriction because the automatic analysis knows that the last call executes once, so an additional assertion that it executes at most 10 times has no effect.

No totalisation

We can build on the last example, *Emit_Message* and *Flush_Buffer*, to illustrate a short-coming of the current assertion language. A real implementation of *Emit_Message* could be more complex and have several statements that put bytes in the *Buffer* followed by conditional calls to *Flush_Buffer*. For example, the message might be divided into a header and a trailer with one loop generating the header and another loop generating the trailer. If the header and trailer lengths can vary independently, but the total message length is still at most 1000 bytes, we know that the total number of calls of *Flush_Buffer* is still at most 10, but we cannot assert this because an assertion like

```
all calls to "Flush_Buffer" repeat <= 10 times; end calls;
```

applies separately to *each* statement that calls *Flush_Buffer*. The call in the header loop will contribute 10 calls to the WCET bound and so will the call in the trailer loop, for a total of 20 *Flush_Buffer* calls in the WCET bound for *Emit_Message*.

You can work around this problem by asserting a smaller number of call repetitions, for example 5 repetitions for each call.

2.6 Assertions on the execution count of an instruction

Why?

An execution-count assertion for an instruction defines the number of times the instruction is executed in each execution of the subprogram that contains this instruction. The form of the assertion is “repeats *N* times” where *N* is a number or a range of numbers.

The reasons for using such assertions are the same as for start-count assertions on loops or execution-count assertions on calls, just explained in section 2.5. In fact, any execution-count assertion for a call is equivalent to the corresponding execution-count assertion for the call instruction. (The same is not true for loop-start assertions because there may be several paths to the loop head, in which case there is no single instruction that “starts” the loop.)

Why not?

The identification of calls (using symbolic names) is much easier and more robust than the identification of instructions (using addresses or offsets). Thus, you should use execution-count assertions for instructions only as a last resort when you need to limit the analysis to paths through or around this instruction and only if the same cannot be done with execution-count assertions for calls because there are no suitable (and identifiable) calls in this part of the target program.

Where?

An instruction block in the Bound-T assertion language can be global (not contained in a subprogram block) or lie within a subprogram block. The placement has some effects on the syntax and meaning of the instruction block, as follows.

For a global instruction block:

- the instruction must be identified by its absolute address, not by an offset;
- the assertions apply to the analysis of all subprograms that contain the instruction.

For an instruction block that is nested in a subprogram block:

- the instruction can be identified by an absolute address or by an offset (from the entry point of the subprogram named in the containing subprogram block);
- the assertions apply only to the analysis of the subprogram named in the containing subprogram block.

You may wonder how a given instruction can be contained in more than one subprogram. This is not uncommon. The most common cause is the optimization of tail calls into jumps, in which case Bound-T usually considers the tail callee to be a part of the calling subprogram, as if the tail call were “integrated” (see section 2.17). The instructions in the tail callee are then considered to be contained in all the callers. The same happens, of course, if there is an explicit **integrate** assertion or if Bound-T by default integrates certain routines, such as prologues and epilogues, into their callers.

Integrated routines can also make the same instruction appear several times in the same subprogram. Execution-count assertions on such instructions are rejected with an error message because the mapping from the instruction address to a place in the subprogram's control-flow graph is ambiguous.

How to find the address or offset of an instruction

Here are some ways:

- Use compilation options that create a listing file including assembly code. Such files usually give the offsets from the start of the subprogram (or from the start of the module) to each instruction.
- Disassemble the target program. This will show absolute instruction addresses. One way is to run Bound-T with the option *-trace decode*.
- Make Bound-T draw the control-flow graph of the subprogram: use the options *-dot* and *-draw*. The option *-draw decode* makes Bound-T show the disassembled instructions and their absolute addresses in the basic blocks of the graph. Use also the option *-draw cond* to show the condition-flag values for each conditional control-flow edge.

The last method has the advantage that it shows the places of all instructions in the control flow, which is necessary knowledge for controlling the analysis with execution-count assertions on instructions.

2.7 Assertions on the execution time of a subprogram

Why?

If you assert an execution time for a subprogram Bound-T will not analyse the subprogram at all (unless it needs to do so for the stack analysis). Instead, Bound-T assumes that any call to this subprogram takes the asserted time. There are several situations in which this is useful:

- The subprogram is not yet implemented, but it has an execution time budget and you want to analyse the overall execution time under this budget.
- Bound-T cannot analyse the subprogram for some reason (for example due to an irreducible flow-graph or recursive calls), but the subprogram's execution time has been determined in some other way.
- The subprogram was already analysed and its WCET bound is known, but you do not want to re-analyse the subprogram, perhaps because the analysis takes a long time. For example, library subprograms or kernel subprograms may be handled in this way.

If you know that the subprogram is never called, and so there is no need to analyse it, you should assert that the subprogram is **unused**; see section 2.17. If you want to exclude the subprogram from the analysis, but do not want to assert an execution time for it, you should use the **omit** assertion; see section 5.6.

Time of a subprogram

An assertion of this kind consists of a subprogram block that contains a time fact. Here is how to assert that any call of the subprogram *Change_Priority* takes 23 cycles:

```
subprogram "Change_Priority"
   time 23 cycles;
end "Change_Priority";
```

2.8 Assertions on the execution time of a call

Why?

The execution time of subprograms often depends on the calling context. Bound-T can sometimes analyse this dependency automatically, for example when loop-bounds depend on parameter values in a simple way. When an automatic context-dependent analysis is not possible you can assert a context-dependent execution time manually, by asserting the execution time of a specific call of the subprogram. This makes the overall WCET bound more accurate than if a context-independent worst-case time were used for all calls.

You would typically determine the execution time for a specific call by analysing the subprogram separately under specific assertions for this call. For example, you may assert that some paths in the subprogram cannot occur in this call. Then you translate the resulting WCET bound into an execution-time assertion for the call and analyse the caller under this assertion.

Calling from one subprogram

Suppose that the subprogram *Find_Angle* contains a conditional call to *Reduce_Argument* as in the following C code:

```
void Find_Angle (double arg; double *angle)
{
    if (fabs(arg) > PI) Reduce_Argument (&arg);
    *angle = Find_Normal_Angle (arg);
}
```

The execution time of a call to *Find_Angle* may depend greatly on whether or not it actually calls *Reduce_Argument*, that is, on the magnitude of the *arg* parameter. However, Bound-T does not analyse floating-point computations and so it cannot solve this context dependency and will use the worst-case time (including *Reduce_Argument*) for all calls of *Find_Angle*. On the other hand, for a given call of *Find_Angle* you may know that *arg* will be small enough so that *Reduce_Argument* is not called. For example, such a constraint may be a precondition as in the following subprogram:

```
void Compute_Shadows (double *args[]; double main_arg)
/* Precondition: All args[0..255] are between -PI and PI. */
/* Note that this precondition does not apply to main_arg. */
{   double angles[256], main_angle;
    ...
    Find_Angle (main_arg, &main_angle);
    ...
    for (i=0; i<255; i++) Find_Angle (args[i], &(angles[i]));
    ...
}
```

The subprogram *Compute_Shadows* contains two calls to *Find_Angle*. The first call (before the loop, for *main_arg*) may call *Reduce_Argument*. The assumed precondition on the *args* parameter means that the second call (in the loop) never leads to a call of *Reduce_Argument*. This means that the WCET bound for *Compute_Shadows* may be greatly over-estimated if the context-independent WCET bound for *Find_Angle* is used for both calls.

To make a context-dependent analysis, analyse *Find_Angle* separately (that is, as a root subprogram) under an assertion that excludes the call of *Reduce_Argument*:

```
subprogram "Find_Angle"
   call to "Reduce_Argument" repeats 0 times; end call;
end "Find_Angle";
```

Assume that this gives a WCET bound of 127 cycles for *Find_Angle*. Now we can analyse *Compute_Shadows* with a context-specific time for the second call of *Find_Angle*. The context of the assertion is *Compute_Shadows* and this call, while the asserted fact is the execution time of the call:

```
subprogram "Compute_Shadows"
   call to "Find_Angle" that is in loop
      time 127 cycles;
   end call;
end "Compute_Shadows";
```

Thanks to the part “that is in loop” the execution time of 127 cycles applies only to the *Find_Angle* call that is in the loop where we know that *Reduce_Argument* is not called. The other call (for *main_arg*) uses the context-independent WCET bound for *Find_Angle* that includes a possible call to *Reduce_Argument*. If this bound for *Find_Angle* is 321 cycles, for example, the context-dependent analysis improves the WCET bound for *Compute_Shadows* by $256 \times (321 - 127) = 49\,664$ cycles.

Calling from any subprogram

If the same execution time assertion should apply to all calls with certain properties within any subprogram, the call-block and time-fact can be written in a global context and not within a subprogram block. For example, here is how to assert that anywhere in the program, any call of the subprogram *Copy_Block* that is executed within a loop that defines (writes to, assigns to) the variable *short_counter* takes at most 912 cycles:

```
all calls to "Copy_Block"
   that are in (loop that defines "short_counter")
   time 912 cycles;
end calls;
```

Remember that Bound-T can only detect that a loop defines *short_counter* if the code in the loop uses a static addressing mode to assign a value to *short_counter*, or a dynamic addressing mode that Bound-T can resolve to a static address for *short_counter*.

Problems with manual work

This manual method of context-dependent analysis is not elegant and causes extra work if the program must be analysed again. In the future, Bound-T may offer a way to write specific assertions for the analysis of a callee subprogram in the context of a specific call. Bound-T will then automatically find a specific WCET bound for this call by re-analysing the callee under these assertions.

2.9 Assertions on the stack usage of a subprogram

Why?

Stack usage assertions may be necessary or useful when you use Bound-T for stack analysis. They are never needed for the analysis of execution time.

If you assert the stack usage and the final stack height (see section 2.10) for a subprogram Bound-T will not analyse the subprogram at all (unless it needs to do so for the execution-time analysis). Instead, Bound-T assumes that any call to this subprogram uses the asserted amount of stack space and leaves the stack at the asserted final height. There are several situations in which this is useful:

- The subprogram is not yet implemented, but it has a stack budget and you want to analyse the overall stack usage under this budget.
- Bound-T cannot analyse the subprogram for some reason (for example due to recursive calls), but the subprogram's stack usage has been determined in some other way.
- The subprogram was already analysed and its stack usage bound is known, but you do not want to re-analyse the subprogram, perhaps because the analysis takes a long time. For example, library subprograms or kernel subprograms may be handled in this way.

If the stack in question is a "stable" stack, in other words a stack for which the final stack height is always known to be zero, you do not need to assert the final stack height; it is enough to assert the stack usage to make Bound-T exclude the subprogram from stack analysis.

If you know that the subprogram is never called, and so there is no need to analyse it, you should assert that the subprogram is **unused**. If you want to exclude the subprogram from the analysis, but do not want to assert a stack usage and final stack height for it, you should use the **omit** assertion. See section 2.17 for more about **unused** and **omit**.

Stack usage of a subprogram

An assertion of this kind consists of a subprogram block that contains a stack-usage fact. Here is how to assert that any call of the subprogram *Change_Priority* uses at most 127 (target-specific) units of stack space, assuming that the target program uses only one stack:

```
subprogram "Change_Priority"
    stack usage 127;
end "Change_Priority";
```

Programs with several stacks

When the program has several stacks, the (target-specific) name of the stack must be inserted after the **stack** keyword. Here is the above example for a stack named “SP”, adding the assertion that the usage of another stack, named “Data”, is 32:

```
subprogram "Change_Priority"
    stack “SP”   usage 127;
    stack “Data” usage  32;
end "Change_Priority";
```

The stack name can be included also when this is the only stack in the program. It is advisable to do so when your cross-compiler has an option to use one stack, or several stacks; even if your program starts out using only one stack, it may later evolve to use several stacks, in which case assertions must use the stack name.

2.10 Assertions on the final stack height of a subprogram

Final stack height in stable and unstable stacks

When one subprogram calls another, the final stack height of the *callee* defines the amount by which the call changes the local stack height in the *caller* – in other words, the net change in the stack pointer, over the call. This is important for the analysis of the computations in the caller when these computations refer to data in the caller's stack frame. The program usually refers to such data by means of static offsets to the stack pointer. The effective address (the datum referred to) therefore depends on the value of the stack pointer, so to have a consistent analysis of stack-pointer-based data references before and after a subprogram call one must know by how much the call changes the stack pointer.

Bound-T considers a stack, as used in a target program under analysis, to be either a *stable* stack or an *unstable* stack. The choice may depend on the target processor, on the cross-compiler used to generate the target program, and on the compilation options, as explained in the relevant Application Note documents. A *stable stack* is a stack for which the final stack height on return is always zero, which means that a call has no net effect on the stack pointer. This means that there is no need to assert a final stack height because Bound-T knows it to be zero. An *unstable stack*, in contrast, is a stack for which the final stack height on return may not be zero, which means that a call may have a significant net effect on the stack pointer and thus affect the stack accesses in the caller. For an unstable stack it may be useful to assert the final stack height on return from a specific subprogram.

Why?

Assertions on the final stack height for an unstable stack may be necessary or useful in execution time analysis or stack usage analysis.

When Bound-T analyses a subprogram it always attempts to compute the final stack height for unstable stacks. However, this does not always succeed, or may produce only bounds on the final stack height but not a single value.

Asserting the final stack height of a subprogram is thus useful when:

- The subprogram is omitted from the analysis by an explicit *omit* assertion or because sufficient assertions on its execution time or stack usage make an analysis unnecessary.
- The subprogram changes the stack pointer in ways too complex for Bound-T to analyse, so that Bound-T cannot compute the final stack height.

Still, even in those cases an assertion on the final stack height is necessary only when its absence makes some necessary analysis of the *callers* fail, due to the unknown effect of the call on the stack pointer and thus on the computations in the caller.

If the final stack height of a subprogram is unknown, the local stack height in any caller of this subprogram will be unknown after the call. This usually makes the stack usage and the final stack height unknown for the caller, too, which effectively disables the stack usage analysis.

Final stack height of a subprogram

An assertion of this kind consists of a subprogram block that contains a final-stack-height fact. Here is how to assert that any call of the subprogram *Change_Priority* has a final stack height of -1 stack unit, assuming that the target program uses only one stack:

```
subprogram "Change_Priority"
   stack final -1;
end "Change_Priority";
```

Programs with several stacks

When the program has several stacks, the (target-specific) name of the stack must be inserted after the **stack** keyword. Here is the above example for a stack named “SP”, adding the assertion that the final height of another stack, named “Data”, is zero:

```
subprogram "Change_Priority"
   stack “SP”   final -1;
   stack “Data” final  0;
end "Change_Priority";
```

The stack name can be included also when these is only one stack in the program. It is advisable to do so when your cross-compiler has an option to use one stack, or several stacks; even if your program starts out using only one stack, it may later evolve to use several stacks, in which case assertions must use the stack name.

Combining stack-usage and final-height assertions

Facts on stack usage and final height can be combined in the same clause. For example, here is an assertion that combines the stack usage assertion from section 2.9 with the final-height assertion above:

```
subprogram "Change_Priority"
   stack usage 127 final -1;
end "Change_Priority";
```

and here is the same with the stack names included:

```
subprogram "Change_Priority"
   stack “SP”   usage 127 final -1;
   stack “Data” usage  32 final  0;
end "Change_Priority";
```

2.11 Assertions on the callees of a dynamic call

Why?

Most programming languages support subprogram calls where the called subprogram – the callee – is determined at run-time by some dynamic computation, and not statically at compile-time. Calls of this sort are known as *dynamic calls* in contrast to *static calls*.

On the source-code level static calls state the name (identifier) of the callee directly, while dynamic calls generally dereference a function pointer variable (in C terms) or an access-to-subprogram variable (in Ada terms). In the machine code, a static call instruction defines the entry address of the callee by an immediate (literal) operand, while a dynamic call uses a register operand or other dynamic operand.

A static call has exactly *one* callee; every execution of the call invokes the *same* callee subprogram. In contrast, a dynamic call may invoke different subprograms on each execution, depending on the entry address that is computed, so a dynamic call in general has a *set* of possible callees.

While Bound-T can try to analyse the computation that defines the callee(s) of a dynamic call, this (currently) succeeds only in very simple cases where the dynamic computation is local to the calling subprogram. Thus, to analyse a program that includes dynamic calls, you must usually tell Bound-T what the possible callees are, based on your understanding of the target program.

Where?

An assertion giving the possible callees obviously must be given in the context of a dynamic call. This dynamic call is usually located in a specific subprogram body, but it can, in principle, also be in a global context.

Dynamic call from a subprogram

Here is how to assert that the (only) dynamic call in the subprogram *Take_Action* always calls one of the subprograms *Stop*, *Brake* or *Shut_Down*:

```
subprogram "Take_Action"
   dynamic call calls "Stop" or "Brake" or "Shut_Down";
   end call;
end "Take_Action";
```

Any dynamic call in a certain kind of loop

If an assertion on the callees of a dynamic calls is written in a global context (without specifying the containing subprogram) it is usually necessary to limit its application to calls with some specific properties; otherwise the same assertion would apply to all dynamic calls in the whole program.

As a (contrived) example, the following asserts that when a dynamic call is contained in a loop that (statically) also calls the subprogram *Start_Speed_Change*, then the possible dynamic callees are *Slow_Down* or *Speed_Up*:

```
all dynamic calls
   that are in (loop that calls "Start_Speed_Change")
   call "Slow_Down" or "Speed_Up";
end calls;
```

2.12 Assertions on variable values

Why?

You use assertions to control the execution paths that Bound-T includes in its analysis. As shown in the preceding sections, assertions on the repetition of loops or the execution count of calls give direct control over the path. However, there are some problems with such assertions. Firstly, they require you to study the code of the subprogram under analysis, to identify the loops and calls for which such bounds should be asserted and to compute these bounds yourself. Secondly, Bound-T interprets loop-repetition assertions relative to the machine code of the loop, which means that the assertion should take into account any compiler optimizations as discussed in section 2.3. Optimizations that duplicate code or merge similar

code may duplicate or merge call instructions and should be taken into account in execution-count assertions for calls. Thirdly, it may be hard or impossible to identify (describe) the loop or call context for an assertion because the loops or calls have no distinguishing properties.

You can avoid these problems with direct repetition or execution-count assertions by instead asserting bounds on the values of the variables that determine the execution path, for example the number of loop repetitions, and letting Bound-T's analysis deduce loop-bounds and feasible paths. On the other hand, this indirect control over execution paths works only if the variables determine the path in a way that is simple enough for Bound-T to analyse and if Bound-T actually performs this analysis. In particular, if Bound-T has found a context-independent WCET bound for a subprogram it will not try to find context-dependent bounds even if more assertions on variable values apply in specific contexts.

Where?

Bounds on variable values can be asserted in all contexts: subprogram body, subprogram entry, loop, call, or global context. The variable in question can be a global variable, a subprogram parameter or a local variable. Note that an assertion on a global variable can be given for a non-global context, for example for a subprogram or a call, and then applies only in this context.

In fact, Bound-T does not really distinguish between global variables and local variables; it just maps the variable identifier to a memory location or a register and applies the assertion there. Global variables are usually statically allocated (static memory address) while local variables are often kept on the stack or in registers, but this distinction is not universal.

Globally

The simplest kind of variable-value assertion applies to a global variable in the global context. For example, assume that a data-logger program has a global variable *num_sensors* that shows from how many sensors it collects data. Here is how to assert that at most 15 sensors are active at any point in the program:

```
variable "num_sensors" <= 15;
```

This assertion should let Bound-T analyse and bound automatically any loops in the program that run from 1 to *num_sensors*, for example.

In a subprogram body

Continuing the above example, assume that the data-logger program has a subprogram *Initialize* that executes additional statements when there are no sensors, that is when *num_sensors* is zero. If you want to exclude this case from the analysis, here is how to assert that *num_sensors* is greater than zero within this subprogram:

```
subprogram "Initialize"
   variable "num_sensors" > 0;
end "Initialize";
```

If this assertion is given together with the earlier global assertion that *num_sensors* is at most 15, the global assertion applies in all subprograms, including *Initialize*, but within *Initialize* the local assertion also holds. Thus, within *Initialize* the *num_sensors* variable must be in the range 1 .. 15. This could also be asserted directly as follows:

```
subprogram "Initialize"
    variable "num_sensors" 1 .. 15;
end "Initialize";
```

Note that these assertions let *Initialize* change *num_sensors*, but they do claim that the value will never be greater than 15 or less than 1 within *Initialize*.

On subprogram entry

Perhaps you know the value that a variable has at the start of a subprogram, but not how the variable changes within the subprogram. You can make a variable-value assertion apply only on entry to the subprogram by writing the assertion in parentheses after the subprogram name, at the start of a subprogram block. Still continuing with the data-logger example introduced above, here is how to assert that *num_sensors* is less than 5 on entry to the subprogram *Initialize*:

```
subprogram "Initialize" (variable "num_sensors" < 5;)
end "Initialize";
```

Since the assertion applies only on entry to *Initialize*, it says nothing about how *Initialize* changes *num_sensors*. For example, *Initialize* can increase *num_sensors* to 11 without violating this assertion.

If this assertion is given together with the earlier global assertion that *num_sensors* is at most 15, the global assertion also holds on entry to *Initialize*, giving an upper bound of *min* (15, 4) = 4 for *num_sensors* on entry to *Initialize*.

In a loop

As an example of a variable-value assertion in a loop context, here is how to assert that the variable *N* is greater than 2 during any execution of the (only) loop in the subprogram *Fill_Buffer*:

```
subprogram "Fill_Buffer"
    loop
        variable "N" > 2;
    end loop:
end "Fill_Buffer";
```

This assertion does not constrain the value of *N* at any point outside the loop. The loop can change *N* as long as the new value is also greater than 2.

Note that the set of statements that belong to the loop are defined by the loop logic rather than by the syntax. For example, in the following Ada loop the statement that sets *N* to 1 is not within the logical loop because it is followed by an **exit** statement and so is not repeated:

```
for k in 1 .. num_sensors loop
    Sample_Sensor (k);
    if Done then
        N := 1;
        exit;
    end if;
end loop;
```

Thus, this loop conforms to the assertion that *N* is greater than 2 in the loop.

For calls

When the time or space usage of a subprogram depends on its parameters, or on some global variables that have different values in different calls, you may want to assert that these parameters or variables have certain values at a specific call or set of calls. You can do so by writing variable-value assertions in the context of this call or set of calls. Here is how to assert that the variable *N* equals 8 at any execution of any call to the subprogram *Clear* that occurs in the subprogram *Fill_Buffer*:

```
subprogram "Fill_Buffer"#
    all calls to "Clear"
        variable "N" 8;
    end calls;
end "Fill_Buffer";
```

Variable value bounds asserted in a call context apply in the caller, immediately before the execution flows from the caller to the entry point of the callee. They do not imply any constraints on variable values during the further execution of the callee.

Variable bounds asserted in a call context are used *only* for the context-dependent analysis of the callee for this call. Such assertions are thus useful only if Bound-T has *not* found context-independent bounds on the callee, because only in this case does Bound-T attempt context-dependent analysis of the callee. The presence of call-context assertions currently does not force a context-dependent analysis of the callee.

Global variables in calls

An assertion on the value of a global variable in a call context has the same effect as the same assertion in the entry context of the callee subprogram. Call-context assertions are however more flexible since you can use different values for different calls. Moreover, call-context assertions may imply bounds on the actual parameter values for this call as explained below.

Local variables in calls

When there are assertions on variable values in a call context, and some of these variables occur in the call's actual parameter expressions, the parameter-passing mechanism of the call translates the asserted bounds on the caller's variables into bounds on the callee's (formal) parameters. For example, consider an Ada call of the form

```
Send_Nulls (N => K + 1);
```

where *N* is a formal parameter to *Send_Nulls* and *K* is a local variable in the caller. Assume that the code for the caller keeps the caller's variable *K* in register r6, but the code for *Send_Nulls* expects the parameter *N* to be passed (by value) in register r0, and that we assert

```
call to "Send_Nulls" variable "K" 4; end call;
```

The result is to assert that r6 at the call has the value 4 and so r0, representing the parameter *N* of *Send_Nulls*, has the value 4 + 1 = 5 on entry to this invocation of *Send_Nulls*.

Do not assert foreign local variables in calls

Take care to assert call-specific bounds only on global variables or variables that are local to the *caller* or formal parameters of the *caller*. Assertions on the value of a variable that is local to other subprograms (such as the callee) will probably not work correctly, because Bound-T

translates the variable name to a machine-level local variable reference (such as a stack offset or a register reference). Bound-T then applies this machine-level reference in the caller, so that the assertion in fact bounds an unrelated local variable of the caller.

In particular, do not use the callee's formal parameter names in a call-context assertion. For example, assume that the formal parameter *N* of *Send_Nulls* (see the example in the preceding subsection) is passed via the stack and not in register r0 as assumed above. Now, although the above assertion on *K* for the call to *Send_Nulls* has the effect of bounding the formal parameter *N*, it *cannot* be written as follows:

```
call to "Send_Nulls" variable "N" 5; end call;   -- Wrong!
```

Since the symbol table maps *N* to “the first stacked parameter”, this (wrong) assertion in fact bounds the value of the first stacked parameter of the *caller*, which probably has nothing to do with *K* or *N*.

You can break this rule only if you are sure that the formal parameter is mapped to a statically addressed memory location or a statically named register so that the machine-level parameter reference points to the same physical storage location when interpreted in the caller and in the callee.

2.13 Assertions on variable invariance

Why?

When Bound-T analyses the computations in a subprogram or a loop it is often important to know if some part of the code, such as the loop body or a call, can change the value of a certain variable, or whether the variable is invariant (unchanged) over that code. Bound-T tries to detect invariant variables automatically but this analysis, like many others in Bound-T, is not complete and can miss some invariances. This can cause some other analysis to fail. For example, Bound-T may fail to find repetition bounds for a loop if it does not detect that the loop-counter variable is invariant over a call in the loop body. You can work around such problems by asserting the invariance of the variable.

However, using invariance assertions is difficult: it is not easy to understand when they can fix a problem and which invariances should be asserted. We aim to strengthen Bound-T's automatic invariance analysis to reduce the need for invariance assertions.

An invariance assertion can apply to a subprogram context, a loop context or a call context. We will discuss the subprogram context last because it is the strongest form.

Running example

Assume that *num_data* is a global integer variable and consider the C subprogram *Scan_Data* that has a loop that counts from 1 to *num_data* and calls *Check*:

```
void Scan_Data
{   int n;
    num_data = 100;
    for (n = 1; n <= num_data; n++) Check(n);
}
```

Assume further that *Check* has a conditional assignment to *num_data*. Since *Check* may change *num_data*, Bound-T cannot deduce that the loop in *Scan_Data* repeats 100 times. However, suppose that we know that the condition in *Check* is false in this context so that in fact *num_data* is unchanged. Below you will see different ways to assert this invariance and let Bound-T analyse the loop.

In a call

An invariance assertion for a call context means that this call does not change the variable in question, although other calls of the same subprogram may change it. Here is how to assert that *num_data* is invariant in the call from *Scan_Data* to *Check*:

```
subprogram "Scan_Data"
   call to "Check"
      invariant "num_data";
   end call;
end "Scan_Data";
```

In any call

An invariance that holds for all calls of a subprogram can be asserted in a global call context, without an enclosing subprogram block. Here is how to assert that no call of *Check* changes *num_data*:

```
all calls to "Check"
   invariant "num_data";
end calls;
```

In a loop

An invariance assertion with a loop context means that the variable retains its value in any repetition of the loop body. In other words, when execution enters the loop head with a certain value for this variable and goes through the loop body and back to the loop head, the variable has the same value again, even if it had different values in between.

For the above example with *Scan_Data*, *Check* and *num_data*, another way to assure Bound-T that *num_data* is invariant in the loop-counter code is this:

```
subprogram "Scan_Data"
   loop
      invariant "num_data";
   end loop;
end "Scan_Data";
```

Note that the final pass through the loop – the pass that ends the loop and does not return to the loop head – *can* change the variable. For example, *num_data* can be asserted as invariant in the following Ada loop, although its value on exit from the loop is different from its value on entry to the loop:

```
loop
   num_data := num_data + 1;
   exit when <some condition>;
```

```
    num_data := num_data – 1;
end loop;
```

In any loop

An invariance that holds for all loops can be asserted in a global loop context, without an enclosing subprogram block. Here is how to assert that *num_data* is invariant in any repetition of any loop that contains a call of *Check*:

```
all loops that call "Check"
    invariant "num_data";
end loops;
```

In a subprogram

An invariance assertion with a subprogram context means that the variable in question is invariant in *all* calls and *all* loops within this subprogram. The subprogram may contain assignments to the variable as long as the variable remains invariant in loop repetitions. Other subprograms called from this subprogram may change the variable temporarily as long as they restore its original value on return.

Here is how to assert that *num_data* is invariant in this sense within *Scan_Data*:

```
subprogram "Scan_Data"
    invariant "num_data";
end "Scan_Data";
```

This assertion implies those call-context and loop-context invariance assertions shown above as nested in subprogram blocks for *Scan_Data*. In fact, it implies the following:

```
subprogram "Scan_Data"
    all loops invariant "num_data"; end loops;
    all calls to "Check" invariant "num_data"; end calls;
end "Scan_Data";
```

It also implies the analogous "all calls" invariance for any call from *Scan_Data* to any subprogram, not just for calls of *Check*.

Note that the invariance in the subprogram context of *Scan_Data* does not conflict with the assignment of 100 to *num_data* in *Scan_Data*. Note also that it does *not* imply invariance over a call of *Scan_Data*. In fact, a call of *Scan_Data* probably changes *num_data* with this assignment.

2.14 Assertions on volatility

What does "volatility" mean?

Most memory locations are "persistent" in the sense that reading the memory location produces the value that was last written into the memory location. That is, the memory location really remembers what the program stored there. However, some memory locations do not behave in this way, and reading the location can return a different value. We call such locations *volatile*. Common examples of volatile memory locations include the following.

- Memory-mapped peripheral control registers. The value read from the register may bear no relation to the last value written, either because activity in the peripheral changes the register contents, or because writing to the register is not even meant to store a value for later reading, but to issue some command to the peripheral.
- Variables shared with other, concurrently executing threads. While the value read from the variable may equal the value last written, the last write may have been executed in another thread, and therefore the value may be different from the last value written in the thread that Bound-T is analysing.

Why assert volatility?

The data-flow and value-analyses in Bound-T by default assume that all memory locations (registers and variables) are non-volatile (unless there are some target-specific address ranges known to be volatile). These analyses can therefore track values from write instructions to read instructions, and so bound the values of computations and logical conditions. If some of these locations are in fact volatile, the analysis may be wrong, and may lead to incorrect (unsafe) "bounds" on execution-time and stack-usage.

For example, if the program uses the value of a volatile variable in a branch condition or loop repetition/termination condition, but Bound-T is not told that the variable is volatile, the incorrect value analysis may incorrectly conclude that the condition always has a certain value, and therefore wrongly consider that one path from the branch is infeasible, or that the loop terminates immediately, or never terminates.

When a memory location is known to be volatile, the analyses considers that reading the location returns an unknown value, which is not defined by the value last written to the location. This prevents the analysis errors described above.

For example, consider a processor with a built-in Analog-to-Digital Converter (ADC), also known as a digital voltmeter. Assume that the ADC has two memory-mapped 8-bit registers: a Command/Status register and a Value register, with the following functions:

- Writing to the Command/Status register sets the number of the input channel to be converted (0..15) and starts the conversion (measurement) process, which may last for several instruction times.
- Reading the Command/Status register returns the status of the ADC, where bit 0 (the least significant bit) shows if a conversion is going on (bit 0 = 0), or is completed (bit 0 = 1).
- Writing the Value register has no effect (and should be considered an error).
- Reading the Value register returns the result of the last (completed) conversion. It is an error to read the Value register when a conversion is going on.

The C statements to measure the voltage on channel 8, say, could look like this, where the variable "adc_cmd_stat" represents the Command/Status register, and "adc_value" represents the Value register:

```
adc_cmd_stat = 8;                 // Start conversion of channel 8.
while (adc_cmd_stat & 1) {};      // Loop (delay) until conversion completed.
result = adc_value;               // Read the conversion result.
```

If Bound-T analyses this code without knowing that adc_cmd_stat should be treated as a volatile memory location, it would propagate the value 8 assigned to adc_cmd_stat in the first statement to the use of adc_cmd_stat in the "while" condition. Since "8 & 1" is false (zero), the loop appears to terminate at once, which leads to an underestimation of the execution time.

The assertion language lets you assert variables as globally volatile, using the variable's name or address. Volatility assertions cannot (yet) apply in a non-global context. The assertion language also lets you assert entire address ranges as volatile, which means that Bound-T will consider as volatile any memory access that the analysis finds to lie in that address range. However, remember that Bound-T does not resolve all dynamic memory addresses.

Globally volatile variables

To make Bound-T analyse the above example correctly, you can mark "adc_cmd_stat" as volatile with this assertion, in the global context:

```
volatile "adc_cmd_stat";
```

If you also want to mark "adc_value" as volatile, you can either write two **volatile** assertions in the above form, or write both variable names in the same assertion, separated by a comma:

```
volatile "adc_cmd_stat", "adc_value";
```

Globally volatile address ranges

In some processors or systems, specific ranges of memory addresses are reserved for memory-mapped I/O registers. If, for example, the range 0x1000 ..0x 1FFF refers to I/O registers, you can make Bound-T consider as volatile any (identified) access to this range by writing this assertion in the global context:

```
volatile range "0x1000" .. "0x1FFF";
```

This assumes that memory addresses for the target processor in question can be written in the C hexadecimal form. The syntax for memory addresses is target-specific as described in the Application Note for the target. Note also that targets which have several separate address spaces may not let you write an address range that crosses over from one space to another.

Note that you cannot define an end-point of a **range** by a variable-name; you must use an address.

One and the same volatile assertion can list any number of variable names and/or memory address ranges, separated by commas.

2.15 Assertions on target-specific properties

Why?

For some target processors, the behaviour or timing of instructions depends on target-specific factors that Bound-T cannot analyse in general. For example, accessing certain memory locations may be delayed by “wait states” and the number of wait states may depend on the memory area or on processor configuration. The version of Bound-T for each target processor defines a set of such "properties" for the target (this set may be empty). Each property has a name and you can assert the value or range of values the property has in a certain context.

The available properties and their meanings are completely target-specific and are explained in the relevant Application Notes.

Bounds on properties can be asserted in all contexts except subprogram entry. However, properties for call contexts are currently not used (they have no effect).

Globally

Assuming that the current target processor has a property *read_ws*, perhaps expressing the number of wait-states necessary for reading memory, here is how to assert that Bound-T should assume the value 1 for this property globally:

```
property "read_ws" 1;
```

Inner context overrides outer context

Property assertions differ from variable-value assertions in that property assertions for inner (more local) contexts *override* assertions for outer (more global) contexts. For example, you can mix global context, subprogram context and loop context as follows:

```
property "read_ws" 1; -- Global context.

subprogram "Copy"
   property "read_ws" 2;  -- Subprogram context.
   loop
      property "read_ws" 3;  -- Loop context.
   end loop;
end "Copy";
```

The result is that Bound-T will use, for *read_ws*, the value 3 in the (single) loop in the *Copy* subprogram, the value 2 elsewhere within *Copy*, and the value 1 everywhere else.

2.16 Assertions on instruction roles

Why?

Some instructions can perform several *roles* in the execution of a program. For example, consider a "return" instruction that pops a return address off the stack and transfers control to that address. Most "return" instructions perform just that role, of returning control from a callee subprogram to the caller subprogram. However, a compiler can also use a "return"

instruction to perform some other kind of transfer of control. For example, a jump to a dynamically computed address can be performed by pushing the address value on the stack and executing a "return".

To analyze such multi-role instructions Bound-T must decide which role the instruction performs, because Bound-T uses different data structures and different analysis methods for different roles. The automatic algorithms built into Bound-T can sometimes choose the wrong role, which may make the analysis fail. In such cases you can tell Bound-T which role to use for that instruction.

An instruction that performs a tail call

For a concrete example, consider the way some compilers implement calls via function pointers on the Intel-8051 processor. This processor has a stack that is normally used to hold return addresses. The **ret** instruction pops a 16-bit address off the stack and into the Program Counter **PC**. Assume now that the 16-bit **DPTR** register holds the dynamically computed address of a subprogram (a function pointer, in C language terms). One way to call the subprogram at this address is to call an intermediate subprogram that we name ICall which contains instructions that push the **DPTR** on the stack, followed by a **ret**. In 8051 assembly language:

```
ICall:  push DPL
        push DPH
        ret
```

Note that although the **ret** instruction occurs in its usual place at the end of a subprogram (ICall), its role is not just to return from this subprogram. What it does is to transfer control to the address in **DPTR**, the entry point of the subprogram we want to call. However, the return address for ICall is still on the stack, so when the subprogram at **DPTR** eventually returns, it returns to that address (the instruction following the call of ICall). This kind of "trampoline" call is often termed a tail call, and here is how you can assert to the Intel-8051 version of Bound-T that this **ret** instruction performs a tail call, not an ordinary return:

```
subprogram "ICall"
   instruction at offset "4"
      performs a "tail call";
   end instruction;
end "ICall";
```

Note that the name of the instruction role is written in quotes as "tail call". This is because the instruction roles are target-specific and are not standard keywords of the assertion language.

To find out which offset to use to identify an instruction you can disassemble the subprogram (for example with the Bound-T option *-trace decode*) and compute the difference of the instruction address and the subprogram entry point address. In the example, the offset is 4 octets because each of the **push** instructions is 2 octets long.

The roles that instructions can perform

The instructions and their roles are of course target-specific. Moreover, most instructions have fixed roles that cannot be changed with role assertions. For example, an instruction that just adds two data registers and puts the sum in a third data register can hardly be required to perform a call. The role-instruction combinations for a given target processor are listed in the Bound-T Application Note for that target. If you assert a role for an instruction that Bound-T is not willing to model for that instruction, the result will be an error message to this effect, or a warning message saying that the assertion was not used at all.

2.17 Special assertions on subprograms

Whether the subprogram returns

Some subprograms never return to the caller; the best known example is the *exit* function in C. Knowing that a subprogram never returns can simplify the analysis of other subprograms that call the non-returning subprogram. Here is how to assert that *exit* never returns:

```
subprogram "exit"
    no return;
end "exit";
```

Whether the subprogram returns to an offset address

Some subprograms do not return to the return address offered by the caller, but to some later (or perhaps earlier) point in the code. This is often the case for special library subprograms that access constant data, stored in the program code immediately after the call instruction.

Consider, for example, a library for 64-bit floating-point arithmetic on an 8-bit processor. Floating-point computations often use constants, in this case 64-bit constants. A convenient way to encode such an 8-octet constant in the program is to put it in the code and precede it by a call to a helper subprogram, with the name fcload, say. This subprogram takes its own return address (from the stack, for example); uses it as a pointer to read the 8-octet floating point constant from the code; does whatever is necessary with the floating-point value (for example, writes it to some working registers of the floating-point library); and increments the return address by 8, thus returning to the instruction *after* the 8-octet constant. For example, in the Intel-8051 assembly language (as used in the SDCC compiler), a call to fcload could look like this:

```
lcall fcload
.byte 0x22      ; This is the offered (normal) return point,
.byte 0xA1      ; but for "fcload" we have instead 8 constant
.byte 0x52      ; data octets.
.byte 0x10
.byte 0x12
.byte 0x31
.byte 0x00
.byte 0xFF      ; This is the last (8th) data octet.
; The call of "fcload" actually returns to the instruction
; following this comment.
```

When Bound-T is constructing the flow-graph of some subprogram that calls fcload, it needs to know the return point of the call, even before analysing fcload itself. In the absence of assertions Bound-T assumes that fcload follows the normal calling protocol, which usually means that the call returns to the point immediately after the call instruction. However, for fcload that point contains the 8-octet constant, not instructions, so Bound-T would try to decode the constant as executable instructions, which would be quite wrong and could lead to too large or too small time and space bounds. For example, if the first octet of the constant happens to be the machine code for a "return" instruction Bound-T will assume that the calling subprogram ends after the call of fcload, which could omit much of the code of this subprogram from the analysis. (This is the case in the Intel-8051 example above, because an 8051 return instruction is one octet with the value 22 hexadecimal, which also happens to be the value of the first octet of the constant.)

The following assertion tells Bound-T that the true point of return from calls to fcload is the offered return address (immediately after the call) plus 8 octets (to skip the 8-octet constant):

```
subprogram "fcload"
    returns to offset "8";
end "fcload";
```

The precise meaning of the offset string (here "8") is target-specific. The example above assumes that offsets are measured in octet units. Some target processors may use larger units such as 16-bit or longer words.

Whether to use arithmetic analysis

The Presburger-arithmetic analysis that Bound-T uses to find loop-bounds and other facts can be quite expensive in time and space. There is a command-line option (*-arithmetic*) to enable or disable this analysis globally for all analysed subprograms, but it is sometimes useful to enable or disable it for individual subprograms. Therefore, the assertion language lets you override this command-line option. Here is how to enable arithmetic analysis for the subprogram *Involutor*:

```
subprogram "Involutor"
    arithmetic;
end "Involutor";
```

And here is how to disable it:

```
subprogram "Involutor"
    no arithmetic;
end "Involutor";
```

Whether to integrate the callee into the caller's analysis

In special cases it may be useful to tell Bound-T not to analyse a subprogram separately, but as a part of the code of every caller, as if the called subprogram were “inlined” in the caller. Such *integrated* analysis may be necessary for subprograms that do not follow the normal calling conventions, for example library routines that the compiler invokes as part of the “prelude” or “postlude” code to set up or tear down local stack frames.

The following assertion shows how to specify integrated analysis for the subprogram *C$setup*:

```
subprogram "C$setup"
    integrate;
end "C$setup";
```

Bound-T may default to use integrated analysis for some predefined routines under some target processors and target compilers; if so, it will be explained in the relevant Application Notes. Such a default cannot be disabled by an assertion.

An integrated subprogram does not, in fact, appear as a “subprogram object” in Bound-T's model of the program structure. Thus it is not useful to assert anything else for such a subprogram. Moreover, since the subprogram is not analysed on its own, Bound-T does not report any analysis results such as a WCET bound or stack usage bound for the subprogram. Instead Bound-T includes the subprogram's execution time and stack usage in the results for the calling subprograms.

Likewise, a call to an integrated subprogram does not appear as a “call object” in Bound-T's model of the program structure. Thus, it is not possible to assert anything for such a call, nor to use the existence of the call as a property that identifies a containing loop, for example.

Whether the subprogram is used at all

A program is often analysed under certain assumptions that define (limit) the scenarios to be included in the analysis. For example, one often wants an analysis of the “nominal” scenarios in which no run-time errors happen. One aspect of such scenarios may be that they never use (call) certain subprograms, for example error-handling subprograms. Bound-T provides a dedicated form of assertion, as in this example that states that subprogram “Show_Error” is never used:

```
subprogram “Show_Error”
   unused;
end “Show_Error”;
```

An **unused** assertion has two effects: firstly, Bound-T considers all calls to this subprogram to be infeasible (never executed); secondly and consequently, Bound-T does not analyse this subprogram. The analysis results would be irrelevant.

The keyword **unused** can also be written as **not used**.

Whether to omit the subprogram from the analysis

It may be impossible or undesirable to analyse some subprograms at all, even if the subprogram is used by the target program. It is simple to tell Bound-T to omit a certain subprogram from its analysis, as in this example for the subprogram “Switch_Task”:

```
subprogram “Switch_Task”
   omit;
end “Switch_Task”;
```

However, such an assertion does *not* omit calls to the omitted subprogram, so the calls (and callers) will be *unbounded* (have unknown execution-time and stack-usage bounds) unless you also assert bounds on time and/or stack usage for the subprogram or for each call.

If you assert sufficient time bounds, stack usage bounds, and (for unstable stacks) final stack height bounds for a subprogram, Bound-T automatically omits the subprogram from the analysis. In this case an **omit** assertion is allowed but is redundant.

The combination of **unused** and **omit** is redundant since **unused** implies **omit**. But note that **omit** does not imply **unused** because the calls to an omitted subprogram remain in the analysis.

Whether other assertions are enough for time analysis

Subprograms with irreducible flow-graphs are problematic for Bound-T because, for such subprograms, Bound-T can neither find loop repetition bounds automatically nor accept assertions on loop repetition bounds – because Bound-T is unable to structure the flow-graph into loops and non-loop parts. However, it is still possible to assert repetition bounds on some parts of such subprograms – calls or instructions – and these assertions may be strong enough to constrain the number of repetitions of all parts of the flow-graph. Consider, for example, a subprogram *Irred* with the flow-graph shown in the figure below.

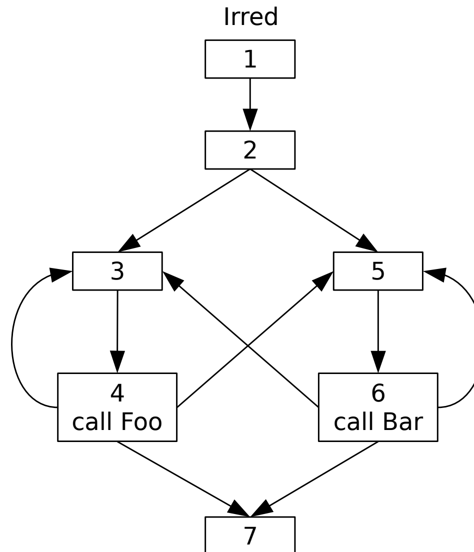

Figure 6: An irreducible flow-graph

The nodes labeled 3, 4, 5, and 6 can be repeated, but they cannot be collected into a “natural” loop with a single entry point (the loop head). Thus, the flow-graph is irreducible. However, all repeated execution paths must pass through nodes 4 or 6, and both nodes (we assume) contain calls, as indicated in the figure. Assertions that bound the number of times these calls are repeated are enough to bound the number of repetitions of all nodes. However, you must explicitly tell Bound-T that this is so, as in the example below:

```
subprogram “Irred”
   call to “Foo” repeats 15 times; end call;
   call to “Bar” repeats 27 times; end call;
   enough for time;
end “Irred”;
```

The assertion **enough for time** makes Bound-T try to find an execution-time bound (using the IPET method) even though the flow-graph is irreducible, and in this example it will succeed because the other two assertions, on the calls, are strong enough.

Whether to show the subprogram in the call-graph drawing

Bound-T can help you understand your program by drawing the call-graph as explained in the Reference Manual. However, sometimes the call-graph is cluttered because some utility subprograms are called from many places. For example, on some processors multiplication or division are implemented by library subprograms so the drawing may have a multitude of call-arcs to these subprograms. You can make the call-graph clearer by asserting that certain subprograms should be hidden (omitted). For example, the following hides the subprogram *m$divi*:

```
subprogram “m$divi”
   hide;
end “m$divi”;
```

The hiding assertion does not apply “recursively”: if *m$divi* calls some other subprograms these are not automatically hidden but will appear in the call-graph drawing unless you assert that they should be hidden too.

Note that **hide** has no effect on the actual analysis, only on the call-graph drawings. Indeed **hide** is independent from the **unused** and **omit** assertions.

3 IDENTIFYING PROGRAM PARTS

3.1 Why and how: the different ways

When you write an assertion you must identify its *context*, which means the part of the program under analysis to which the assertion applies. For assertions on variable values you must also identify the variable or storage cell to be constrained. The Bound-T assertion language provides three ways to identify a program part or entity:

- by its *name*, for example the name of a subprogram or variable,
- by its *relationship* to other parts, for example that it is contained in a known subprogram,
- by its *position in the source code*, given by a source-file name and source-line number,
- by its *position in the machine code*, given by a machine address for code or data.

Some of these identification methods cannot be used for some kinds of program entities, as currently implemented in Bound-T. Table 1 below shows the allowed combinations. Table cells shaded grey indicate the unavailable methods. Some entities have obligatory identification methods: a static call must always be identified by its callee (by name or address), and an instruction by its address; the other methods can be added to constrain the assertion to apply only to some calls with this callee, or some instances of the instruction at this address.

Table 1: Allowed means of identification of program parts

Entity	By name of	By relationship to	By source line # of	By address of
subprogram	the subprogram			the entry point
loop	the label of a statement in the loop	the containing subprog-ram, or a call in the loop, or an access to a variable in the loop, or an inner or outer loop	the loop head, or any code in the loop, or any line in the range of lines spanned by the loop	any instruction in the loop
call, static	the callee	the containing subprogram or loop	the call instruction	the callee
call, dynamic		the containing subprogram or loop	the call instruction	
instruction		the containing subprogram		the instruction
variable or other cell	the variable	a scope (a block of local variable declarations)		the register or memory location

Section 3.2 below describes, in a general way, how program entities are identified by their names, possibly qualified by scope information. Section 3.3 describes, again generally, how entities can be identified by their source-code position. The later sections in this chapter discuss each kind of entity (subprogram, loop, ...) and how such entities can be identified using any allowed method.

3.2 Names, scopes, and qualified names

Scopes qualify names

Assertions can refer to program entities by names (identifiers, symbols). A name can be a subprogram name, a variable name or the name of a statement label. It is common to use the same basic name for many different variables in a program, for example, many loop counters may be called *i* or *count*. Sometimes the same basic name is used for different subprograms, for example in different modules. Bound-T tries to separate such synonyms by adding *scopes* to the names.

Scopes are nested hierarchically. The scope levels that are used depend to some extent on the target processor and the target compiler and linker, but typically the top level identifies the module (source-code or object-code file) and the next level (if any) identifies the subprogram that contains the entity in question. The scope system is explained in the relevant Application Notes.

The "fully qualified" name of an entity consists of the scope names followed by the basic name, all enclosed in quotes and separated by a delimiter character that is usually the vertical bar '|'. For example, the local variable *i* defined in the subprogram *fill_buffer* defined in the module (file) *buffering* would have the fully qualified name "buffering|fill_buffer|i". If the *buffering* module contains another subprogram *initialize* that has its own local variable *i*, this would be "buffering|initialize|i". If another module *sink* contains another subprogram *initialize* that has its own local variable *i*, this would be "sink|initialize|i".

Unique suffix suffices

You can always use the fully qualified name to identify a subprogram or a variable, but it is enough to give those scope levels (starting from the bottom) that make the name unambiguous.

In the examples above, the variable name "i" is clearly ambiguous. The partially qualified name "initialize|i" is also ambiguous because it occurs in two modules, *buffering* and *sink*, so you must use the fully qualified names "buffering|initialize|i" and "sink|initialize|i" to refer to these two *i* variables.

The partially qualified variable name "fill_buffer|i" is enough to identify the *i* in *fill_buffer* because (in this example) there is only one subprogram called *fill_buffer*.

The unqualified subprogram name "fill_buffer" is also unambiguous for the same reason. The two *initialize* subprograms have to be qualified as "buffering|initialize" and "sink|initialize" respectively.

Default scope

The assertion language provides the keyword **within** to let you set a *default scope* that is prefixed to all names. Continuing on the examples above, after the default scope definition

```
within "buffering|initialize";
```

you can write just "i" instead of "buffering|initialize|i".

When a default scope is defined it applies to all name strings that start with a normal character. Here the name "fill_buffer|i" would be interpreted as "buffering|initialize|fill_buffer|i" which would probably not be a valid name. To ignore (escape) the current default scope, put the delimiter character at the start of the name, as in "|fill_buffer|i".

The default scope can be cleared by defining a null string as the default scope: **within** "".

Different delimiters

Some target compilers may use the vertical bar character '|' *within* names which means that it cannot be used to delimit scope levels. The assertion language provides the keyword **delimiter** for changing the scope delimiter, for example to a diagonal slash as follows:

```
delimiter '/';
```

Afterwards you would write for example "fill_buffer/i" to refer to the variable *i* in the subprogram *fill_buffer*.

3.3 Source-code positions

The source-to-object mapping

Bound-T itself never reads the source-code files of the program under analysis, but the symbol table (debugging information) embedded in the executable program file usually contains a compiler-generated line-number mapping that connects the machine addresses of instructions to the source-code lines that gave rise to those instructions. This gives Bound-T some knowledge of the source-code lines that correspond to specific instructions, for example all the instructions in a loop, or the call instruction that transfers control from a calling subprogram to a callee subprogram. Bound-T can use this mapping in reverse to identify the loop or call that corresponds to given source-code lines.

However, the mapping between source-code and machine-code positions is often incomplete and inexact, simply because the relationship is complicated by the extensive transformations that powerful compilers apply to the code – optimizations, reorderings, code sharing, and so on.

Source-line numbers versus marks

There are two ways to specify the source-code position of a program part:

- directly, by the *actual number* of the source-code line and (perhaps) the name of the source-code file, or
- indirectly, by the *name of a marker* embedded in the source code at this place.

The direct method can be used without any alteration or annotation of the source code. However, if the source code is then changed, the line numbers may also change, which may force you to update the assertions.

The indirect, marker-based method requires adding marks to the source code. Marks are usually written as comments in a special form. As comments, they should have no effect on the generated code. However, you need a program of some sort to scan the source-code files, recognize the marks, and create a *mark-definition file* that Bound-T can use. Tidorum provides one such program called *find_marks*.

The same marker name can mark several places in the same source file or in several files. An assertion using this marker name applies to all those places, for example to all calls marked by this marker name. This cannot be done easily by direct use of source-code line numbers.

Example

For example, consider the following C function *add_up*, where line numbers are shown on the left and only the start of the function is shown in detail.

```
 1    int add_up (int A[], int n)
 2    {
 3       int sum = 0, i;
 4       for (i = 0; i < n; i++)
 5       {
 6          sum += A[i];
 7          A[i] = 0;
 8       }
 9       if (sum < 0)
10          normalize (
11             &sum,
12             n/2);
...      ...
31    }
```

The function contains a loop, *for* (*i* ...), and a call to the function *normalize*. We now consider how these parts of *add_up* can be identified using their source-code locations.

Identifying the loop by its source-line numbers

The loop in the example above stretches over five source lines (numbers 4 .. 8), but it is unlikely that all these lines will be represented in the source-to-object mapping. For example, the source-code lines 5 and 8 hold only curly braces and do not directly cause any code to be generated. On the other hand, some compilers may map the last code generated in the loop to line 8, simply because that line indicates the end of the loop and in this sense "makes" the compiler generate the branch back to the start of the loop and perhaps some of the code that checks loop termination. Lines 4, 6, and 7 should normally give rise to machine code and be mapped to that code. This means that the loop is likely to be identifiable with these lines.

Using the line at the loop head (line 4), you can write a loop-bound assertion as:

```
subprogram "add_up"
   loop on line 4 repeats 15 times; end loop;
end "add_up";
```

That works if the compiler maps line 4 to some of the loop initialization code (for example, the assignment $i = 0$) or to the loop head (for example, the comparison $i < n$). However, there is no guarantee that the compiler does that.

Using a line within the loop body, for example line 6, the same assertion can be written as:

```
subprogram "add_up"
   loop containing line 6 repeats 15 times; end loop;
end "add_up";
```

That works if the compiler maps line 6 to some code in the loop body (or in the loop head, which we consider part of the loop body). However, there is no guarantee that the compiler does that.

Perhaps the most robust way is to use the "spanning" form:

```
subprogram “add_up”
    loop spanning line 6 repeats 15 times; end loop;
end “add_up”;
```

This works as long as the range of line-numbers that the compiler maps to some code in the loop contains the line-number 6. It is not necessary for line 6, itself, to be mapped to some loop code.

Identifying the call by its source-line number

In the example above, the call to *normalize* stretches over three lines (numbers 10 .. 12). However, from Bound-T's point of view the “call” in the program under analysis consists of only one instruction, the last instruction executed in *add_up* before control passes to *normalize*. Different compilers may choose to map this instruction to line 10, because that is where the call starts, or to line 12, because that is where the call statement is complete and code can be generated for the call instruction. A compiler is unlikely to map the call to line 11.

For a mapping to line 10, you can write an execution-time assertion for this call as follows:

```
subprogram “add_up”
    call to “normalize” at line 10
        time 213 cycles;
    end call;
end “add_up”;
```

In case the compiler maps the call to line 12, you have to change the assertion to say “at line 12”. The fact that the mapping is compiler-dependent is annoying and can be cumbersome.

Specifying the “fuzz” for comparing line numbers

The compiler-specific variation in the source-line numbers mapped to a loop or call can sometimes be overcome by telling Bound-T to tolerate a small difference or “fuzz” between the line number written in the assertion, and the line number actually connected to the loop or call. Bound-T has a default fuzz that is controlled with the command-line option *-line_fuzz*. The fuzz can also be defined for each assertion, overriding the default fuzz for that assertion. This is done by the keyword **within** followed by the allowed difference range, as in this example:

```
subprogram “add_up”
    call to “normalize” at line 10 within 0 .. 3
        time 213 cycles;
    end call;
end “add_up”;
```

This assertion matches any call to *normalize* that is connected to a source line with a number in the range 10 .. 13, that is, the nominal line number 10 plus the interval 0 .. 3.

The risk of using a large fuzz is of course that the assertion may mistakenly match also some other, nearby call or loop that falls within the fuzzy range.

Using line-number offsets

If you write assertions that use source-line numbers, and then change the target program by adding or removing lines from those source files, you must update the assertions to use the changed line-numbers, which is obviously inconvenient. One way to avoid this inconvenience is to use source-code marks, as described below; another is to use line-number offsets instead of absolute line numbers.

Line-number offsets can be used only in assertions within subprogram blocks, because the offset is defined relative to the *first line* of the subprogram. The first line of a subprogram is the source-line number that the compiler maps to the machine address of the subprogram's entry point (the first instruction in the subprogram). Unfortunately, the number of this first line depends on the compiler; some compilers use the line that contains the first text of the subprogram profile – this would be line 1 in the source-code for *add_up*, above – while others use a later line that begins the subprogram's body, for example (in C code) the line with the opening '{', which is line 2 in *add_up*, or the line that holds the first executable statement.

Assuming that the compiler uses line 2 as the first line for *add_up*, the assertion on the loop in *add_up* can be written as follows with a line-number offset:

```
subprogram “add_up”
   loop spanning line offset 4 repeats 15 times; end loop;
end “add_up”;
```

The position **line offset** 4 is the same as **line** 6, because 6 = 2 (first line) + 4 (offset).

Identifying the loop by a mark

To illustrate the use of marks we must add mark lines to the source-code of *add_up*. The form (syntax) of source-code marks is defined by the tool that you use to find the marks in the source code. In this example, we use the syntax defined by the *find_marks* program from Tidorum, in which a mark in C code is written as a C comment starting with the string /**Mark, followed by some optional keywords describing the marked place, and then by the marker names which are usually enclosed in quotes. Here is the source-code of *add_up* with two marks, one for the loop and one for the call:

```
 1    int add_up (int A[], int n)
 2    {
 3       int sum = 0, i;
 4       for (i = 0; i < n; i++)
 5       {
 6          sum += A[i];
 7          /**Mark “summer” */
 8          A[i] = 0;
 9       }
10       if (sum < 0)
11          /**Mark call below “norm” */
12          normalize (
13             &sum,
14             n/2);
...      ...
31    }
```

When *find_marks* processes this source code it associates the marker name “summer” with the line-number 7 and the marker name “norm” with the line-number 12. You can then identify the loop by the marker “summer”:

```
subprogram "add_up"
   loop spanning marker "summer" repeats 15 times; end loop;
end "add_up";
```

The meaning of **spanning marker** "summer" is the same as that of **spanning line** 7.

Similarly, the call can be identified by "norm", perhaps also using **within** to define a fuzz:

```
subprogram "add_up"
   call to "normalize" at marker "norm" within 0 .. 3
      time 213 cycles;
   end call;
end "add_up";
```

How to write marks for the *find_marks* program is described in more detail in the User Manual for *find_marks*, *http://www.bound-t.com/manuals/find-marks-manual.pdf*.

Naming the source-code file

When you identify a program part by its source-code position, whether directly by a source-line number or indirectly by a marker name, you can also specify the name of the source-code file that contains this source line or mark. For assertions within a subprogram block you can usually omit the source-file name because the assertion context is then implicity limited to parts within this subprogram, which usually lie in exactly one source-code file. However, for assertions in global scope it may be necessary and useful to specify the source-file name.

For example, assume that the target program has a subprogram *output* that can emit various kinds of messages, some of which are classified as error messages, and you want to assert that no error messages can be generated in the code from the source file inversions.c. You can assert this as follows, assuming that the error-message-emitting *output* calls are marked with the marker "error-message" to separate them from other calls of *output*:

```
all calls to "output"
   at marker "error-message" in file "inversions.c"
   repeat 0 times;
end calls;
```

This assertion does not apply to any calls of *output* outside the source file inversions.c, whether or not those calls are marked "error-message". If the same assertion is used without the part **in file** "inversions.c" it applies to all *output* calls marked "error-message" in any source-code file.

This completes the explanation of source-code positions and how they can be used to identify program parts. The rest of this chapter describes how to identify each kind of program entity – subprograms, variables, loops, calls, instructions – by any method, but the focus is on methods other than the source-code position.

3.4 Identifying subprograms

When writing assertions you may need to identify a subprogram in four places:

- To define a subprogram context: subprogram "Foo" .. end "Foo";
- To define a call context: call to "Foo" .. end call;
- To characterize a loop by a call: loop that calls "Foo";

- As a callee of a dynamic call: dynamic call calls “Foo”.

In all places you can either use the name or the entry address of the subprogram. Source-code positions cannot be used.

By symbolic name

Subprograms are usually identified by writing the subprogram name in quotes: ”Foo”. If the name is ambiguous (occurs in several modules, for example) it has to be qualified by a sufficient number of scope levels: ”database|Foo”.

You must use the subprogram's *link-name*, that is, the name that the linker uses for this subprogram. In some target environments the link-name equals the source-code name (the identifier). In other environments the name is slightly modified, for example by prefixing an underscore so that the source-code name *Foo* becomes the link-name ”_Foo”. The Application Notes for the target will explain this modification, if any. You can find out the link-names assigned by the compiler and linker by dumping the target program with some dumping tool such as the Unix tools *nm* or *objdump*, or by dumping the target program with Bound-T as explained in the Reference Manual, or by running Bound-T with the option *-trace symbols*. The last method also shows the scope that Bound-T assigns to each symbol.

By machine address

Subprograms can also be identified by their machine-level entry-addresses, in the form

```
subprogram address "12345"
```

The form and meaning of the quoted string following the **address** keyword are in principle target-dependent and explained in the Application Notes. The string is usually a hexadecimal number giving the entry address. Of course this is a last-resort method, to be used only if the function has no symbolic identifier.

By an offset added to a symbolic name or machine address

A subprogram can also be identified by an entry address that is *offset* by a constant number from a *base* address. The base address can be given as a numerical address or as a symbolic name. For example, the following identifies a subprogram that is offset by "34" address units from the subprogram "Foo":

```
subprogram "Foo" offset "34"
```

If the entry address of "Foo" is 1A36 in hexadecimal digits, and if the offset "34" is also interpreted as a hexadecimal number (the interpretation of offsets is target-specific), the above line identifies a subprogram with the entry address 1A36 + 34 = 1A6A hexadecimal. The same subprogram could be identified using either of the two following forms:

```
subprogram address "1A6A"
```

or

```
subprogram address "1A36" offset "34"
```

Identifying a subprogram by an offset relative to another, named subprogram is useful when a library contains local subprograms that have no public, symbolic name. In such cases the offsets between subprograms in the library are often constant, although the absolute addresses depend on the memory map of the program to which the library is linked.

3.5 Identifying variables

When writing assertions you may need to identify variables in three places:

- To assert bounds on the value: variable ”count” <= 15;
- To characterize a loop by a variable it uses or defines: loop that uses “count”;
- To assert invariance: invariant “count”.

In all places you can either use the name or the machine address of the variable. Source-code positions cannot be used.

By symbolic name

Most compiler tool-chains generate symbolic information giving the names and addresses of all global variables, even for an optimised executable. Thus, global variables can be named and tracked without problems. The same holds for formal and actual parameters.

Many examples of variable naming appeared earlier in this chapter.

In the current version of Bound-T it is not possible to name record/structure components (members) or array components. Only stand-alone variables can be named symbolically.

Symbolic information on local variables is sometimes not provided in an optimised executable. Moreover, it seems likely that optimisation can have drastic effects on the set of local variables, such as placing them in registers, perhaps even in different registers for different instructions. The Application Notes should detail how local variables can be named with specific target processors and target compilers.

You can find out the symbols that are available in the target program by dumping the target program as explained in section 3.4.

By machine address

Variables can also be identified by their machine-level addresses, in the form

```
variable address "12345"
```

The form and meaning of the quoted string following the address keyword are in principle target-dependent, just as for subprogram addresses discussed above. It will usually be a hexadecimal number giving the memory address, but targets may also make processor registers accessible in this way. For example, the register called r3 in assembly language might be named as follows in an assertion:

```
variable address "r3"
```

The syntax for register names is explained in the relevant Application Note.

Careful with the scope

Please note that Bound-T translates the variable name, as written in the assertion, to an internal low-level data reference, such as a memory address, or a register name, or a stack offset relative to the current call-frame pointer. Bound-T does not memorize which high-level scope was used in this translation. Confusion can result if these scopes are mixed up.

For example, assume that subprogram *Foo* has a loop that uses a local variable *x* which the compiler has placed in register r3, and subprogram *Eek* has a loop that uses its local variable *y* which the compiler has also assigned to register r3.

Under these assumptions, the global assertion

```
all loops that use "Foo|x" repeat 5 times; end loop;
```

is translated into an internal form that corresponds to

```
all loops that use address "r3" repeat 5 times; end loop;
```

This loop description will match the loop in *Foo* but also the loop in *Eek*, and probably will also match loops in a great number of subprograms that have nothing to do with either *Foo* or *x* but use r3 for their own local purposes. So be careful when describing loops or calls by means of local variables.

3.6 Identifying loops

When writing assertions you may have to identify specific loops for the following reasons:

- To define a loop context.
- To help identify a call by identifying the loop that contains the call.
- To help identify another loop by identifying an inner or outer loop.

Unlike subprogram and variables, loops seldom have names and thus we identify loops indirectly through the properties or characteristics of the loop, or through the source-code position of the loop.

Loop properties

A loop can be identified firstly by the subprogram that contains the loop and secondly by specific properties of the loop itself, including its source-code position.

Writing loop assertions within a subprogram block specifies that the loop(s) to be identified lie in this subprogram. Writing loop assertions in the global context specifies that the loop(s) to be identified can lie in any subprogram.

Section 3.3 already showed how to use source-code positions to identify loops. In addition, the following specific properties or keywords can be used to identify loops:

labelled	The loop contains (or does not contain) a specific statement label.
calls	The loop calls (or does not call) a specific subprogram.
uses	The loop reads (or does not read) a specific variable.
defines	The loop assigns (or does not assign) to a specific variable.
in	The loop is contained (nested) in another loop (or is not so contained).
contains	The loop contains (or does not contain) another loop.

executes The loop contains (or does not contain) the instruction at a given machine address. This property is meant as a last resort and is obviously not robust against changes in the target program, recompilation with different compiler options, or even relinking with a different memory lay-out.

The properties **contains** and **in** make this identification scheme recursive in the sense that the properties of an outer loop can be used to identify the inner loop, or vice versa.

Single loops or sets of loops are thus identified by listing some of their properties. Examples follow. The examples mainly show loop repetition assertions but of course the same loop identifications can be used to assert other kinds of facts, such as bounds on variable values within the loop.

A silly example: all loops in the program

There is probably no target program where this would be useful, but just as an example here is how to assert that *every* loop in the target program repeats 7 times. Write this in a global context (not within a subprogram block):

```
all loops repeat 7 times; end loops;
```

The only loop in a subprogram

When there is only one loop in the subprogram under analysis, the loop can be identified simply by writing the loop block within the subprogram block. It is not necessary to add specific loop properties. For example, here is an assertion that the single loop in subprogram *Stop_Motor* repeats 11 times:

```
subprogram "Stop_Motor"
    loop
        repeats 11 times;
    end loop;
end "Stop_Motor";
```

All loops in a subprogram

Another case where no loop properties need be given is when the same assertion applies to all loops in the subprogram in question. The keyword **all** is then placed before **loop**, as in this example that asserts that all loops in the subprogram *Print_Names* repeat 25 times:

```
subprogram "Print_Names"
    all loops repeat 25 times; end loops;
end "Print_Names";
```

The loop that calls

When there are several loops in the subprogram that must be distinguished in the assertions, one or more properties are needed. For example, here is the loop that calls subprogram *Foo*:

```
loop that calls "Foo"
```

Assuming that this loop is in the subprogram *Master*, here is a complete assertion that this loop repeats up to 9 times:

```
subprogram "Master"
    loop that calls "Foo"
        repeats <= 9 times;
    end loop;
end "Master";
```

In a *Calls* property, the call is identified only by naming the callee subprogram. It is not currently possible to identify the call using the other call-properties explained in section 3.7.

The loop that accesses

The variables that a loop accesses (reads or writes) can be used as properties of the loop. However, only statically accessed integer variables can be used here. Floating-point variables cannot be used because Bound-T generally does not model floating-point computations. Arrays (indexed variables) or variables accessed via pointers cannot be used because the accessed memory location is not statically known.

As an example, here is a C subprogram *Subtract_Average* that subtracts the average value of one integer vector from another integer vector:

```
void Subtract_Average (int input[], int output[])
/* Subtracts the average of input[] from output[]. */
/* Both vectors are terminated by zero elements.   */
{   int i; int sum = 0; int avg;
    for (i = 0; input[i] != 0; i++) sum += input[i];
    avg = sum/i;
    for (i = 0; output[i] != 0; i++) output[i] -= avg;
}
```

Here are some assertions that set a bound of 40 repetitions for the first loop that computes the *sum* and 120 repetitions for the second loop that modifies *output*:

```
subprogram "Subtract_Average"
    loop that defines "sum" repeats 40 times; end loop;
    loop that uses "avg" repeats 120 times; end loop;
end "Subtract_Average";
```

Note that

- the counter variable *i* cannot be used to separate the loops because both loops use *i* in the same way (reading and writing), and
- the array variables *input* and *output* cannot be used to separate the loops because the loops access their elements using dynamic (indexed) addressing.

The example identifies the first loop with the property `defines "sum"`. Based on the source code the property `uses "sum"` should work, too, and indeed it may work. However, Bound-T inspects the machine-code form of the loops. In this example an optimizing compiler may well assign the *same* storage location (perhaps a register) to *both* the variables *sum* and *avg*. Both loops would then read this storage location so the `uses` property would apply to both loops. Using `defines` for the first loop is more robust.

You may wonder how we can use the local variables *sum* and *avg* in these properties when they are allocated on the stack and so do not have static addresses. This works because such variables are usually accessed with static offsets relative to the stack pointer. Bound-T analyses such accesses as using or defining statically identified (local) variables.

Labelled loop

Despite the general acceptance of “structured” coding styles loops are still sometimes built from **goto** statements and statement labels, for example as in this C code:

```
void search (void)
{
    start_over:
        ... some code ...
        if (!done) goto start_over;
}
```

Assuming that the compiler and linker place the statement label *start_over* in the symbol-table, the loop can be identified by the label, for example as follows:

```
subprogram "search"
    loop that is labelled "start_over"
        repeats 10 times;
    end loop;
end "search";
```

The same holds for a loop that is written in a structured way with **for** or **while** but still contains a statement label for some reason. The label can be placed anywhere in the loop; it does not have to be at the start.

Last chance: the loop that executes "address"

If there is no better way, you can identify a loop by stating the machine address of an instruction in the loop. Any instruction in the loop will do; you do not need to pick the first or last one. This description of the loop is very fragile because any change to the program or to the libraries it uses is likely to move the loop to a different place in memory which means that the address in the assertion may also have to be changed. However, the address can optionally be given as an offset from the start of the containing subprogram, a slightly more robust definition.

The address or offset is written in a target-specific form, but usually it is simply a hexadecimal number. For example, here is the loop that executes (contains) the instruction at address 44AB hex:

```
loop that executes "44AB"
```

and here is the same with an offset address:

```
loop that executes offset "3A0"
```

When Bound-T lists the unbounded loops (a form of output described in the Reference Manual) the listing shows the offset from the start of the subprogram to the head of the loop. You can use this value to identify the loop by “executes offset”.

Nested loops

The way loops are nested can be used to identify a loop by identifying an inner or outer loop with the keywords **contains** or **is in**. See section 2.3 above for examples.

However, note carefully that an outer loop *inherits* most of the properties of its inner loops. Thus, if an inner loop calls a subprogram, Bound-T considers that the outer loop also does so because the outer loop also contains this call. The same goes for the properties **defines**, **uses**, and **is labelled**, but not for **executes**. You may have to extend the loop identification to compensate for this. For example, here is how to identify an outer loop that itself calls *Check_Power*, rather than inheriting that **calls** property from an inner loop:

```
loop that calls "Check_Power"
    and not contains (loop that calls "Check_Power")
```

Unfortunately this description does not match an outer loop that itself calls *Check_Power* if the inner loop also calls *Check_Power*.

As said, the **executes** property is not inherited from an inner loop to its outer loops. This also means that if you want to use **executes** to identify an outer loop, you must use an instruction address that is in the outer loop but is *not* in any nested, inner loop.

Multiple loop properties

The keyword **and** can be used to form the logical conjunction of loop properties for describing a loop or a set of loops. Here is how to assert that any loop that contains a call of *Set_Pixel* and is also within an outer loop that contains a call of *Clear_Row* repeats at most 600 times:

```
all loops that
    call "Set_Pixel"
    and are in (loop that calls "Clear_Row")
    repeat 600 times;
end loops;
```

Getting fancy

By combining properties, quite detailed and complex characterisations can be given, such as: The loop that is within a loop that calls *Foo*, and contains a loop that calls *Bar* but does not call *Fee*, and does not contain a loop that defines variable *Z*:

```
loop that
    is in (loop that calls "Foo")
    and contains (
        loop that
            calls "Bar"
            and does not call "Fee")
    and does not contain (loop that defines "Z")
```

However, it may make more sense to divide the program into smaller subprograms so that loops can be identified with simpler means.

All N loops

Sometimes the compiler makes loops in the machine code that do not correspond to loops in the source code. For example, an simple assignment of a multi-word value can lead to a machine-code loop that copies the words one by one. An "all loops" assertion will apply to such loops, too, so it may be safer to specify *how many* loops you expect to cover with the assertion. Put the number (or a number range) between **all** and **loops**, as in the following assertion that bounds the number of repetitions of the three loops in the subprogram *Tripler*:

```
subprogram "Tripler"
    all 3 loops repeat 25 times; end loops;
end "Tripler";
```

If Bound-T finds a different number of loops that match the assertion it reports an error. You must then change the assertion to identify the loops by some suitable properties.

You can use the **all** keyword and the optional number of matching loops in the same way also when the assertion uses loop properties. A loop-block that starts with **loop** without **all** is equivalent to "all 1 loops".

All N loops in any subprogram

When a loop block in the global context (not within a subprogram block) identifies a certain number of loops with **all**, the number of matching loops is counted separately for each analysed subprogram; it is not added up over the whole target program. Thus, if you write in a global context

```
all 2 loops repeat 27 times; end loops;
```

you are asserting that every subprogram to be analysed shall contain two loops and each of these loops repeats 27 times. This is an unrealistic example; it seems unlikely that all the subprograms have this structure. A more likely example could be the following:

```
all 0 .. 1 loops that call "PutStdErrChar"
    repeat <= 20 times;
end loops;
```

This assertion states that every subprogram to be analysed shall contain at most one loop that calls *PutStdErrChar*, and that this loop (if it exists) repeats at most 20 times. The former fact may reflect some design or coding rule for the program; the latter fact may show the maximum length of the error messages in this program.

Optimisation as the enemy

The assumption that these loop properties are invariant under optimisation is perhaps optimistic. Some optimisations that might alter the properties are listed below, together with some counter-measures.

- The **calls** property might be altered by inlining the called subprogram. Inlining can usually be prevented by placing the caller and callee in different compilation units (source files).
- The **uses** and **defines** properties might be altered by optimisation to keep the variable or parameter in a register. This can be prevented by specifying the variable as "volatile".
- The **uses** and **defines** properties might be altered by optimisation to move loop-invariant code outside the loop. This can be prevented by specifying the variable as "volatile".
- The **contains** and **in** properties might be altered if some loops are entirely unrolled.

While the WCET of an unrolled loop can be computed automatically, and thus an assertion on the repetitions of this loop is not needed, the disappearance of the loop means that it cannot be used to characterise a related loop with **contains** or **is in**.

Apparent but unreal looping and nesting

Sometimes a loop description derived from the source code fails to match the machine-code loop because the programmer has written, *within* the loop syntax, statements that are really *external* to the loop. For example, the following Ada loop seems to contain a call of the subprogram *Discard_Sample*:

```
for K in 1 .. N loop
    if not Valid(K) then
        Discard_Sample(K);
        exit;
    end if;
end loop;
```

Note that the call is followed by an **exit** statement that terminates the loop. Thus the call is logically *not* a part of the loop; the loop cannot repeat the call. This means that a loop description such as `loop that calls "Discard_Sample"` will *not* match this loop.

The same can happen with loop nesting. For example, at first sight this C code seems to contain nested loops:

```
for (k = 0; k < N; k++)
{
    if (overlimit[k])
    {
        for (i = 0; i < k; i++) recalibrate (i);
        return;
    }
}
```

Note that the inner loop (over *i*) is followed by a **return** statement that terminates the outer loop (over *k*). Thus the loop over *i* is logically *not* nested in the loop over *k*. This means that the *k* loop does *not* have the property `contains loop` and the *i* loop does *not* have the property `is in loop`.

3.7 Identifying calls

When writing assertions you may have to identify specific calls for the following reasons:

- To define a call context.
- To help identify a loop by identifying a call within the loop.

Unlike subprograms and variables, calls seldom have names and thus we identify calls indirectly through the properties or characteristics of the call, or through the source-code position of the call.

Static vs dynamic calls

The most important property of a call is whether the called subprogram – the callee – is statically defined in the call instruction, or is defined at run-time by some dynamic computation. Calls of the first kind are *static calls* and the others are *dynamic calls*.

On the source-code level static calls state the name (identifier) of the callee directly, while dynamic calls generally dereference a function pointer variable (in C terms) or an access-to-subprogram variable (in Ada terms). In the machine code, a static call instruction defines the entry address of the callee by an immediate (literal) operand, while a dynamic call uses a register operand or other dynamic operand.

A static call has exactly *one* callee; every execution of the call invokes the *same* callee subprogram. In contrast, a dynamic call may invoke different subprograms on each execution, depending on the entry address that is computed, so a dynamic call in general has a *set* of possible callees.

Call properties

Section 3.3 showed how to identify a call by its source-code position. The following other properties can be used to identify calls:

- The name of the called subprogram (callee). Required for static calls, absent for dynamic calls.
- The name of the calling subprogram (caller). Optional, since bounds on calls can appear globally or in the context of the caller.
- The identity of the containing loop. Optional.

All calls must be identified at least by the name of the callee or by saying that the call is dynamic. For a static call the syntax consists of the keywords **call** and **to** followed by the name of the callee (as explained in section 3.4). The **to** keyword is optional (syntactic sugar). A dynamic call is described as **dynamic call** without naming the callee.

To specify the caller, write the call-block within a subprogram block for the caller.

Examples of call identifications follow. The examples mainly show execution-count assertions, but of course the same call identifications can be used to assert other kinds of facts, such as bounds on variable values at the call.

The only call from here to there

The most common way to identify a call is by the names of the caller and the callee. If there is only one such call, no other call properties need be given. The assertion consist of a subprogram block that names the caller and contains the call block that names the callee. Here is how to assert that the only call from *Collect_Data* to *Flush_Buffer* is executed at most 4 times in one execution of *Collect_Data*:

```
subprogram "Collect_Data"
   call to "Flush_Buffer" repeats <= 4 times; end call;
end "Collect_Data";
```

The absence of the keyword **all** before **call** means that Bound-T expects to find *exactly one* call from *Collect_Data* to *Flush_Buffer*.

The only dynamic call

If a subprogram contains only one dynamic call it can be identified simply by this property – of being dynamic. Here is an assertion to say that the sole dynamic call in the subprogram *Dispatch* can only call the subprograms *Start_Pump* or *Start_Engine:*

```
subprogram “Dispatch”
    dynamic call calls “Start_Pump” or “Start_Engine”;
    end call;
end “Dispatch”;
```

All calls from here to there

Another case where no specific call properties need be given is when the same assertion applies to all calls from one caller to one callee. The keyword **all** is then placed before **call**, as in this example that asserts that no call from *Drive* to *Start_Motor* is executed more than once, in one execution of *Drive*:

```
subprogram "Drive"
    all calls to "Start_Motor" repeat 0 .. 1 times; end calls;
end "Drive";
```

All calls from anywhere to there

If the same assertion applies to calls from any caller to a given callee, the call block should be written in a global context (without an enclosing subprogram block). Here is how to assert that no subprogram ever executes more than one call to *Start_Motor*:

```
all calls to "Start_Motor" repeat 0 .. 1 times; end calls;
```

Call in a loop

In the current form of Bound-T, there are only two ways to identify a *subset* of calls from the same caller to the same callee: the first way is to describe the calls by the loops that contain them; the second way is to mark all calls by the same marker name. For example, here is how to assert that the (only) call from *Compute* to *Abort* that is in a loop is not executed at all:

```
subprogram "Compute"
    call to "Abort" that is in loop
        repeats 0 times;
    end call;
end "Compute";
```

This can also be done in a global context (not nested in a subprogram block). To assert that no call to *Abort* from an inner loop in any subprogram is ever executed, place the following assertion in a global context:

```
all calls to "Abort"
    that are in (loop that is in loop)
    repeat 0 times;
end calls;
```

Note that even if the source-code nests a call statement within the high-level syntax of a loop statement, this does not always mean that the machine-code call is *logically* within the loop. See the discussion of “apparent but unreal nesting” at the end of section 3.6.

Non-returning subprograms are never in a loop

A call to a subprogram that is marked "no return" (see section 2.17) is a special case. Logically, such a call is never contained in a loop because executing the call also means terminating any on-going loop.

All N calls

Some code transformations or optimizations in the compiler can change the number of machine-code call instructions (call sites) relative to the number of call statements in the source code. For example, unrolling loops can increase the number of call instructions, while merging duplicated code can decrease the number of call instructions. Neither transformation changes the *total* number of calls executed, but *can* change number of times each call instruction is executed. This should be taken into account in any "all calls" assertion on execution counts.

For example, assume that a source-code loop in the subprogram *Foo* contains two conditional calls to *Bar* and you know that each of these call statements is executed at most 10 times although the loop repeats a greater number of times. You could assert this fact as

```
subprogram "Foo"
    all calls to "Bar" that are in loop
        repeat <= 10 times;
    end calls;
end "Foo";
```

This assertion allows a total of at most 2 × 10 = 20 executions of *Bar* from the loop. However, if the compiler unrolls the loop body by duplicating it once, the machine-code loop will contain four instructions that call *Bar* and the above assertion would allow up to 4 × 10 = 40 executions of *Bar* from the loop, leading to an overestimated WCET.

To detect when code transformations change the number of call sites, you can specify how many call sites you expect to cover with the assertion. Put the number between **all** and **calls**:

```
subprogram "Foo"
    all 2 calls to "Bar" that are in loop
        repeat <= 10 times;
    end calls;
end "Foo";
```

All N calls from any subprogram

When a call block in the global context (not within a subprogram block) identifies a certain number of calls with **all**, the number of matching calls is counted separately for each analysed subprogram; it is not added up over the whole target program. Thus, if you write in a global context

```
all 2 calls to "Foo" repeat > 5 times; end calls;
```

you are asserting that every subprogram to be analysed shall contain two calls to *Foo* and each of these calls is executed more than five times for each execution of the calling subprogram. This is an unrealistic example; it seems unlikely that all the subprograms have this structure. A more likely example could be the following:

```
all 0 .. 1 calls to "PutStdErrChar"
    repeat <= 20 times;
end calls;
```

This assertion states that every subprogram to be analysed shall contain at most one call to *PutStdErrChar*, and that this call (if it exists) repeats at most 20 times for one execution of the calling subprogram. The former fact may reflect some design or coding rule for the program; the latter fact may show the maximum length of the error messages in this program.

Note that neither of these assertion examples bounds the *total* number of calls (call sites) in the program nor the *total* number of executions of these calls.

3.8 Identifying instructions

When writing assertions you will need to identify a specific machine instruction, in the program under analysis, for just one reason:

- To define a context that contains this instruction and nothing else.

The only way to identify an instruction is by its machine address. The address can be given directly, as an *absolute* address, or indirectly as an *offset* in machine-address units from the entry address of the subprogram that contains the instruction.

By absolute address

The syntax of machine addresses depends on the target processor. Typically, the address is written in hexadecimal form enclosed in (double) quotes. This example names the instruction at the hexadecimal address 4A0C and asserts that the instruction is exected at most 7 times:

```
instruction at “4A0C”
    repeats <= 7 times;
end instruction;
```

If the instruction is identified by its absolute address, the identification is sensitive to the memory lay-out of the program to be analysed. If the program is changed, recompiled, and relinked, the instruction may move to another address and then you must change the address in the assertion, too.

By offset from subprogram entry point

The syntax of an offset value also depends on the target processor, but is typically also written in hexadecimal form. For example, assuming that the subprogram that contains the instruction at address 4A0C (hex) starts at the address 4B02 (hex), the following identifies the instruction using the offset form and makes the same assertion as above:

```
instruction at offset “10A”
    repeats <= 7 times;
end instruction;
```

where the offset 10A (hex) is the difference between the instruction address 4A0C and subprogram entry point 4B02.

An offset is less sensitive to changes in the memory lay-out than the corresponding absolute address would be. As long as the subprogram that contains this instruction is not changed, the offset usually remains the same even if the program is recompiled and relinked (of course assuming the same versions of the compiler and linker and the same compilation and linking

options) and even if the subprogram now starts at a different address. Of course, if you change the subprogram itself, the offset to the interesting instruction may change and you must then change the offset in the assertion, too.

Some instruction sets divide the memory space into “pages” and require a longer branch instruction to jump across a page boundary than to jump to a location in the same page. For such target processors, the offset of a given instruction within a subprogram may depend on the absolute address of the subrogram, because the absolute address influences where page boundaries fall. Thus the sizes of the branch instructions in the subprogram may change, which changes the offsets of the instructions.

To be safe, you should check all addresses and offsets that your assertions use, after any changes to the target program and any recompilation and relinking.

4 ETERNAL LOOPS AND RECURSION

4.1 Handling eternal loops

What is eternity?

Much has been said about finding bounds on the number of iterations of loops. But what if the program contains an eternal loop?

We define an *eternal loop* as a loop that cannot possibly terminate, either because there is no instruction that could branch out of the loop, or because all such branch instructions are conditional and the condition has been analysed as infeasible (always false). Obviously, the execution time of a subprogram that enters an eternal loop is unbounded. Nevertheless, since real-time, embedded programs are usually designed to be non-terminating, they usually contain eternal loops.

Eternal tasks

Eternal loops are typically used in the top-level subprograms of tasks or threads. The loop body first waits for the event or real-time instant that should activate (trigger) the task, then executes the actions of the task, and then loops back to wait for the next activation.

A typical task body in the Ada language has the following form:

```
task body Sampler is
begin
   loop
      wait for my trigger;
      execute my actions;
   end loop;
end Sampler;
```

Here we have a *syntactically* eternal loop: there is no statement that terminates or exits the loop. (The loop could be terminated by an exception, but Bound-T generally does not consider exceptions in its analysis.)

The same task in the C language might have the following form:

```
void Sampler (void)
{
   while (1)
   {
      wait for my trigger;
      execute my actions;
   }
}
```

Here we have a *logically* eternal loop: in principle, the **while** statement can terminate the loop if its condition becomes false; however, the condition is always true here.

For a logically eternal loop the compiler may or may not generate a conditional branch instruction to exit the loop. If the compiler finds it unnecessary to generate an exit branch, the loop will be syntactically eternal on the machine code level. If the compiler does generate an

exit branch, Bound-T will probably discover that the branch condition is always false, whereupon Bound-T will prune (remove) the infeasible exit-branch from the control-flow graph and find that the loop is indeed eternal.

Bounding eternity

When Bound-T finds an eternal loop in a subprogram it of course reports it and refuses to compute an execution time bound for the subprogram – unless you assert a bound on the number of repetitions of the loop. But what is the point of such an unrealistic assertion? The point is that you usually need an upper bound on the execution time of *one activation* of a task: the statements illustrated as "execute my actions" in the examples above, perhaps including all or part of the statement "wait for my trigger" depending on where you draw the boundary between the application task and the real-time kernel. Thus, you need a WCET for the loop body, which is one iteration of the loop.

Whatever repetition bound you assert for the eternal loop, the WCET that Bound-T computes also includes the code that leads from the subprogram entry point into the loop. The way to find a WCET bound for one loop iteration is therefore to analyse the subprogram twice, with the repetition bounds 0 and 1 (for example), and take the difference of the results.

To avoid this eternal loop stuff, you could separate all the code for one task activation into a dedicated subprogram so that the eternal loop just contains a call of this subprogram. The WCET bound for this subprogram is very close to the WCET bound for one task activation; the difference is just the call instruction and the looping branch instruction, usually just a pinch (less than a handful) of machine cycles.

Eternity as an alternative

Sometimes an eternal loop is used as a last-resort error-handler, for example as in the following:

```
void Check_Voltage (void)
{
    if (Supply_Voltage() < Min_Supply_Volts)
    {
        // The supply voltage is too low.
        // Wait in a tight loop for a reset.
        while (1);
    }
    // The supply voltage is good. Display it.
    Display_Voltage();
}
```

In this case, you probably want an execution-time bound for this function that does not include the eternal loop. You should then use assertions to exclude the loop from the analysis. In the example above you can assert that the call to *Display_Voltage* actually occurs. However, Bound-T also requires a bound on the loop, so the assertions would be:

```
subprogram "Check_Voltage"
    call to "Display_Voltage" repeats 1 time; end call;
    loop repeats 0 times; end loop;
end subprogram;
```

The number of repetitions asserted for the loop is arbitrary, because the assertion on the call means that the loop is never entered (assuming that the compiler or Bound-T detects that the loop is eternal so that there is no execution path from the loop to the call).

4.2 Handling recursion

The perils of recursion

Guidelines for embedded and real-time programming usually advise against recursion because recursion is often associated with dynamic and unpredictable time and memory consumption. Moreover, some small embedded processors (microcontrollers) have poor mechanisms for stacks and subprogram calls, which means that a reentrant or recursive subprogram must use slower or less efficient code for parameter passing and local variables. These are some of the reasons why Bound-T assumes that the target program is free of recursion.

Trivial recursions: an example

Sometimes target programs use recursion in very limited and predictable ways. For example, an error-logging module may want to log some of its own errors, such as the fact that the log buffer was full and some (real) errors were not logged. While this could certainly be programmed without recursion, it gives us a simple example of limited recursion and how to handle it in Bound-T. This example is taken from a real application.

Let's define the interface of the error-logging module as follows (example in Ada):

```
package Errors is

   type Message_Type is Integer;
   -- An error message is just an integer number here.
   -- Really it would be something more.

   Log_Full : constant Message_Type := 99;
   -- An error message that means that the Error Log became
   -- full and some error messages were not logged. This is
   -- always the last message in the (full) log.

   procedure Handle (Message : in Message_Type);
   -- Handles the error Message and then inserts the
   -- Message in the Error Log.
   -- If the Error Log would then be full, the Log_Full
   -- message is inserted instead of the Message, and is
   -- also handled as an error message in its own right.

end Errors;
```

This module could be implemented as follows:

```
package body Errors is

   Buffer_Size : constant := 100;
   -- The total size of the buffer for the Error Log.

   Buffer : array (1 .. Buffer_Size) of Message_Type;
   -- The buffer itself.
```

```
    Free : Natural := Buffer_Size;
    -- The space left in the buffer.

    procedure Log (Msg : in Message_Type)
    -- Inserts the Msg in the Buffer and decrements the count
    -- of the remaining space. If this would make the log
    -- quite full, the procedure signals a Log_Full error.
    is begin
       if Free = 1 and Msg /= Log_Full then
          -- The buffer is full, the Msg is not logged.
          Handle (Log_Full);
       else
          Free := Free – 1;
          Buffer(Buffer'Last – Free) := Msg;
       end if;
    end Log;

    procedure Handle (Message : Message_Type)
    is begin
       Handle the Message in some way;
       Log (Message);
    end Handle;

 end Errors;
```

Here you can see that buffer overflow is detected in the lowest-level subprogram *Log*, but to report the overflow it calls *Handle* (*Log_Full*), which creates a recursion: *Handle* → *Log* → *Handle*. However, *Log* calls *Handle* only if the *Message* is not *Log_Full*, which means that the recursion terminates in the second call of *Log*. The longest possible call-path is thus

Handle → *Log* → *Handle* → *Log*

This call-path determines the WCET of *Handle*. The figure below illustrates the path when the incoming *Message* has the value 31.

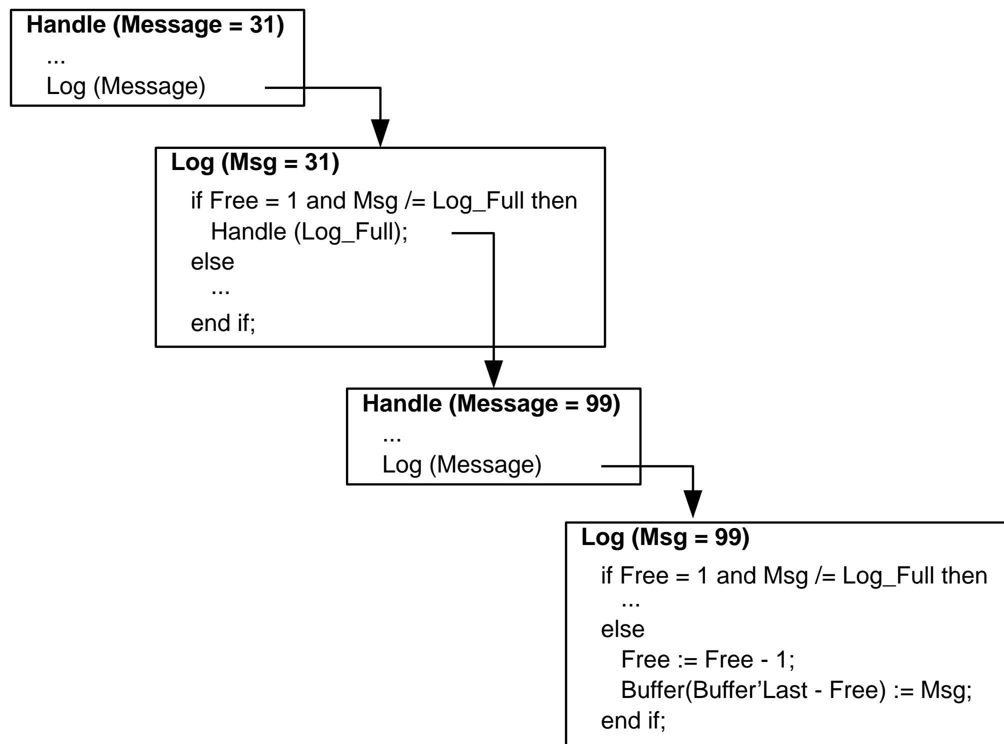

Figure 7: Longest call path in recursion example

Slicing recursive call-paths

How can we find an upper bound on the execution time of the recursive call-path in the above example? Asking Bound-T to analyse *Handle* will just result in error messages that complain about the recursion.

You can make Bound-T analyse a *piece* of a recursive call-path by asserting the execution time of one of the subprograms in the call-path. The calls in this subprogram are thereby hidden from Bound-T which breaks the recursive cycle (if there are several recursion cycles you may have to break the other cycles in the same way). This analysis gives the WCET for the rest of the call-path. Then you analyse the call-path again but this time you assert the execution time of another subprogram in the call-path. You can then combine the WCET bounds on the pieces to compute the WCET bound for the whole call-path. However, you also have to be careful to guide Bound-T to choose the right paths *within* each subprogram. Below we show how to do it for the example.

Slicing the example

For our example we can start by hiding the *Log* subprogram and analysing the *Handle* subprogram. Since *Handle* always calls *Log*, the analysis always includes the desired path within *Handle* whatever execution time we assert for *Log*; assume we choose 0 cycles so that the assertions for this analysis are

```
subprogram "Errors.Log" time 0 cycles; end "Errors.Log";
```

Assume that the resulting WCET bound for *Handle* is 422 cycles. Since zero cycles are assumed for *Log* this means that the WCET for *Handle* alone is 422 cycles.

Next, we hide the *Handle* subprogram and analyse *Log*. Since *Log* contains a conditional statement we must choose which path to analyse. In fact, both cases occur in the recursive call-path we are considering: the first call of *Log* uses the path within *Log* that calls *Handle*, and the second call of *Log* uses the other path within *Log*, the one that actually inserts the error message in the buffer. Therefore we must analyse both cases.

To analyse the first path within *Log* we could either assert a very large time for *Handle*, so that the path that calls *Handle* surely seems to take longer than the other path, or we can force Bound-T to choose this path in some other way, as in these assertions:

```
subprogram "Errors.Handle"
   time 0 cycles;
end "Errors.Handle";

subprogram "Errors.Log"
   call to "Errors.Handle" repeats 1 time; end call;
end "Errors.Log";
```

Assume that this gives a WCET bound of 56 cycles for *Log*. Since zero cycles are assumed for *Handle* these 56 cycles are also the WCET bound for the first call of *Log* in the recursive call-path.

To analyse the second path within *Log* we use analogous but opposite assertions for *Log*. We must still also assert an execution time for *Handle*, to hide it from Bound-T, but now the asserted time plays absolutely no role because it is not included in the WCET for *Log*:

```
subprogram "Errors.Handle"
   time 0 cycles;  -- This time is irrelevant.
end "Errors.Handle";

subprogram "Errors.Log"
   call to "Errors.Handle" repeats 0 times; end call;
end "Errors.Log";
```

Assume that this gives a WCET bound of 28 cycles for *Log*. Since the assertions exclude the *Handle* call these 28 cycles are directly the WCET bound for the second call of *Log* in the recursive call-path.

Finally, we add up the WCET bounds for the recursive call-path:

- 422 cycles for the first call of *Handle*,
- 56 cycles for the first call of *Log*,
- 422 cycles for the second call of *Handle*,
- 28 cycles for the second call of *Log*.

The total, 928 cycles, is the WCET bound for the recursive call-path.

In summary, to analyse a recursive set of subprograms you must yourself find out the longest (slowest) recursive call-path, break that call-path into at least two non-recursive pieces, analyse them separately, and add up the results. Sometimes the longest call-path can be found or seen easily, as in this example; if that is not the case, you may have to consider a number of candidates for the worst-case call-path and analyse each candidate as shown here.

5 ASSERTION LANGUAGE SYNTAX AND MEANING

5.1 Introduction

The command-line option *-assert filename* makes Bound-T read the assertions from the text file by the name *filename*. Chapter 2 explained why and how to use the assertion language with examples. The present chapter defines the precise syntax and meaning of the assertion language. A formal grammar notation defines the syntax. Informal prose explains the meaning of each grammar symbol and production.

5.2 Assertion syntax basics

Syntax notation as usual

A conventional context-free syntax notation is used, with nonterminal symbols in Plain Style and Capitalised; literal keywords in **bold** style; and user-defined identifiers in *italic* style. However, when nonterminal symbols are quoted in the running text we use *Italic Capitalised Style*.

Alternatives are separated by '|'. Repetition of one or more symbols for one or more times is denoted by enclosing the symbol(s) between curly brackets '{' and '}'. Optional symbols are enclosed between square brackets '[' and ']'.

The symbol *character* stands for any printable character enclosed in single quotes (apostrophes). Example: 'x'.

The symbol *null* stands for the empty string.

The symbol *integer* stands for a string of digits 0 .. 9 representing an integer number in decimal form. A sign (+, -) may precede the integer. The underscore character '_' can be used in the string to group digits with no effect on the numeric value. For example, 33_432_167 means the same as 33432167. The numeric range of integers may depend on the host platform and target system, but is at least -2^{31} .. $2^{31}-1$.

The symbol *string* stands for any string of printable character enclosed in double quotes. Example: "foo|memo". To write a string that itself contains double-quote characters, write each double-quote character twice. For example, “foo””tick””” represents the string foo”tick” that contains one internal and one trailing double-quote character.

Comments

An assertion file may contain comments wherever whitespace can appear. A comment begins with two consecutive hyphens (--) and extends to the end of the line.

Symbols with scopes

The symbol *symbol* stands for a *string* which is interpreted as the identifying symbol of an entity (a subprogram, a variable or a statement label) in the target program. Even though the assertion syntax as such does not restrict the contents of *symbol* strings they must follow a pre-defined (target-dependent) format.

If the *symbol* string contains occurrences of the current scope-delimiter character, these divide the string into a sequence of scope names followed by a basic name. For example, using the default delimiter character '|', the *symbol* string "API|Init" is considered to consist of the scope "API" and the basic name "Init". The scope delimiter character is set by the **delimiter** keyword as explained below.

Finally, the interpretation of *symbol* strings may be affected by the current default scope string set by the **within** keyword as explained below.

The target compiler and linker may modify the symbols for subprograms and variables so that assertions have to name them in a different way than by using the name in the source-code file. These name-mangling rules are discussed in the Application Notes for the respective target processors.

You can find out the symbols that are available in the target program by dumping the target program as explained in section 3.4.

Machine addresses

The strings following the keyword **address** are denoted by the symbol *address* and identify a target-program element in some low-level, machine-specific way, such as by its memory address.

Each target processor to which Bound-T is ported has a specific "sub-syntax" for *address* strings. The syntax may also be different for variable addresses and for code (subprogram or label) addresses, and so we use the symbols *variable-address* and *code-address*, respectively, for these. Some assertions may use code offsets instead of absolute code addresses, and then we use the symbol *code-offset*. The syntax of *code-offset* is also target-dependent.

From the user's point of view, the *address*, *variable-address*, *code-address* or *code-offset* is written as a *string* (ie. enclosed between double quotes). Scope delimiters and the current default scope play no role in the handling of *address* strings.

Instruction roles

The strings following the keyword **performs** (and an optional sugar keyword **a** or **an**) are denoted by the symbol *instruction-role* and identify the role performed by a specific instruction in the target program under analysis.

Each target processor to which Bound-T is ported has a set of possible values of *instruction-role* strings, often including strings like "call", "branch", "return".

From the user's point of view, the *instruction-role* is written as a *string* (ie. enclosed between double quotes). Scope delimiters and the current default scope play no role in the handling of *instruction-role* strings.

Variable names as used in several places

An element that will occur in several syntactic forms is the variable name:

Variable_Name → *symbol* | **address** *variable-address*

The variable is identified either by its high-level *symbol*, which is a possibly qualified, possibly mangled source-level identifier enclosed in double quotes, or by its low-level address, which can be a data-memory address or a register name, also enclosed in quotes. The *variable-address* part is written in a syntax that is specific to the target processor and explained in the relevant Application Notes.

Bounds as used in several places

Another element that will occur in several syntactic forms is the definition of bounds on a number:

Bound → *integer*
| **=** *integer*
| *integer* **..** *integer*
| **..**
| *integer* **..**
| **..** *integer*
| **>** *integer*
| **>=** *integer*
| **<** *integer*
| **<=** *integer*

A *Bound* defines an interval subset of the integers as follows. If a single *integer* is given, possibly preceded by an equals symbol (=), the subset consists of this value only. If two *integers* are given separated by two periods (..) the subset consists of the interval from the first *integer* to the second *integer*, inclusive.

If only the two periods are given, with no *integer* before or after, the subset contains all values. The rest of the forms define subsets that are bounded at only one end: If an *integer* is followed by two periods (without a following second *integer*) the subset consists of all values greater or equal to the given *integer*. Conversely, when the two periods precede the single *integer* the subset consists of all values less or equal to the given *integer*. If a relational symbol followed by an *integer* is given, the subset consists of those values that stand in the given relation to the given *integer*. Thus, the *Bound* form “>= 4” has the same meaning as “4 ..”, and “<= 4” has the same meaning as “.. 4”.

In some contexts, the subset can contain only non-negative values. For example, a bound on the number of executions of a call or the number of repetitions of a loop contains an implicit lower bound of zero even if the *Bound* explicitly states only an upper bound or no bound at all.

Singular and plural keywords and other alternatives

Some keywords can be written in singular or plural form, interchangeably, to make the assertion syntax closer to normal grammar. To avoid clutter in the grammar, the grammar rules use only one form, but the assertion text can use either form (with a few exceptions explained later). Some verb-like keywords such as **contain** have a third equivalent *-ing* form: **containing**. Moreover, a few keyword pairs have obsolete single-word equivalents with embedded underscores. Here are the equivalent keywords:

First form	*Equivalent second and third forms*	
call	**calls**	**calling**
contain	**contains**	**containing**
cycle	**cycles**	
define	**defines**	**defining**
do	**does**	
execute	**executes**	**executing**
is	**are**	
loop	**loops**	
repeat	**repeats**	
return	**returns**	
span	**spans**	**spanning**
start	**starts**	
time	**times**	
use	**uses**	**using**

Keyword pair	*Obsolete but equivalent keyword*

call to	**call_to**
end call	**end_call**
end loop	**end_loop**
is in	**is_in**
loop that	**loop_that**
no arithmetic	**no_arithmetic**

Using these alternative forms, we can use the singular forms when proper:

```
loop that calls "Foo" repeats 1 time; end loop;
```

We can instead use the plural forms when they are more suitable:

```
all loops that call "Foo" repeat 10 times; end loops;
```

For variation, the *-ing* forms can be used, too:

```
all loops calling "Foo" repeat 10 times; end loops;
```

Bound-T does not check that the equivalent forms are used consistently within each assertion, so you can also say, ungrammatically but acceptably:

```
all loop that calling "Foo" repeats 10 time; end loops;
```

Here are the exceptional cases where the different forms of a keyword are not equivalent:

- The keywords **call** and **calls** are not allowed immediately after the keywords **loop** or **loops**, although **calling** is allowed. See section 5.7 (page 87).
- An *Execution_Time_Bound* accepts only the keyword **time**, not **times**. See section 5.11.

Source-code position as used in several places

Some program parts (currently loops and calls) can be identified by their source-code position as follows.

Source_Position → Source_Relation Source_Point

The *Source_Point* identifies a source-code line; the *Source_Relation* limits the intended position to code at, before, or after this line:

Source_Relation	→	[**exactly**] At_On
	|	**before**
	|	**after**
At_On	→	**at** | **on**

The keywords **at** and **on** are equivalent. The optional keyword **exactly** means that the line numbers shall match exactly (zero fuzz). The keywords **before** and **after** affect the default fuzz used for comparing source-line numbers when looking for this source position, as shown in Table 2 below. In the table *Z* means the default amount of fuzz (difference in line number) that is set with the Bound-T command-line option *-line_fuzz*.

Table 2: Line-number comparison fuzz

Source relation	Fuzz bounds (when *-line_fuzz Z*)
exactly at/on	0 .. 0
at/on	$-Z$ .. Z
before	$-Z$ .. 0
after	0 .. Z

The *Source_Point* construct identifies a source line (number):

Source_Point → Source_Line Source_Context

Source_Line → **line** *integer*
| **line offset** *integer*
| **marker** *string*

The *integer* after **line** is an absolute line number; it must be positive. Line 1 is the first line in a file. The *integer* after **line offset** is a line-number offset relative to the first line of the containing subprogram; it can be zero or negative, but is usually positive.

In the third and last form, the *string* after **marker** is a marker name and matches any mark with exactly this name. Such a source position can match several different marks (positions) in the same or different files, as long as they use the same marker name.

Source_Context → [**within** Bound] [**in** Source_File]

The optional *Bound* sets the fuzz for comparing source-line numbers while looking for this source position. It overrides the default fuzz defined by the *Source_Relation* and Table 2.

The optional *Source_File* limits the matching source positions to lines in the named source-code file. If absent, lines in any file can match this source position.

Source_File → [**file**] *string*

The optional **file** keyword has no effect. The *string* is the quoted name of the source-code file. Depending on the Bound-T command-line option *-file_match* the comparison of the *string* to the file-names given in the symbol table of the target program can be exact or approximate in one or both of the following respects:

- Under *-file_match base* the comparison ignores all directory/folder paths and compares only the base-names. For example, "sources/subs.c" matches "archive/subs.c" and also "subs.c".
- Under *-file_match cis* the comparison is case-insensitive (both names are converted to lower-case for the comparison). For example, "subs.c" matches "Subs.C".

An exact match is required under *-file_match full -file_match cs*.

5.3 Overall assertion structure

The grammar's start symbol is *Assertions*, representing a whole assertion file. An assertion file is a non-empty list of four types of elements: scope delimiter definitions, scope definitions, global bounds and subprogram blocks:

Assertions { Scope_Delimiter | Scope | Global_Bound | Sub_Block }

The order of the elements is arbitrary except that the scope-delimiter definition and scope definition have an effect only on the following elements, up to the next such definition. In summary, the elements have the following roles:

- A *Scope_Delimiter* defines the character that separates scope levels in all *symbols* in the following assertions. It does not itself assert anything.
- A *Scope* defines a default scope string to be assumed for all *symbols* in the following assertions. It does not itself assert anything.
- A *Global_Bound* asserts facts that apply in all subprograms to be analysed. The assertions can further delimit their context to selected calls, loops, or instructions in each subprogram.
- A *Sub_Block* asserts facts that apply in (or to) one subprogram. The assertions can further delimit their context to selected calls, loops, or instructions in this subprogram.

5.4 Scopes

The assertion language lets you set the scope delimiter character and the default scope. The role of these items in the interpretation of scope-qualified *symbols* was explained in sections 3.2.

Scope delimiter definitions

Scope_Delimiter → **delimiter** *character*

Sets the delimiter character to be used for parsing any *symbol* strings in the following assertions. The default delimiter is the vertical bar or solidus '|'. It is necessary to change the delimiter only if this character occurs within a scope-name or an identifier.

Scope definitions

Scope → **within** *string*

Sets the default scope string to be prefixed to any *symbol* strings in the following assertions, unless the *symbol* string itself starts with the delimiter character.

For example, if the module *API* contains a subprogram *Init* and the delimiter character is the default ' |' so that the full name of this subprogram is "API|Init", after the *Scope* definition

within "API"

the subprogram can be named either as "Init" or as "|API|Init"; both are equivalent to "API|Init". However, the string "API|Init" would be interpreted as "API|API|Init" which would probably not be the name of any subprogram.

5.5 Global bounds

Any assertions that occur outside subprogram blocks (outside any *Sub_Block* construct) are global bounds and are considered valid throughout the target program under analysis. There are five types of global bounds, namely variable bounds, property bounds, volatility marks, loop blocks and call blocks:

Global_Bound	→	Variable_Bound **;**
	|	Volatility_Mark **;**
	|	Property_Bound **;**
	|	Loop_Block **;**
	|	Call_Bock **;**
	|	Instruction_Bock **;**

The order of the global assertions is arbitrary. The syntax and meaning of each type of global bound are explained later.

All these types of bounds except volatility marks can also be asserted within a subprogram block and thus applied only to that subprogram. When the bound is written as a global one (not within a subprogram block), it is applied in the analysis of *each* subprogram, just as if it were written within a subprogram block for that subprogram.

For global loop blocks and call blocks the *Population* specified for the block (see section 5.7) is counted within each analysed subprogram, not added up over all subprograms. For example, if the *Population* of a loop block is 2, then this loop block should match exactly two loops in each subprogram that is analysed.

5.6 Subprograms

Subprogram blocks and subprogram names

A subprogram block collects assertion statements that shall be applied only to the analysis of a certain subprogram. The subprogram can be identified by a symbolic name or a machine-level entry address. An optional offset can be added.

Sub_Block	→	**subprogram** Sub_Name [**(** { Parameter } **)**] [{ Statement }] [**end** [**subprogram**] [Sub_Name] **;**]
Sub_Name	→	Sub_Base [**offset** *code-offset*]
Sub_Base	→	*symbol* **|** **address** *code-address*

The optional *Parameter* part contains the assertions in the subprogram *entry* context. The optional *Statement* part contains the assertions in the subprogram *body* context.

The **end** part that closes the subprogram block is optional but can be used to show that any following variable bounds, loop blocks *etc*. are global bounds and not specific to this subprogram. If the **end** part contains a *Sub_Name,* this must be exactly the same as the *Sub_Name* at the start of the block.

Subprogram parameter assertions

In a subprogram entry context, only assertions on variable values are allowed:

Parameter Variable_Bound ;

These variable bounds apply at a single point in the program: immediately before the first instruction in the subprogram. The bounded variables can be formal parameters or global variables or registers.

Subprogram body assertions and options

Several types of assertions can be stated in a subprogram body context, in any order, and the order is not significant:

Statement	→	Sub_Option ;
	|	Loop_Block ;
	|	Call_Block ;
	|	Instruction_Block ;
	|	Clause

This rule has no semicolon after the *Clause* alternative, because as will be seen later each *Clause* contains its own terminating semicolon.

A *Sub_Option* can require or forbid the arithmetic analysis of the subprogram, can declare the subprogram as "non-returning", and can specify “integrated” analysis of the subprogram:

Sub_Option	→	**arithmetic**
	|	**enough for time**
	|	**hide**
	|	**integrate**
	|	**omit**
	|	**return** [**normal** | **normally**]
	|	**return** [**to**] **offset** *code-offset*
	|	**unused**
	|	**used**
	|	Nix Sub_Option
Nix	→	**no**
	|	**not**

The **arithmetic** option can locally override the command-line option for arithmetic analysis (*-arithmetic* or *-no_arithmetic*, see the Bound-T Reference Manual) and the automatic decision that checks if arithmetic analysis is needed for a particular subprogram.

The option composed of the three keywords **enough for time** may be useful for subprograms that have an irreducible control-flow graph for which, thus, Bound-T cannot determine the loop structure and cannot compute loop repetition bounds nor accept assertions on loop repetition bounds. If you can assert bounds on the number of repetitions of some repeating parts (calls) of the subprogram, these assertions may be strong enough to let the Integer Linear Programming stage (IPET method) compute an overall execution-time bound even for an irreducible flow-graph. When you assert **enough for time** Bound-T will try the ILP phase even if the flow-graph is irreducible or if some loop bounds are unknown in a reducible flow-graph.

Most subprograms eventually return to the caller but some do not. Non-returning subprograms are typically those that raise exceptions or terminate the program in some other way, for example the *_exit* function in C. When Bound-T finds a call to a subprogram that is marked **no return**, Bound-T will consider that the call terminates the caller’s execution. This can simplify and improve the analysis of the caller, especially if the cross-compiler also knows that the call never returns (perhaps because the callee is a compiler-defined error handler).

Most subprograms return to the return address offered by the caller, in accordance with the calling protocol defined for the target processor or used by the compiler. Some subprograms, however, return to a different address. If the actual return address has a constant offset from the offered (normal) return address, use the **return to offset** *code-offset* option to inform Bound-T of this unusual behaviour of the subprogram. This option cannot be negated so it cannot be preceded by an odd number of negation keywords. If the actual return address is computed in some more complex way, ask Tidorum for help.

If you want to emphasize that a subprogram returns, and returns to the normal return address, you can use the option **return normal**, which, again, cannot be negated.

The **integrate** option means that any call to this subprogram will be analyzed as if the code of the subprogram were an integral part of the calling subprogram. In other words, the flow-graph of the callee subprogram will become a part of the flow-graph of the caller, as if the compiler had inlined the callee. This option is useful for subprograms that do not follow the normal calling protocols. For example, some compilers use special “helper” routines to set up the stack frame on entry to an application subprogram (prelude code) and to tear down the stack frame before return from the aplication subprogram (postlude code). Such routines often violate the normal calling protocols and must be analyzed as integral parts of their callers.

The **unused** option means that this subprogram should be excluded from the analysis. This has two consequences: firstly, the subprogram itself is not analysed; secondly, any call to this subprogram is considered to be infeasible. This option can be written either as **unused** or as **not used**. It is an error to say just **used**, or **not unused**; subprograms are "used" by default.

The **omit** option prevents Bound-T from analysing this subprogram. If an analysed subprogram contains a feasible call to an omitted subprogram you must assert the resource consumption (time and/or space) of the omitted subprogram, or of the call; otherwise the call and the caller are unbounded. Note that **omit** does not make calls to the omitted subprogram infeasible (as **unused** does); **omit** only prevents the analysis of the omitted subprogram but calls to the subprogram are included in the analysis.

The **hide** option excludes this subprogram from the call-graph drawings. It has no effect on the analysis; the subprogram is still analysed and included in the analysis of other subprograms that call it. Some programs have subprograms that are called from many places (for example, floating-point library subprograms such as *sin* and *cos*) which makes the call-graph very cluttered; using **hide** for such subprograms makes the call-graph clearer for the other subprograms. Note that the callees of a hidden subprogram are not automatically hidden, too; they may need their own **hide** options.

The **no** and **not** keywords (which here are equivalent) negate the option setting. The keyword can be repeated, so **no no return** is the same as **return**. This may be useful in assertion files constructed by scripts or preprocessors. However, the properties **integrate**, **omit**, and **unused** cannot be negated (disabled); they can only be asserted (enabled). It follows naturally that the **used** property cannot be asserted as such, only in the negated form as **not used**. This is no limitation because the forbidden forms of these properties correspond to the defaults that Bound-T assumes in the absence of any assertion: not integrated, not omitted, and used.

5.7 Loops

Loop blocks and populations

A loop block describes a set of loops and applies assertion clauses to all of these loops:

Loop_Block → Population Loop_Description { Clause } **end loop**

If the *Loop_Block* occurs within a *Sub_Block*, the loop block and its assertion clauses apply to the described loops in this subprogram only. If the *Loop_Block* occurs as a *Global_Bound*, it applies to the described loops in any analysed subprogram.

Population → [**all**] [Bound]

The *Population* part defines how many loops we expect to match the loop-description, in each subprogram to which this *Loop_Block* applies. An empty *Population* is the same as "1", that is we expect exactly one matching loop. If the keyword **all** appears without the *Bound* part, any number (zero or more) of loops can match. If the *Bound* part is included (with or without **all**) it defines the allowed range for the number of matching loops.

If the number of matching loops in the subprogram under analysis violates the *Population* range, Bound-T emits an error message.

Loop descriptions and loop properties

The set of loops to which a *Loop_Block* applies is described (identified) by the properties of the loops or by their source-code positions:

Loop_Description → **loop** [Loop_Properties]

Loop_Properties → Loop_Property [**and** Loop_Properties]

If the loop description contains no *Loop_Properties* any loop matches the description. If some properties are listed, a loop matches the description if and only if all the listed properties are true for this loop.

Loop_Property → Property_Prefix Basic_Loop_Property

Property_Prefix → [**that**] [Do_Is] [{ **not** }]

Do_Is → **do** | **is**

The optional keywords **that**, **do** (or **does**), and **is** (or **are**) have no logical meaning and are used only to make the text more grammatically pleasing. Each occurrence of the keyword **not** inverts the logical sense of the *Basic_Loop_Property*. Thus, an even number of **not**s has no effect and any odd number of **not**s has the same effect as one **not**.

An important property of a loop is whether it is nested in outer loops or contains inner loops or contains calls of some kind. For this we define:

Other_Loop → **loop**
| **(** Loop_Description **)**

Other_Call → Some_Call
| **(** Call_Description **)**

Count → [Bound]

The constructs *Some_Call* and *Call_Description* are defined in section 5.8 below. The *Count* defines how many inner loops or calls of a certain kind are required. An empty *Count* means "at least 1" so it is the same as a *Count* of ">= 1", or equivalently "1 ..".

The basic loop properties are then defined as follows:

Basic_Loop_Property → Source_Position
| **marked** *string* Source_Context
| **in** Source_File
| **in** Other_Loop
| **contains** Count Other_Loop
| **contains** Count Other_Call
| **contains** Source_Point
| **spans** Source_Point
| **calls** Sub_Name
| **uses** Variable_Name
| **defines** Variable_Name
| **labelled** Label_Name
| **executes** *code-address*
| **executes offset** *code-offset*

Label_Name → *symbol*

Table 3 below defines exactly the meaning of each type of loop property in its positive form (that is, assuming no negation through preceding **not** keywords).

Table 3: Meaning of loop properties

Property	Loop *L* has this property if and only if:
Source_Position	Some instruction in the loop-head node of *L* or in the start nodes for *L* is connected (through the compiler-generated mapping) to the source-code line identified by the *Source_Position*, within the precision allowed by the fuzz of the *Source_Position*. The term "start nodes" is defined in section 5.13 and illustrated in Figure 8 in that section.
marked *string* Source_Context	Same as for the *Source_Position* consisting of "**on marker** *string Source_Context* ".
in Source_File	Some instruction (step) in *L* is connected to a source-code line in the file named by the *Source_File*. Note that some (other) instructions in *L* can be connected to other source-code files.
in Other_Loop	*L* is directly contained in an outer loop that matches the *Other_Loop* description.
contains Count Other_Loop	Within the set of all inner loops directly contained in *L*, the given *Count* of loops match the *Other_Loop* description.
contains Count Other_Call	*L* (or some inner loop) contains the given *Count* of calls matching *Other_Call*.
contains Source_Point	*L* (or some inner loop) contains an instruction that is connected to the source-code line identified by the *Source_Point*, within the precision allowed by the fuzz of the *Source_Point*.
spans Source_Point	If the *Source_Point* specifies a *Source_File* : The line number specified in the *Source_Point* falls in the interval of lines from this source file connected to some instructions in *L*, within the precision allowed by the fuzz of the *Source_Point*. If the *Source_Point* does not specify a *Source_File* : For some source file connected to some instructions in *L* the line number specified in the *Source_Point* falls in the interval of lines from this source file connected to some instructions in *L*, within the precision allowed by the fuzz of the *Source_Point*.
calls Sub_Name	Same as "**contains** >= 1 **calls to** *Sub_Name*".
uses Variable_Name	*L* (or some inner loop) contains an instruction that reads (uses) the value of the variable identified by *Variable_Name*.
defines Variable_Name	*L* (or some inner loop) contains an instruction that writes (assigns a value) to the variable identified by *Variable_Name*.
labelled Label_Name	*L* (or some inner loop) contains the instruction that has the code address assigned to *Label_Name*.
executes *code-address*	*L* (or some inner loop) contains the instruction that has the given code address.
executes offset *code-offset*	*L* (or some inner loop) contains the instruction at the given offset from the start (entry point) of the subprogram that contains the loop. This form can be used only when the containing subprogram is given, that is, within a *Sub_Block*.

Note that the properties that state something about what the loop contains are usually satisfied also when the desired item is actually in some inner loop, nested to any depth. For example, if an inner loop contains a call to *Foo* then also any outer loop has the property **calls** "Foo". If it is necessary to select only loops that directly contain the desired item, an additional "does not contain (loop ...)" property must be written, for example as in:

```
loop that calls "Foo"
   and does not contain (loop that calls "Foo")
```

However, this loop-description will not match a loop which directly contains a call to *Foo* and also contains an inner loop that calls *Foo*, so it may be too limiting.

No “call” or “calls” immediately after “loop”

As an exception to the lack of meaning of the keywords **that**, **do**, **is** and to the equivalence of the three forms of the keyword **calls**, the first *Loop_Property* after **loop** or **loops** must not start with the keywords **call** or **calls**. This avoids ambiguity in assertions that use **call**/**calls** to specify the possible callees of a dynamic call, while identifying the dynamic call using a containing loop, such as the following (which is forbidden by this rule):

```
dynamic call in loop calls “foo”; end call;
```

This could be read in two ways. The first meaning could be this:

```
dynamic call in (loop calls “foo”); end call;
```

Here **calls** “foo” describes a property of the loop that contains the dynamic call. The second meaning could be this:

```
dynamic call in (loop) calls “foo”; end call;
```

Here **calls** “foo” asserts the (single) possible callee of the dynamic call. It is easy to avoid writing **call** or **calls** immediately after **loop**. If the first meaning is intended, just insert a **that**, or write **calling**, as in:

```
dynamic call in (loop that calls “foo”) ...
```

or

```
dynamic call in (loop calling “foo”) ...
```

You must enclose the loop description in parentheses as required by the syntax for *Other_Loop*. If the second meaning is intended, put parentheses around the **loop** keyword even though it has no property list:

```
dynamic call in (loop) calls “foo” ...
```

5.8 Calls

Call blocks and populations

A call block describes a set of subprogram calls and applies assertion clauses to all of these calls:

Call_Block → Population Call_Description {Clause} **end call**

If the *Call_Block* occurs within a *Sub_Block*, the call block and its assertion clauses apply to the described calls in this subprogram only. If the *Call_Block* occurs as a *Global_Bound*, it applies to the described calls in any analysed subprogram.

Population → [**all**] [Bound]

The *Population* part has the same syntax and meaning as for a loop-block population: it defines how many calls we expect to match the call-description, in each subprogram to which this *Call_Block* applies. An empty *Population* is the same as "**all** 1", that is we expect exactly one matching call. If the keyword **all** appears without the *Bound* part, any number (zero or more) of calls can match. If the *Bound* part is included (with or without **all**) it defines the allowed range for the number of matching calls.

If the number of matching calls in the subprogram under analysis violates the *Population* range, Bound-T emits an error message.

Call descriptions and call properties

Calls are identified by their properties or by their source-code position. The most important property is the callee subprogram, when this is statically known, that is, when the instruction sequence that implements the call specifies the address of the callee statically. For calls where the callee is specified dynamically (computed address, function pointer) the call cannot be identified by its callee(s). However, the property of being a dynamic call can be used as identification.

Call descriptions thus have two forms, for static and dynamic calls respectively. In both cases the same kind of additional call-properties can be specified:

Call_Description	→	Some_Call [Call_Properties]
Some_Call	→	Static_Call | Dynamic_Call
Static_Call	→	**call** [**to**] Sub_Name
Dynamic_Call	→	**dynamic call**
Call_Properties	→	Call_Property [**and** Call_Properties]

The callee subprogram is either statically known (the *Sub_Name* of a *Static_Call*) or is computed in some dynamic way, for example by use of a function-pointer variable (*Dynamic_Call*).

If the call description contains no *Call_Properties* any call to the subprogram identified by the *Sub_Name* (for a *Static_Call*) or any dynamic call (for a *Dynamic_Call*) matches the description. If some *Call_Properties* are listed, a match in addition requires that all the listed properties are true for this call.

Call_Property → Property_Prefix Basic_Call_Property

The construct *Property_Prefix* was already defined in section 5.7 above but we repeat it here for your convenience:

Property_Prefix	→	[**that**] [Do_Is] [{ **not** }]
Do_Is	→	**do** | **is**

The optional keywords **that**, **do** (or **does**), and **is** (or **are**) have no logical meaning and are used only to make the text more grammatically pleasing. Each occurrence of the keyword **not** inverts the logical sense of the *Basic_Call_Property*. Thus, an even number of **not**s has no effect and any odd number of **not**s has the same effect as one **not**.

Call_Property	→	Source_Position
	|	**marked** *string* Source_Context
	|	**in** Source_File

| **in** Other_Loop
| **uses** Variable_Name
| **defines** Variable_Name

Table 4 below defines the meaning of each type of call property.

Table 4: Meaning of call properties

Property	Call *C* has this property if and only if:
Source_Position	The call instruction for *C* is connected (through the compiler-generated mapping) to the source-code line identified by the *Source_Position*, within the precision allowed by the fuzz of the *Source_Position*.
marked *string* Source_Context	Same as for the *Source_Position* consisting of "**on marker** *string Source_Context* ".
in Source_File	The call instruction for *C* is connected to a source-code line in the file named by the *Source_File*.
in Other_Loop	*C* is contained in a loop that matches the *Other_Loop* description. Note that this loop is not necessarily the innermost loop that contains *C*.
uses Variable_Name	Not implemented. Has no effect.
defines Variable_Name	Not implemented. Has no effect.

5.9 Instructions

Instruction blocks

An instruction block describes a single machine instruction (whatever that means for the given target processor) and applies assertion clauses at this instruction:

Instruction_Block → **instruction** [**at**] Address_Or_Offset {Clause} **end instruction**

If the *Instruction_Block* occurs within a *Sub_Block*, the instruction block and its assertion clauses apply to the analysis of this subprogram only, even if other subprograms also contain this instruction. If the *Instruction_Block* occurs as a *Global_Bound*, it applies to the analysis of any and all subprograms that contain this instruction.

Address_Or_Offset → *code-address* | **offset** *code-offset*

If the *Instruction_Block* occurs as a *Global_Bound* the instruction must be identified by an absolute address (*code-address*); an **offset** cannot be used.

Instructions in control-flow graphs

When Bound-T analyses a subprogram, it actually analyses an internal model of the subprogram. This internal model is a control-flow graph decorated with information about the instructions, branch conditions, and possible execution states. A node in the control-flow graph represents a sequence of consecutively executed machine instructions (a basic block) and is further divided into a list of "steps" where each step typically represents one instruction and is associated with the machine address of that instruction. However, in some cases one and the same instruction can be represented by two or more steps in the same control-flow graph; this happens if the instruction is modelled in different ways depending on its execution context.

At present, Bound-T accepts an *Instruction_Block* assertion only when the instruction's address maps to exactly one step (and thus exactly one basic block) in the control-flow graph of the subprogram under analysis.

Clauses in instruction blocks

At present, two forms of assertion clause are accepted in an instruction block: the *Repetition_Bound* clause and the *Role_Bound* clause. The former asserts how many times the instruction can be executed; the latter asserts the role that the instruction performs when it is executed. The role is typically some kind of dynamic transfer of control: branch, call, or return.

5.10 Clauses and facts

Fact clauses

The actual facts that are claimed to hold in some context (globally or locally in a subprogram, loop or call) are collected into the following production:

```
Clause  →  Execution_Time_Bound ;
        |  Stack_Bound ;
        |  Start_Bound ;
        |  Repetition_Bound ;
        |  Variable_Bound ;
        |  Property_Bound ;
        |  Variable_Invariance ;
        |  Callee_Bound ;
        |  Role_Bound ;
```

Note, however, that some fact clauses are not allowed in some contexts as discussed further below.

Allowed combinations of fact and context

An assertion states a specific *fact* in a specific *context*, as explained in chapter 2. The '+' entries in the following table show which combinations of fact and context are allowed. The meaning of each combination is explained in the subsection dedicated to the fact. For information, the table includes as its last row the volatility mark or fact, although it is not really a *Clause* in the assertion syntax.

Table 5: Fact and context combinations

Asserted fact	Global	Subprogram entry	Subprogram body	Loop	Static call	Dynamic call	Instruction
Variable bound	+	+	+	+	+	+	
Property bound	+		+	+	+ (no effect)	+ (no effect)	
Variable invariance			+	+	+	+	
Repetition bound				+	+	+	+
Execution time bound			+		+	+	
Start bound				+			
Stack bound			+				
Callee bound						+	
Role bound							+
Volatility mark	+						

Unsupported combinations of fact and context

Several combinations in the above table are marked as not allowed (blank grey). Here is some rationale for this.

Global assertions can be given only for variable and property values. A global assertion of variable invariance, repetition count or execution time would have no meaning because there is nothing to which the assertion could apply.

Property assertions are not allowed in a subprogram entry context because this context does not contain any instructions that could be affected by the properties. Further, property assertions have no effect in a call context, but this may well change in future versions of Bound-T.

It is not possible to specify a repetition count for a particular subprogram. While such an assertion on the total number of times the subprogram is executed would be quite reasonable and could be useful, the current design of Bound-T cannot support it (because Bound-T finds the worst-case path within each subprogram separately, not within the program as a whole). Instead, the user can assert a separate limit on the repetition count for each call of this subprogram, in the context of this call.

It is not possible to assert the execution time of a loop. There is no technical obstacle that would prevent this but the benefit seems small while the implementation effort would be non-trivial.

Stack bounds could logically be asserted for particular calls as well as for subprograms. This ability will no doubt be added in a future version of Bound-T.

The set of possible callees (*Callee_Bound*) is obviously relevant only to dynamic calls and cannot be asserted in any other context.

At present, only the repetition count and role can be asserted in an instruction context. Assertions on variable and property values at instructions will probably be allowed in future versions of Bound-T.

Volatility of variables could reasonably be asserted in a local context, but is not yet implemented.

5.11 Execution-time bounds

Bounds on the execution time of a subprogram or a call are written as follows:

Execution_Time_Bound → **time** Bound Time_Unit

Time_Unit → **cycles**

The *Execution_Time_Bound* clause can be used in a subprogram context as a *Clause* in a *Sub_Block*, or in a call context as a *Clause* in a *Call_Block*. The following table explains the meaning of asserting the execution time in each context where such an assertion is allowed.

Table 6: Meaning of execution time assertion

Context	Assertion applies to
Subprogram	The execution time of any one call of this subprogram except when another execution time is asserted for a specific call.
Call	The execution time for any execution of this call.

To elaborate:

- In a *subprogram context*, the assertion defines the WCET of the subprogram in processor cycles. Bound-T will not analyze the subprogram but will instead create a synthetic "stub" control-flow graph (typically containing one or two nodes) that "consumes" the given amount of execution time. Every call of this subprogram will be assigned this WCET unless another WCET is asserted specifically for some calls.
- In a *call context*, the assertion defines the WCET for these particular calls. Bound-T will still analyze the callee subprogram (unless a WCET is asserted in the context of this subprogram) and try to find WCET bounds to be used for all calls of this subprogram that do not have an asserted WCET.

Thus, if you want to omit a subprogram from the execution-time analysis, it is not enough to assert a WCET for every call of the subprogram; you must assert a WCET for the whole subprogram or assert **omit** for the subprogram, and then you can assert other WCET values for specific calls if you wish.

An *Execution_Time_Bound* must use the keyword **time**, not **times**. The two forms of this keyword are not equivalent in this construct.

5.12 Stack bounds

Bounds on the usage or final height of a particular stack, for a particular subprogram (that is, within a subprogram block), are written as follows:

Stack_Bound → **stack** [Stack_Name] { Stack_Value }

Stack_Name → *string*

Stack_Value → **usage** Bound
| **final** Bound

The Stack_*Bound* clause can be used only in a subprogram context as a *Clause* in a *Sub_Block*. (In the future, it may also be allowed in a call context in a *Call_Block*.)

The *Stack_Name* is optional if the program under analysis contains only one stack. If the program contains several stacks you must identify the stack in question with its *Stack_Name*. The stacks and their names are described in the Application Notes for your target processor and cross-compiler. For programs with no stacks a *Stack_Bound* assertion results in an error message.

A *Stack_Value* sets bounds on the stack **usage** or on the **final** stack height, according to the keyword used. The same *Stack_Bound* clause can contain a usage bound and a final-height bound, in either order. (A warning is emitted if it contains more than one *Stack_Value* of the same kind.)

The lower bound for stack usage should be non-negative; a warning results if a negative lower bound is given, and the effective lower bound is then zero.

For a stable stack, which by definition has a final height of zero, it is never necessary to assert a final height, and it is an error to assert bounds that allow a non-zero final height.

If you assert the usage of all stacks for a subprogram, that subprogram is excluded from the stack analysis (but may still be analysed for execution time).

5.13 Repetition bounds

Bounds on the number of repetitions of a loop or the number of executions of a call or an instruction are written as follows:

Repetition_Bound → **repeats** Bound **times**

The *Repetition_Bound* clause can be used in a loop context as a *Clause* in a *Loop_Block*, or in a call context or instruction context as a *Clause* in a *Call_Block* or an *Instruction_Block*. When the clause appears in a call or instruction context we sometimes call it "execution count" bounds instead of "repetition" bounds. The following table explains the meaning of asserting the repetition (or execution) count in each context where such an assertion is allowed.

Table 7: Meaning of repetition count assertion

Context	Assertion applies to
Loop	The number of times the loop-body can be executed for each activation of the loop.
Call	The number of times the call can be executed for each activation of the containing subprogram (the caller).
Instruction	The number of times the instruction can be executed for each activation of the subprogram that contains the instruction. In other words, referring to the control-flow graph of the containing subprogram, the number of executions of the basic block that contains the step that represents the instruction. Note that this assertion is accepted only when the subprogram contains exactly one such step.

The rest of this subsection explains the meaning more precisely, especially for loops.

Repetition bounds for calls and instructions

When a repetition bound applies to a call or an instruction, it constrains the worst-case execution path of the containing subprogram so that the number of executions of the call or instruction is bounded by the *Bound*. Note that both the lower and upper bounds of *Bound* are used.

Note that *increasing* the execution count of a call or instruction can *decrease* the overall execution time, since forcing the execution to pass more often through this call or instruction may allow it to pass less often through other statements that would use more execution time. As an example, consider the following Ada pseudo-code:

```
for N in 1 .. 50 loop
   if Simple (N) then
      Quick (N);
```

```
        else
            <long computation>;
        end if;
    end loop;
```

If the WCET bound of the long computation in the else-branch is larger than that of procedure *Quick*, and in the absence of any assertions, Bound-T will assume as the worst case that the else-branch is taken on each iteration, so 50 times. If you assert that *Quick* is called at least 10 times, Bound-T is forced to assume that the else-branch is taken only 40 times, thus reducing the overall WCET bound because 10 calls of *Quick* are faster than 10 executions of the else-branch.

Repetition bounds for loops

To define the precise meaning of a *Repetition_Bound* for a loop we must first define some terms related to loops in flow-graphs.

In Bound-T the nodes in the flow-graph are the "basic blocks" of the machine instructions in the subprogram. A basic block is a maximal sequence of instructions such that the flow of execution enters this sequence only at the first instruction and leaves only at the last instruction. Thus, all instructions in the sequence have one successor (except for the last instruction which may have several or none) and one predecessor (except for the first instruction which may have several or none). The edges in the flow-graph of course represent the flow of execution between the basic blocks.

Loops correspond to cyclic paths in the flow-graph. Bound-T currently requires that the structure of the flow-graph be *reducible*, which means that two loops are either completely separate (share no nodes or edges) or one is completely nested within the other.

Reducibility also means that each loop has a distinguished node called the *loop head* with the property that the loop can be entered *only* through the loop head. On the source-code level, the loop head is analogous to the "for" or "while" syntax that introduces the loop; reducibility forbids jumps from outside the loop into the loop body, "around" the loop head.

Figure 8 below illustrates a loop in a flow-graph, including the loop head and the following other terms:

- The *loop body* is the set of all nodes that lie on some cyclic path from the loop-head back to the loop head. The loop body thus includes the loop head itself.
- A *start edge* is any edge from outside the loop body into the loop body. A start edge must lead to the loop head because that is the only point of entry to the loop.
- The *start nodes* are the source nodes of the start edges. They are not part of the loop body. They often contain the instructions that initialize the loop variables, including the loop counters if the loop has counters. The start nodes have no role in the meaning of a Repetition_Bound but are important for identifying a loop through its source-code position as explained in section 3.3.
- A *neck edge* is any edge from the loop head to a node in the loop body. It can lead to some other node in the loop body or directly back to the loop head itself.
- A *repeat edge* is any edge from the loop body to the loop head. Repeat edges are also known as "back edges". An edge from the loop head to itself is both a repeat edge and a neck edge.
- An *exit edge* is any edge from the loop body to a node outside the loop body.

A loop is called an *exit-at-end* loop if, for any exit edge, all the edges with the same source node are either exit edges or repeat edges (in the same loop). The example loop in the figure above is not an exit-at-end loop because the exit edge from node 3 to node 6 violates this

condition; the edge from node 3 to node 5 has the same source node (3) but is neither an exit edge nor a repeat edge. If either of these edges were removed the loop would become an exit-at-end loop.

We say that a loop is a (syntactically) *eternal* loop if it has no exit edges or if all exit edges are known to be infeasible. We consider such loops to also be exit-at-end loops.

In the source code, an exit-at-end loop is often a "bottom-test" loop. However, compilers can turn top-test loops into exit-at-end loops by coding the first instance of the loop termination test as a special case that is not within the loop body.

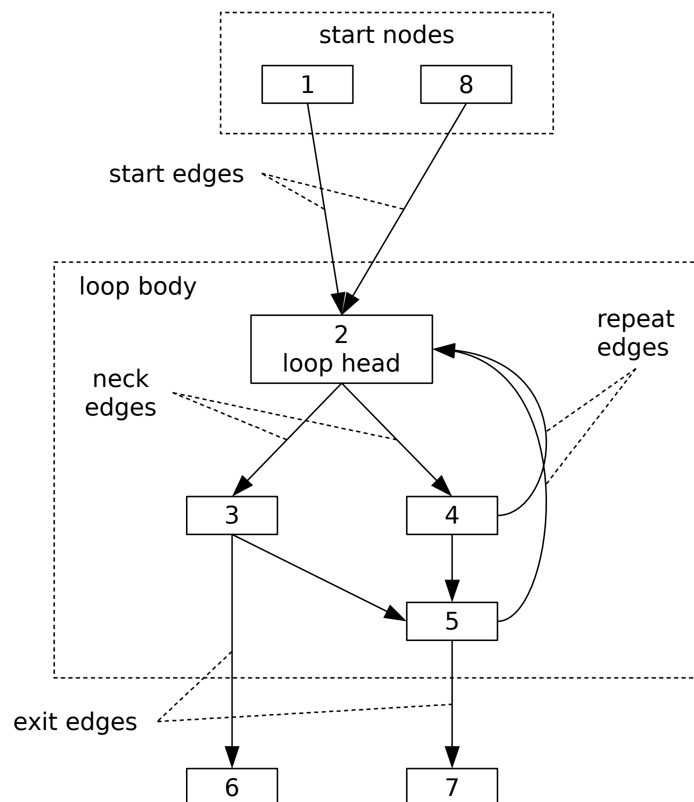

Figure 8: A loop in a flow-graph

The loop head is node 2; the loop body consists of nodes 2, 3, 4, and 5; the start nodes are nodes 1 and 8; the start edges are those from nodes 1 and 8 to node 2; the repeat edges are those from nodes 4 and 5 to node 2; and the exit edges are those from node 3 to node 6 and from node 5 to node 7.

We can now define the meaning of a repetition bound for a loop:

- When a repetition bound with the number R as the upper *Bound* applies to a loop that is not an exit-at-end loop it constrains the worst-case execution path of the containing subprogram as follows. If the start edges are executed a total of A times, then the neck edges are executed in total at most $R \times A$ times. Note that the loop-head can be executed up to $(R + 1) \times A$ times, because each of the $R \times A$ executions of the loop-body may jump back to the loop-head along a repeat edge.
- When a repetition bound with the number R as the upper *Bound* applies to an exit-at-end loop it constrains the worst-case execution path as follows. If the start edges are executed a total of A times, then the repeat edges are executed in total at most $(R - 1) \times A$ times.

The meaning of the lower bound of the repetition *Bound* is analogous.

Although not mentioned in the definition above, the practical effect of a repetition bound also depends on whether there are exit edges from the loop head. While-loops and other “top-test” loops often have exit edges from the loop head.

Figure 9 below shows an example of the most general form of a loop with exit edges both from the loop head and from the loop body. The nodes and edges in the flow-graph are labelled with execution counts assuming an assertion that the loop repeats $R = 6$ times and one start of the loop, $A = 1$. (For a larger number of starts the execution counts are multiplied proportionately.) The execution counts 4 and 2 for the alternative internal paths (the neck edges in this case) are examples; any two numbers that add up to 6 are possible in the absence of other assertions or knowledge. The worst-case path (again in the absence of other constraints) executes the repeat edge 6 times and the loop-head 7 times.

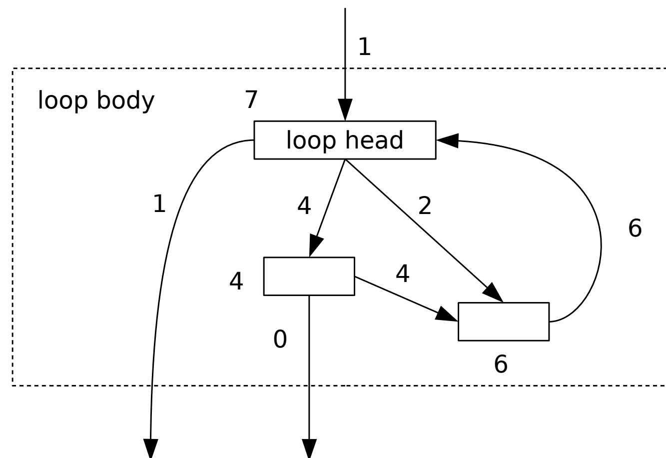

Figure 9: A general kind of loop asserted to repeat 6 times

A loop that is not an exit-at-end loop and has no exit edges from the loop head can be called a “middle-exit” loop. Figure 10 below shows a middle-exit loop after asserting that the loop repeats $R = 6$ times and assuming that the loop is started once, $A = 1$. As above, the execution counts 4 and 2 for the alternative internal paths are examples. Note that the node in the loop body from which the repeat edge originates executes only 5 times. In real code, this node might hold most of the code in the loop; it is then questionable if the assertion has the intended effect, or if asserting 7 repetitions would be more suitable, giving 6 executions of this node.

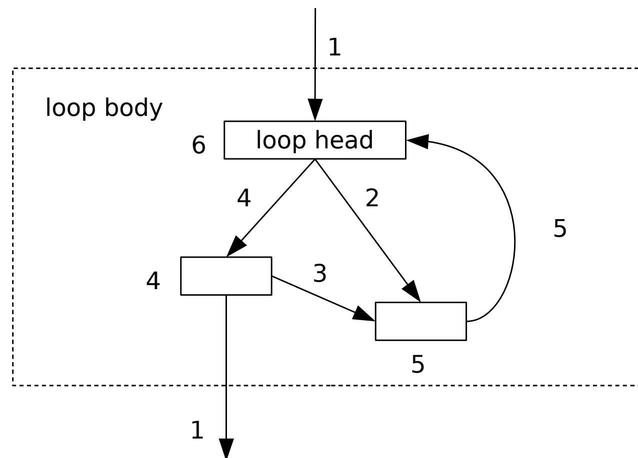

Figure 10: A middle-exit loop asserted to repeat 6 times

Figure 11 below shows an exit-at-end loop after asserting that the loop repeats R = 6 times and assuming that the loop is started once, A = 1. Note that the node from which the repeat edge originates now executes 6 times, equal to the asserted number of repetitions.

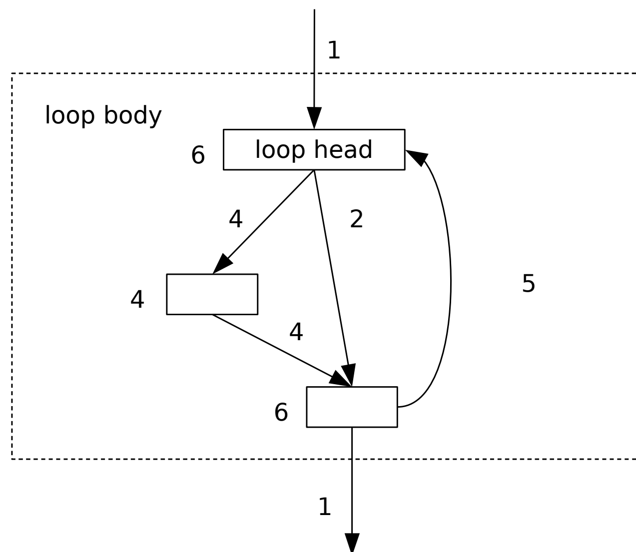

Figure 11: An exit-at-end loop asserted to repeat 6 times

Which repetition bound is right?

As the examples above show, the "right" value for a loop-repetition bound depends on the form of the flow-graph, in particular on where the "important" parts of the loop lie with respect to the loop-head and exit edges. Unfortunately there is no sure way to deduce the form of the machine-code flow-graph from the source code of the loop. For small target processors the evaluation of a simple condition may need several instructions and conditional jumps; consider, for example, the comparison of two 16-bit integers on an 8-bit processor. This means

that a while-loop with such a condition probably will not have an exit edge from the loop head because the loop head node contains only the first part of the instruction sequence that evaluates the condition.

You should therefore ask Bound-T to draw the flow-graphs of subprograms with asserted loop repetition bounds and check that the execution counts agree with your intention. If they disagree, you should either adjust the repetition bound or use other kinds of assertions, for example on the execution count of calls.

Asserting zero repetitions

Asserting a zero number of repetitions may have an unexpected effect for loops that have no exit edge from the loop-head node. This happens in many exit-at-end loops and all syntactically eternal loops. Consider the following Ada pseudo-code:

```
loop
   if some condition then
      do something;
   end if;
   exit when done enough;
end loop;
```

The compiler very likely codes this as a loop-head that evaluates “some condition” and with the only exit edge at the end of the loop after evaluating “done enough”. For such a loop, the loop body is executed at least once, if the execution reaches this loop at all. If you assert zero repetitions for this loop, Bound-T considers the whole loop unreachable which might not be what you wanted.

Combining loop repetitions and call or instruction repetitions

When a call, or a specific instruction, is in a loop, a bound on the number of executions of the call or the instruction may *implicitly* bound the number of loop repetitions. However, Bound-T still requires an *explicit* bound on each loop in the subprogram before it tries to compute the WCET of the subprogram, unless you use an **enough for time** assertion to tell Bound-T that the implicit bounds are strong enough and it should try to compute the WCET even if some loops lack explicit bounds.

The explicit loop-bounds can be computed automatically or asserted. The worst-case path computation will then consider the conjunction of the implicit bounds (number of executions of the call, or the specific instruction) and the explicit bounds (number of loop repetitions). The WCET value will reflect the strictest bounds.

For example, assume that the target subprogram *Foo* has a loop that calls two subprograms *Lift* and *Drop* and is of the following form:

```
while <complex condition> loop
   if Need_To_Lift then
      Lift;
   else
      Drop;
   end if;
end loop;
```

Assume further that Bound-T cannot bound the loop iteration automatically (because the loop-condition is complex) and that we assert the number of executions of the two calls as follows:

```
subprogram "Foo"
   call to "Lift" repeats 10 times; end call;
   call to "Drop" repeats 15 times; end call;
end "Foo";
```

Since every loop iteration calls either *Lift* or *Drop*, these assertions imply that the loop can be executed at most (or in fact exactly) 25 times. However, Bound-T does not detect this implication and refuses to compute a WCET in the absence of more assertions. There are two ways to proceed. One way is to assert an explicit loop-bound, for example by adding to the above subprogram block the *Clause*

```
   loop repeats <= 40 times; end loop;
```

Under these assertions, Bound-T computes a worst-case path that executes the loop 25 times so that *Lift* is called 10 times and *Drop* is called 15 times, which also satisfies the explicit loop-assertion of no more than 40 repetitions. The other way is to add the *Sub_Option* assertion

```
   enough for time;
```

This assertion makes Bound-T compute the same worst-case path (25 executions of the loop, 10 of *Lift* and 15 of *Drop*) even if the loop is not explicitly bounded.

5.14 Start bounds

Bounds on the number of times a particular loop starts are written as follows:

Start_Bound → **starts** Bound **times**

The *Start_Bound* clause can be used only in a loop context as a *Clause* in a *Loop_Block*. The clause bounds the number of executions of the loop's start edges (see Figure 8 in section 5.13) for each execution of the subprogram that contains the loop. If there are several start edges, the bound is on the total number of executions of these edges. For example, if the loop has two start edges and you assert **starts** 2 **times** then both start edges may execute once each, or one edge may execute twice and the other edge not at all.

Note that the compiler may change the structure of a loop in ways that change the actual meaning of a *Start_Bound* as explained in section 2.4.

5.15 Variable bounds

Bounds on the value of a variable are written as follows:

Variable_Bound → **variable** Variable_Name Bound

The *Variable_Bound* clause states the possible range of the values of a variable and can be applied to any kind of context. However, the meaning is different for call contexts and subprogram-parameter contexts than for other contexts. The following table explains the meaning in each context where such an assertion is allowed.

Table 8: Meaning of variable value assertion

Context	Assertion holds:
Globally	During the entire analysed execution at every reached point.
Subprogram entry	For any execution of this subprogram, but only at the entry point, before executing the first instruction of the subprogram.

Context	Assertion holds:
Subprogram body	For any execution of this subprogram and at all points in the subprogram.
Loop	For any execution of this loop and at all points in the loop.
Call	For any execution of this call, immediately before entering the callee.

The rest of this section discusses each context in more detail.

Variable bounds for subprogram bodies, loops or globally

When variable value bounds are asserted for any context other than a call or subprogram entry, they apply *throughout* the *whole* context. This makes the assertion powerful but also means that you can easily create contradictions if you specify too narrow a range in the *Bound*. For example, assume that the context is a subprogram that contains two assignments to the variable *V*:

```
procedure Foo is
begin
   V := 3;
   some statements, not changing V;
   V := V + 1;
   further statements;
end Foo;
```

If you now assert for that *V* is 3 for this subprogram:

```
subprogram "Foo"
   variable "V" 3;
end "Foo";
```

then Bound-T may find that the further statements after the second assignment to *V* are unreachable because there *V* would have the value 4, which is forbidden by the assertion. This will often result in warning message. You should instead assert that *V* is in the range 3 .. 4 or put the assertion in the subprogram entry context as shown below.

Variable bounds on subprogram entry

When variable value bounds are asserted in a subprogram entry context (within parentheses following the first *Sub_Name* in a *Sub_Block*) they apply at one specific point in the program: immediately before the execution of the first instruction in the subprogram (the entry point of the subprogram).

The following asserts that the variable *V* has the value 3 on entry to the subprogram *Foo*:

```
subprogram "Foo" (variable "V" 3)
end "Foo";
```

Since the assertion applies only on entry to the subprogram, the subprogram can change *V* in any way without contradicting this assertion. However, if *Foo* is called again, the assertion must again hold (V must equal 3) on the new entry to *Foo*.

Variable bounds for calls

When variable value bounds are asserted for a call, they apply to the variables as visible in the caller, immediately before the execution flows from the caller to the entry point of the callee.

When the named variables occur in the call's actual parameter expressions, the parameter-passing mechanism of the call translates the asserted bounds on the caller's variables into bounds on the callee's (formal) parameters.

Taking into account the parameter-passing mechanism is especially important for target processors that rename registers during a call instruction. One example is the SPARC architecture with its "register windows". In the SPARC version of Bound-T, the variable value fact

```
variable address "o3" 123;
```

refers to output register 3. However, the register-window mechanism means that the physical register that the caller refers to as "o3" is visible in the callee as "i3" (input register 3), while "o3" in the callee refers to a different physical register. Thus, if the above assertion on "o3" is given in a call context, it has the same effect as the corresponding assertion on "i3" in the callee's entry context.

Please refer also to section 2.12 where you will find a warning on the use of "foreign" local variables in assertions.

Some compilers are sloppy with the mapping of variable names to registers, in particular at calls. They may use a register to pass a parameter to the callee although the symbol-table in the target program allocates this register to a variable that has nothing to do with this parameter. Bound-T cannot detect such symbol-table flaws which means that assertions on this variable may not have the correct effect. Check the Application Note for your target and compiler for advice.

When a *Variable_Bound* applies to a global variable (a statically addressed memory location) it is not affected by the parameter-passing mechanism and the asserted bounds apply to the same global variable for the callee.

Whether they concern parameters or global variables, the variable bounds asserted for a call are applied only to the *entry point* of the callee, not throughout the callee's code.

Variable value bounds for a call are currently used only for the call-path-specific analysis of the callee at this call. The variable bounds are not used in the subsequent analysis of the caller, although it would be reasonable, and future versions of Bound-T will probably do it.

5.16 Variable invariance

The invariance (unchanged value) of a variable is asserted as follows:

Variable_Invariance → **invariant** Variable_Name

A *Variable_Invariance* clause asserts that the named variable retains its value over any execution of the context to which the clause applies. This kind of assertion is not often used, but in some cases it can help Bound-T complete its arithmetic analysis and find loop-bounds automatically, as section 2.13 explained.

The following table explains the meaning of asserting the invariance of a variable in each context where such an assertion is allowed.

Table 9: Meaning of variable invariance assertion

Context	Assertion holds:
Loop	For any repetition of this loop and means that the variable has the same value whenever execution enters the loop head. The loop body may change the variable but must restore its value before going back to the loop head. However, the value may change on the last execution of the loop body, when the loop terminates.
Call	For any execution of this call and means that the execution of the call and the callee do not modify the variable, that is, the value on return from the call is the same as the value before the call. However, the variable can be modified within the callee as long as its original value is restored on return.
Subprogram body	For any loop and call in the subprogram. The subprogram itself may change the variable's value.

One consequence of asserting the invariance of a variable in a loop is that the variable cannot be a counter for the loop.

An invariance assertion in a context does not mean that the variable always has the same value when execution *reaches* the context. For example, if a variable is asserted as invariant for a call, the variable may have the value 5 on the first execution of the call and the value 207 on the second execution of the call. The invariance means only that the variable still has the value 5 after the first execution of the call and still has the value 207 after the second execution of the call.

Likewise, a variable that is asserted as invariant in a loop may have a different value each time execution reaches the loop from outside the loop, that is, each time the loop *starts*. For example, assume that the variable has the value 11 when the loop is first started and that the loop repeats five times before terminating. The loop head is thus executed six times. The invariance means that the variable still has the value 11 on each of these six entries to the loop head. When the loop terminates the variable may have a different value. If the program starts the loop again, the variable may have yet another different, for example 31; if the loop now repeats twice so that the loop head is executed three times the invariance means that the variable has the value 31 on each of these three entries to the loop head. Again, the value may change when the loop terminates.

Asserting invariance in a subprogram (body) context is equivalent to asserting invariance in all loops and calls in the subprogram. Note that it does *not* mean that the variable is invariant over a call of this subprogram.

5.17 Volatility Marks

The volatility of some variables and/or memory address ranges is asserted as follows:

Volatility_Mark → **volatile** Volatile_Area [{ **,** Volatile_Area }]

Volatile_Area → Variable_Name
| **range** *variable_address* **..** *variable_address*

When a *Variable_Name* is asserted as volatile, then any access to the Bound-T "storage cell" which models this variable is analysed as a volatile access. This does *not* include memory accesses that have a different address and/or width, and therefore map to a different storage cell, but which *overlap* the cell that is asserted as volatile. This is arguably a defect and should be corrected, but is a consequence of the generally weak analysis of aliasing and overlapping in Bound-T.

When a **range** of addresses is asserted as volatile, then any storage cell that overlaps this range is analysed as a volatile cell.

Target-specific rules define which *variable_address* values can be used in a volatile range assertion and which pairs of such values can be combined into an address range. For example, although the *variable_address* syntax for a target usually includes syntax for naming registers, it may not make sense to specify an address range that extends from a register to a location in RAM. Similarly, for processors with several different address spaces such as the Intel-8051, you probably will not be allowed to define an address range which starts in one space and ends in another.

Example

Assume a memory that is addressed by octet but can be read or written one octet at a time or one 4-octet word (32 bits) at a time, even with unaligned addresses. Assume that the 32-bit variable *x* is statically allocated at address 0x40, and therefore occupies the octets at 0x40, 0x41, 0x42, and 0x43. Assume also that the program under analysis has an instruction which reads *x* as a word from address 0x40, an instruction which reads one octet from 0x41, and an instruction which reads one word from 0x42.

If the variable *x* is asserted volatile, only the first instruction, which reads all and only *x*, is analysed as a volatile access. The other two instructions access different storage cells (in the Bound-T model) and are therefore not considered volatile.

If the address range 0x40 .. x043 is asserted as volatile, all three instructions are analysed as volatile accesses because all the corresponding storage cells overlap the volatile address range.

5.18 Property bounds

Bounds on the value of a target-specific "property" are written as follows:

Property_Bound → **property** Property_Name Bound

Property_Name → *string*

A *Property_Bound* clause asserts that some target-specific *property* of the target processor or of the target program under analysis has a given value or a given range of values throughout the context to which the clause applies. Since the properties are completely target-specific, please refer to the relevant target Application Notes for a list of the available properties and their meaning.

A typical use for such properties is to define the number of memory wait-states that should be assumed for specific types of memory accesses in this specific context. For example, boot code that executes from a narrow PROM may need a much larger number of program-memory wait-states than application code that executes from fast RAM memory with a wide instruction bus.

The following table defines the meaning of assertions on property values, in each context where such an assertion is allowed.

Table 10: Meaning of property value assertion

Context	Assertion holds:
Globally	During the entire analysed execution at every reached point, unless overridden by an assertion on this property in another context.
Subprogram body	For any execution of this subprogram and at all points in the subprogram, unless overriden by an assertion on this property in a loop or call within this subprogram.
Loop	For any execution of this loop and at all points in the loop, unless overridden by an assertion on this property in an inner loop or a call within this loop.

Context	Assertion holds:
Call	This context is allowed by the assertion language but the assertion currently has no effect.

Note that an assertion on a property value in an inner context *overrides* any assertions on this property in outer contexts (only the innermost assertion holds). This is in contrast to assertions on variable values where such nested assertions are combined (all applicable assertions hold).

5.19 Callee bounds

When a call instruction uses a dynamically computed callee address, Bound-T is often unable to find the possible callee subprograms by analysis. In such cases you can list the possible callees as a *Callee_Bound* fact in the context of the dynamic call:

Callee_Bound → **calls** Sub_Name { **or** Sub_Name }

The dynamic call is then analysed as if it were a case statement with one branch for each listed callee (*Sub_Name*) contain a call of this callee.

5.20 Role bounds

Some instructions can perform several *roles* in the execution of a program. For example, consider a "return" instruction that pops a return address off the stack and transfers control to that address. Most "return" instructions perform just that role, of returning control from a callee subprogram to the caller subprogram. However, a compiler can also use a "return" instruction to perform some other kind of transfer of control. For example, a jump to a dynamically computed address can be performed by pushing the address value on the stack and executing a "return". Furthermore, a very similar instruction sequence can be used to perform a call to a dynamically identified subprogram.

To analyze such multi-role instructions Bound-T must decide which role the instruction performs. The automatic algorithms built into Bound-T can sometimes choose the wrong role, which may make the analysis fail. In such cases you can tell Bound-T which role to use by a *Role_Bound* fact in the context of an instruction block:

Role_Bound → **performs** [**a** | **an**] *instruction-role*

The instruction-role is written as a quoted string; the contents of the string identify one of the possible roles defined for the current target processor. For example, the following subprogram block and the nested instruction block assert that the instruction at offset "1" from the entry point of the subprogram "ICall" performs a "tail call":

```
subprogram "ICall"
   instruction at offset "1"
      performs a "tail call";
   end instruction;
end "ICall";
```

What the "tail call" role means (and what the offset "1" means) depends on the target processor chosen for the analysis.

5.21 Combining assertions

The assertion language lets you assert several facts that apply to the same aspect of the program's behaviour. For example, you can write several bounds on the value of the same variable in the same context, or in different contexts that intersect, such as an assertion on variable *V* in subprogram *Foo*, combined with an assertion on *V* in a loop nested in *Foo*. The table below explains how Bound-T combines or conjoins such assertions.

Table 11: Effect of multiple assertions on the same item

Asserted fact	Effect in same context	Effect in nested context
Variable value range	The effective range is the intersection of all the asserted ranges.	The effective range is the intersection of all the asserted ranges.
Property value range	The effective range is the intersection of all the asserted ranges.	The range for the inner context is used.
Variable invariance	Multiple assertions have the same effect as a single assertion.	The assertion for the inner context holds.
Loop start count	The effective range is the intersection of all the asserted ranges.	Not applicable. The number of starts of the outer loop has no meaning for the inner loop.
Loop repetition count	The effective range of repetitions is the intersection of all the asserted ranges.	Not applicable. The number of repetitions of the outer loop has no meaning for the inner loop.
Call execution count	The effective range is the intersection of all the asserted ranges.	Not applicable. One call cannot be nested within another, in the opinion of Bound-T.
Instruction execution count	The effective range is the intersection of all the asserted ranges.	Not applicable. One instruction cannot be nested within another, in the opinion of Bound-T.
Subprogram execution time	The effective WCET bound is the smallest asserted time.	Not applicable. The context of a subprogram is always the global context.
Subprogram stack usage	The effective range is the intersection of all the asserted ranges.	Not applicable. The context of a subprogram is always the global context.
Subprogram final stack height	The effective range is the intersection of all the asserted ranges.	Not applicable. The context of a subprogram is always the global context.
Callees of a dynamic call	The effective set of callees is the union of the asserted sets of callees.	Not applicable. One call cannot be nested within another, in the opinion of Bound-T.
Role of an instruction	It is an error to assert different roles for the same instruction.	Not applicable. One instruction cannot be nested within another, in the opinion of Bound-T.

Contradictory repetition counts

Multiple assertions that affect the same or nested program elements can lead to contradictions. For example, assume that subprogram *Initialize* contains a loop that on each iteration executes a call of the subprogram *Allocate_Block*, and that the following assertions are stated:

```
all loops that call "Allocate_Block"
    repeat <= 10 times;
end loops;

subprogram "Initialize"
    all calls to "Allocate_Block"
        repeat 20 times;
    end calls;
end "Initialize";
```

The second assertion requires the call to *Allocate_Block* to occur 20 times and so requires 20 repetitions of the loop, but the first assertion only allows 10 repetitions. When such a contradiction occurs, the WCET computation will fail with an error message saying “infeasible execution constraints”.

Contradictory start bounds

A start-count assertion for an inner loop can contradict a repetition assertion (or a computed repetition bound) for the outer loop. This happens if the inner loop is asserted to start more often than the outer loop repeats, or if an unconditional inner loop is asserted to start less often than the outer loop repeats. When such a contradiction occurs, the WCET computation will fail with an error message saying “infeasible execution constraints”.

Contradictory value bounds

When several assertions constrain the value of the same variable in some context, Bound-T uses all the constraints. If the constraints are contradictory, the context in question may appear infeasible (unreachable). The same can happen if the assertions conflict with value bounds that Bound-T has found through analysis.

Two kinds of contradictions between assertions may arise:

- directly conflicting assertions on the same variable in the same context, and
- indirect conflict between assertions on two or more variables that contradict a relationship between these variables that Bound-T has deduced from its analysis.

The next two subsections discuss these further.

Direct conflict between assertions on the same variable

Bound-T can usually detect and report a direct conflict when it collects all the assertions for the analysis of a subprogram in a certain context. For example, if there is a global assertion that variable *V* is in the range 1 .. 5, and for subprogram *Foo* an entry assertion that *V* has the value 7, Bound-T will detect this direct conflict and warn about “conflicting assertions on entry” to *Foo*. Moreover, Bound-T will also list all the assertions that it collected for this analysis, grouping them as follows:

```
Global assertions:
    1<=DM0<=5
Subprogram entry assertions:
    DM0=7
Subprogram body assertions:
    None.
Call assertions and computed bounds:
    None.
```

Target-dependent implicit bounds:
SH=1
ZSH=0

As you can see, the global assertion and the subprogram-entry assertion conflict. Here "DM0" is the machine-level name for the source-code variable *V*.

The group “Call assertions and computed bounds” includes the bounds on input parameters and global variables that Bound-T has computed from the calling context.

Indirect conflict between assertions and deduced relationships

An indirect conflict can occur, for example, when one assertion constrains the variable *X* to values less or equal to 20 and another constrains the variable *Y* to the range 18 .. 25. These assertions as such are compatible, but if they apply to a context that includes an instruction that assigns *X* the value $Y + 3$, a conflict arises because the new value of *X* would be in the range 21 .. 28, which contradicts the assertion on *X*.

Such relationships between the values of variables are the main result of Bound-T's arithmetic analysis. The analysis deduces relationships from arithmetic assignments (instructions that compute a value and store it in a variable) and from the logical conditions of conditional branches. For example, if the above assertions on *X* and *Y* apply to a part of the program that is entered only through a conditional branch with the condition $X = Y + 3$, any arithmetic analysis in this part of the program (for example, finding bounds on a loop here) will discover the conflict.

When an indirect conflict between assertions and deduced variable relationships occcurs Bound-T is usually unable to decide if the reason is in the assertions or in the logic of the target program itself. Bound-T classifies the relevant program part as unreachable (and warns about it, if the option *-warn reach* is in effect).

Note that Bound-T does not search for such conflicts systematically; it discovers them only if the relevant program part needs some analysis, for example loop analysis. Thus, an analysis with contradictory assertions can succeed without discovery of the conflicts, but the conflicts may be revealed in a later re-analysis with a changed target program if there is now a loop, for example, that is covered by the conflicting assertions and relationships.

6 TROUBLESHOOTING

6.1 Warning messages

Some problems in an assertion file can make the assertion parser in Bound-T issue a *Warning* message in the basic output format explained in the Bound-T Reference Manual. The following table lists all these warning messages in alphabetical order, ignoring punctuation characters and letter case. For each message, the table explains the problem in more detail. For some warning messages, the table may suggest possible reasons for the problem and specific solutions.

Warning messages from sources other than the assertion parser are listed in the Bound-T Reference Manual. Some of them may also be due to problems in the assertions, although the assertion parser may not detect the problem itself and therefore does not issue a warning itself. Some warnings described in the Reference Manual are also described here, because they may arise during assertion parsing.

Table 12: Warning messages

Warning message		Meaning and remedy
Lower bound on stack usage set to zero	*Reasons*	The lower bound given in a stack usage assertion is negative. This is silly because usage cannot be negative, so Bound-T instead uses zero for the lower bound.
	Action	Change the lower bound to a non-negative value.
Multiple assertions on stack final height	*Reasons*	The same *Stack_Bound* clause gives more than one **final** stack height bound for the same stack. This is either redundant or contradictory.
	Action	Use only one **final** bound.
Multiple assertions on stack usage	*Reasons*	The same *Stack_Bound* clause gives more than one stack **usage** bound for the same stack. This is either redundant or contradictory.
	Action	Use only one **usage** bound.
Negative counts are meaningless	*Reasons*	The *Count* specified in this *Loop_Property* for the number of "contained" calls or inner loops allows negative numbers. For example, "loop that contains -4 .. 6 loops". However, the number of contained calls or loops cannot be negative.
	Action	Correct the assertion to allow only non-negative numbers.
Negative numbers are meaningless here	*Reasons*	This *Repetition_Bound* allows negative numbers. For example, "repeats -4 .. 6 times". However, the number of repetitions cannot be negative.
	Action	Correct the assertion to allow only non-negative numbers.
Negative populations are meaningless	*Reasons*	The *Population* specified for this *Call_Block* or *Loop_Block* allows negative numbers. For example, "all -4 .. 6 calls". However, populations of calls or loops cannot be negative.
	Action	Correct the assertion to allow only non-negative numbers.
No valid assertions found in this file	*Reasons*	This assertion file seems to be empty, or to contain only comments, or only assertions that are in error.

Warning message		Meaning and remedy
	Action	Correct the erroneous assertions, if any. Check that assertions have not, by mistake, been commented out or removed.
Other identifier connection not used.	*Reasons*	An assertion uses a symbolic (source-code) name for a subprogram or statement-label, but this name is connected to more than one machine-code location (multiply defined). This message shows a connection that Bound-T did not use.
	Action	Add scope context to the name in the assertion, to make the name unambiguous. See section 3.2.
Overriding an "exactly" qualifier.	*Reasons*	The assertion identifies its context by a source-code position as **exactly** on a specific source-code line (number) or mark, which means that source-line numbers should be compared exactly, with no fuzz. However, the assertion also includes a **within** part that specifies the fuzz to be used when comparing line numbers. This overrides the **exactly** qualifier.
	Action	Correct the assertion by removing either the **exactly** keyword or the **within** part, as desired.
Subprogram not found: *S*	*Reasons*	The subprogram named for this subprogram block (after the **subprogram** keyword) was not found in the target program. The name may be mistyped, or the name may have been "mangled" by the compiler and linker.
	Action	Correct the assertion file use the subprogram name as it exists in the target program executable (the link-name). To find the possibly mangled name, use *-trace symbols* or dump the target program file.
This identifier connection used.	*Reasons*	An assertion uses a symbolic (source-code) name for a subprogram or statement-label, but this name is connected to more than one machine-code location (multiply defined). This message shows the connection that Bound-T uses.
	Action	Add scope context to the name in the assertion, to make the name unambiguous. See section 3.2.
Unlimited interval may have no effect.	*Reasons*	The *Bound* written in this assertion is unlimited; it places neither a lower nor an upper bound on the number in question.
	Action	Check that the assertion is what you intended. Correct if wrong. Remove the assertion if it has no effect.
Void interval may create contradiction	*Reasons*	The *Bound* written in this assertion allows no values at all because its lower bound is greater than the upper bound. For example, 5 .. 3.
	Action	Check that the assertion is what you intended. Correct if wrong.

6.2 Error messages

When the assertion parser in Bound-T finds an error in an assertion file it issues an *Error* message in the basic output format explained in the Bound-T Reference Manual. The following table lists all these error messages in alphabetical order, ignoring punctuation characters and

letter case. For each message, the table explains the problem in more detail. For some error messages, the table may suggest possible reasons for the error and specific solutions. Otherwise, the general reason is an error in the assertion file and the general solution is to correct the assertion file and re-run Bound-T.

Error messages from sources other than the assertion parser are listed in the Bound-T Reference Manual. Some of them may also be due to errors in the assertions. In those cases the assertions are syntactically and semantically correct, so the assertion parser accepts them, but later stages of the analysis find some conflict between different assertions or between the assertions and the target program under analysis. For example, the number of loops that actually match a *Loop_Description* may be different from the expected *Population* for this loop description.

Table 13: Assertion error messages

Error message		Meaning and remedy
".." expected, at "*T*"	*Problem*	At this point in the syntax, the double-dot separator ".." should appear instead of the token *T*.
"*S*" expected, at "*T*"	*Problem*	At the end of a subprogram block, for a subprogram identified by *S*, the subprogram identifier is repeated in the form *T* which does not match *S*. See also the error message that begins "Mismatch...".
"*A*" is not a valid cell address	*Problem*	An assertion contains the string *A* that is meant to denote a variable (storage cell) address, but is rejected by the assertion parser.
	Reasons	The string *A* is not written according to the rules for variable addresses that Bound-T uses for this target processor.
	Solution	Refer to the Application Note for this target and correct the string.
"*A*" is not a valid code address, at "*T*".	*Problem*	An assertion contains the string *A* that is meant to denote a code address, but is rejected by the assertion parser. The next token is *T*.
	Reasons	The string *A* is not written according to the rules for code addresses that Bound-T uses for this target processor.
	Solution	Refer to the Application Note for this target and correct the string.
"*A*" is not a valid code offset, at "*T*".	*Problem*	An assertion contains the string *A* that is meant to denote a code offset, but is rejected by the assertion parser. The next token is *T*.
	Reasons	The string *A* is not written according to the rules for code offsets that Bound-T uses for this target processor.
	Solution	Refer to the Application Note for this target and correct the string.
Ambiguous label name: *N* *or* Ambiguous subprogram name: *N* *or* Ambiguous variable name: *N*	*Problem*	The assertion tries to identify a label, subprogram, or variable by the name *N* but the name matches more than one label, subprogram, or variable (respectively) in the program and so is ambiguous.
	Reasons	The program contains more than one label, subprogram, or variable (respectively) with the name *N* but in different scopes. The name in the assertion does not specify the scope (well enough).
	Solution	Add scope levels to the name to make it unambiguous.

Error message		Meaning and remedy
Assertion expected, at "*T*".	*Problem*	The current token *T* cannot be the start of an assertion, as would be expected here. See the nonterminal *Assertions* in section 5.3 (page 80). The assertion parser silently skips the following text until it finds the start of the next assertion.
Assertion file contained errors.	*Problem*	Some errors were noted in the assertion file (and reported by the corresponding other error messages in this table). The analysis stops (after reading the rest of the assertion files, if any).
Assertion file could not be read.	*Problem*	The assertion file could not be opened because the user does not have read access to the file. The analysis stops (after reading the rest of the assertion files, if any).
Assertion file is not a text file.	*Problem*	The assertion file could not be read because it seems not to be a text file (it may be a directory or some other special kind of file). The analysis stops (after reading the rest of the assertion files, if any).
Assertion file was not found.	*Problem*	The assertion file could not be opened because it seems not to exist. The analysis stops (after reading the rest of the assertion files, if any).
"at" or "on" after "exactly" expected, at "*T*"	*Problem*	When a *Source_Position* starts with the keyword **exactly** it must continue with either of the keywords **at** or **on**, instead of the token *T*. See section 5.2 (page 79).
Base-point is in "*B*", not in "*F*".	*Problem*	This *Source_Point* defines the source-line number in the form **line offset** *offset* **in file** "*F*". However, the base-point, as defined by the containing subprogram, lies in a source-code file with the different name *B*. It is not possible to move from file *B* to file *F* by adding an *offset* to the base-point line number.
	Solution	Change the assertion so that the file-names are the same, or remove the part **in file** "*F*" if it is unnecessary.
Bound expected, at "*T*"	*Problem*	The assertion file should have a bound here, instead of the token *T*. See the nonterminal *Bound* in section 5.2 (page 78).
"call" after "end" expected, at "*T*"	*Problem*	The keyword **call** should here follow the keyword **end**. See the nonterminal *Call_Block* in section 5.8.
Call or loop description expected, at "*T*"	*Problem*	When describing a loop by what it contains, the key-word **contains** and the possible *Count* should be followed by a *Call_Description* or a *Loop_Description* or a *Source_Position*. The actual token *T* is none of these. See the nonterminal *Basic_Loop_Property* in section 5.7.
Call properties expected, at "*T*"	*Problem*	The assertion file should have a call property here, instead of the token *T*. See the nonterminal *Call_Property* in section 5.8.
Calls do not have the "defines" property.	*Problem*	The assertion file tries to use the **defines** keyword to identify a call. This property is not yet supported for calls.
Calls do not have the "uses" property.	*Problem*	The assertion file tries to use the **uses** keyword to identify a call. This property is not yet supported for calls.
Cannot assert callees for a static call.	*Problem*	The assertion file tries to assert the possible callees for a static call, which is nonsense. In other words, there is a *Callee_Bound* in a *Call_Block* for a static call.

Error message		Meaning and remedy
Clause expected, at "*T*"	*Problem*	The assertion file should have an assertion clause here, instead of the token *T*. See the nonterminal *Clause* in section 5.10.
Clause or "end call" expected, at "*T*"	*Problem*	The assertion file should have an assertion clause here, or the **end call** keywords, instead of the token *T*. See the nonterminals *Clause* in section 5.10 and *Call_Block* in section 5.8.
Clause or "end instruction" expected, at "*T*"	*Problem*	The assertion file should have an assertion clause here, or the **end instruction** keywords, instead of the token *T*. See the nonterminals *Clause* in section 5.10 and *Instruction_Block* in section 5.9.
Clause or "end loop" expected, at "*T*"	*Problem*	The assertion file should have an assertion clause here, or the **end loop** keywords, instead of the token *T*. See the nonterminals *Clause* in section 5.10 and *Loop_Block* in section 5.7.
Closing parenthesis after call or loop expected, at "*T*"	*Problem*	A call or loop description that is enclosed in parentheses should be closed by a ')', instead of the token *T*. See the nonterminals *Other_Call* and *Other_Loop* in section 5.7.
Closing parenthesis after parameters expected, at "*T*"	*Problem*	The assertions on subprogram parameters should be followed by a ')', instead of the token *T*. See the nonterminal *Sub_Block* in section 5.6.
Closing parenthesis expected, at "*T*"	*Problem*	This assertion describes a call or a loop as being contained in another loop which is described by its properties enclosed in parentheses. However, the closing parenthesis ')' seems to be missing; it was expected at this point but the token T was found instead. See the nonterminal *Other_Loop* in section 5.7.
Context defines no source file for line number.	*Problem*	This global assertion describes its context by a *Source_Position* that gives a source-line line number but does not have a *Source_File* part to give the name of the source-code file. A global assertion must also give the source-file name.
Context provides no base for code offset	*Problem*	1. A loop description uses the property "**executes offset** *code-offset*" but the subprogram that contains the loop is not specified so there is no base address for the offset. In other words, the description is in a *Loop_Block* that is a *Global_Bound*. 2. An instruction block identifies the instruction by an offset, but the subprogram that contains the instruction is not specified so there is no base address for the offset. In other words, the instruction block is a *Global_Bound*.
Context provides no base for line-number offset.	*Problem*	This global assertion describes its context by a *Source_Line* of the form **line offset**. However, the line **offset** form can be used only in a *Subprogram_Block* because the subprogram defines the base line-number to which the offset is added. See section 5.2 (page 79).
"cycles" expected, at "*T*"	*Problem*	This execution-time assertion should have the keyword **cycles**, instead of the token *T*, after the asserted execution time. It should be "**time** *N* **cycles**".

Error message		Meaning and remedy
Dynamic call cannot have “call to” *or* Dynamic call cannot have “call_to”.	*Problem*	The assertion describes a call that is **dynamic** but is also a **call to** a named callee, which is an impossible combination.
“end call” expected, at “*T*”	*Problem*	The present *Call_Block* should end with **end call**, instead of the token *T*.
“end loop” expected, at “*T*”	*Problem*	The present *Loop_Block* should end with **end loop**, instead of the token *T*.
“exactly”, “at”, “on”, “after” or “before” expected, at “*T*”	*Problem*	A *Source_Position* should start with one of the keywords listed in the message, instead of the token *T*. See section 5.2 (page 79).
Execution time bounds for a loop are not allowed.	*Problem*	The assertion file tries to assert the **time** for a loop. This assertion is not supported for loops, only for sub-programs and calls.
Execution time bounds for an instruction are not allowed.	*Problem*	The assertion file tries to assert the **time** for an instruction. This assertion is not supported for instructions, only for subprograms and calls.
Execution time bounds for the program are not allowed.	*Problem*	The assertion file tries to assert the **time** as a global fact. This assertion is not supported in the global context, only for subprograms and calls.
Execution time bounds not allowed in this context.	*Problem*	The assertion file tries to assert the **time** in a context where **time** facts are not supported.
Final height of stable stack must be zero: *S*	*Problem*	This *Stack_Bound* clause contains a **final** stack-height assertion that allows a non-zero value. This is a contradiction because this stack, named *S*, is known to Bound-T as a “stable” stack, where the final height is always zero.
	Reasons	Perhaps a change in the compilation options for the target program, or in the command-line options for Bound-T, have changed the stack to be classed as “stable” instead of “unstable”.
	Solution	If the stack really is stable, the assertion is redundant; remove it. If the stack should be classed as unstable, refer to the Application Notes for your target processor to understand why it is classed as stable.
“for” after “enough” expected, at “*T*”	*Problem*	The keyword **for** is expected after the keyword **enough**, to form the assertion **enough for time**. See section 2.17 and section 5.2 (page 83).
“instruction” after “end” expected, at “*T*”	*Problem*	The keyword **instruction** should here follow the keyword **end**, while the actual token is *T*. See the nonterminal *Instruction_Block* in section 5.9.
“instruction” expected, at “*T*”	*Problem*	At this point in the syntax, the keyword **instruction** should appear instead of the token *T*.
Instruction role "*R*" ("*L*") not recognized.	*Problem*	The instruction role-name *R* used in this *Role_Bound* assertion is not one of the role names defined for the present target processor, even in the "normalized" form *L*.
	Solution	The Application Note for this target defines the known role-names. Correct the assertion accordingly.
Integer value expected, at “*T*”	*Problem*	The assertion file should have an integer literal here, instead of the token *T*.

Error message		Meaning and remedy
Integrated analysis cannot be negated.	*Problem*	The assertion tries to negate (disable) integrated analysis of a subprogram; this is not possible.
	Solution	Non-integrated analysis is the default, so remove the assertion as redundant. See section 5.6.
"invariant" expected, at "*T*"	*Problem*	At this point in the syntax, the keyword **invariant** should appear instead of the token *T*.
Label not found: *L*	*Problem*	The assertion file names a label *L* but the target program's symbol table does not have a statement label named *L* in this scope.
	Reasons	The name *L* may be mistyped; if the default scope is used perhaps another scope should be named explicitly; or the target compiler may have mangled the names.
	Solution	Check for typos. Check the target program's symbol table using eg. *-trace symbols* or by dumping the file.
"line" or "marker" expected, at "*T*"	*Problem*	At this point in the *Source_Position* part of an assertion one of the keywords **line** or **marker** is expected (to start a *Source_Line* structure), instead of the token *T*.
"loop" after "end" expected, at "*T*"	*Problem*	The keyword **loop** should here follow the keyword **end**. See the nonterminal *Loop_Block* in section 5.7.
Loop description expected, at "*T*"	*Problem*	A loop description is expected here, instead of the token *T*. See the nonterminal *Loop_Description* in section 5.7.
Loop or call expected, at "*T*"	*Problem*	A loop-block or call-block is expected here, instead of the token *T*. See the nonterminals *Loop_Block* in section 5.7 and *Call_Block* in section 5.8.
Loop property expected, at "*T*"	*Problem*	The assertion file should here have a loop property, instead of the token *T*. See the nonterminals *Loop_Property* and *Loop_Properties* in section 5.7.
Mismatched subprogram identifier "*S*" after "end" *followed by* "*N*" expected, at "*T*"	*Problem*	The subprogram identifier *S* after the final **end** of a *Sub_Block* does not match the identifier *N* given at the start of the block. This message is followed by another error message that shows the expected identifier *N*.
Must include non-negative numbers : *range*	*Problem*	The *Bound* in this *Repetition_Bound* or *Start_Bound* allows only negative numbers, as shown in the *range*. The number of times a loop starts or repeats, or the number of times a call or instruction is executed, cannot be negative, so this assertion makes no sense.
"normally" or "[to] offset" expected, at "*T*"	*Problem*	In this *Sub_Option* the keyword **repeat** (or **repeats**) is followed by an unacceptable token *T*.
Numbers are too large : *min* .. *max*	*Problem*	The lower-bound *min* and/or upper-bound *max* in this *Repetition_Bound* are too large; Bound-T cannot handle so large execution counts.
	Solution	Use smaller numbers. Tell Tidorum about the problem.
"offset" expected, at "*T*"	*Problem*	In this *Sub_Option* the keywords **repeat** (or **repeats**) and **to** are not followed by the **offset** keyword, as required by the syntax for **return to offset**.
"performs" expected, at "*T*"	*Problem*	At this point in the syntax, the keyword **performs** should appear instead of the token *T*.

Error message		Meaning and remedy
Properties are not allowed for this call.	*Problem*	This *Other_Call* structure is not enclosed in parentheses and therefore it cannot include a list of *Call_Properties*. See section 5.7.
	Solution	Add parentheses around the *Other_Call*, thus allowing a full *Call_Description*.
Properties are not allowed for this loop.	*Problem*	This *Other_Loop* structure is not enclosed in parentheses and therefore it cannot include a list of *Loop_Properties*. See section 5.7.
	Solution	Add parentheses around the *Other_Loop*, thus allowing a full *Loop_Description*.
Property bounds for an instruction are not implemented.	*Problem*	The assertion tries to assert bounds on a target-specific property at an instruction (in an *Instruction_Block*). This assertion is not implemented for instructions.
"property" expected, at "*T*"	*Problem*	At this point in the syntax, the keyword **property** should appear instead of the token *T*.
Quoted character expected , at "*T*"	*Problem*	The assertion file should have a character in single quotes here ('c'), instead of the token *T*.
Quoted string expected , at "*T*"	*Problem*	The assertion file should have a string in double quotes here ("string"), instead of the token *T*.
"range" expected, at "*T*"	*Problem*	At this point in the syntax, the keyword **range** should appear instead of the token *T*.
Repetition bounds not allowed in this context	*Problem*	The assertion file contains an assertion on execution count (**repeats** *N* **times**) for a context where that is not allowed, such as a subprogram context. This assertion is allowed only for loops, calls, and instructions.
Role bounds ("performs") apply only to instructions	*Problem*	This *Role_Bound* fact is not in an instruction context.
Roles can be asserted only for instructions	*Problem*	This *Role_Bound* fact is not in an instruction context.
Semicolon after callees expected, at "*T*"	*Problem*	The list of callee subprograms in a *Callee_Bound* (section 5.19) should be followed by a semicolon, instead of the token *T*.
Semicolon after "end call" expected, at "*T*"	*Problem*	The keywords **end call** should be followed by a semicolon, instead of the token *T*.
Semicolon after "end instruction" expected, at "*T*"	*Problem*	The keywords **end instruction** should be followed by a semicolon, instead of the token *T*.
Semicolon after "end loop" expected, at "*T*"	*Problem*	The keywords **end loop** should be followed by a semicolon, instead of the token *T*.
Semicolon after "end subprogram" expected, at "*T*".	*Problem*	A subprogram block lacks the terminating semicolon. Instead, the next token is *T*.
Semicolon after global bound expected, at "*T*"	*Problem*	An assertion (on the values of a variable or a property) in the global context lacks the terminating semicolon. Instead, the next token is *T*.
Semicolon after subprogram option expected, at "*T*"	*Problem*	A subprogram option lacks the terminating semicolon. Instead, the next token is *T*.
Semicolon after volatility assertion expected, at "*T*"	*Problem*	A volatility assertion lacks the terminating semicolon. Instead, the next token is *T*.

Error message		Meaning and remedy
Semicolon expected after clause, at "*T*"	*Problem*	An assertion clause lacks the terminating semicolon. Instead, the next token is *T*.
Stack #*n*: *name*	*Problem*	This message follows an error message of the form "The program has no stack named S" and shows the actual name of stack number *n* in this target program.
Stack bounds for a loop are not allowed.	*Problem*	The assertion tries to assert stack bounds for a loop. This assertion is not allowed for loops, only for subprograms.
Stack bounds for an instruction are not allowed.	*Problem*	The assertion tries to assert stack bounds for an instruction. This assertion is not allowed for instructions, only for subprograms.
Stack bounds for the program are not allowed.	*Problem*	The assertion file tries to assert the stack usage or final stack height as a global fact. This assertion is not supported in the global context, only for subprograms.
Stack bounds not allowed in this context.	*Problem*	The assertion file tries to assert stack bounds in a context where such facts are not supported.
Stack-name required	*Problem*	The program has more than one stack, but this *Stack_Bound* clause does not name the stack to which the assertions apply.
	Reasons	Mistake in the assertion file, or a change in the compilation options or the structure of the target program that makes it use more than one stack.
	Solution	Add the stack name. The Application Note for your target processor shows the applicable stack names.
Start bounds apply only to loops.	*Problem*	The assertion tries to assert *Start_Bounds* in a context where they are not allowed. *Start_Bounds* can appear only for a loop, in a *Loop_Block*.
Subprogram address is invalid	*Problem*	An assertion tries to identy a subprogram by giving its (entry) address, but the address string is not in the proper form for this target processor. The invalid address string is shown in output field 4.
	Solution	Refer to the Application Note for your target processor and write the entry address in the proper form.
"subprogram" expected, at "*T*"	*Problem*	At this point in the syntax, the keyword **subprogram** should appear instead of the token *T*.
Subprogram name expected, at "*T*"	*Problem*	The assertion file should have a subprogram name (or address) here, in double quotes, instead of the token *T*.
Text at or after column *C* not understood: "*T*"	*Problem*	The text at or after column *C* on the current line in the assertion file is not a valid lexical "token" of the assertion language. The string *T* contains (part of) this text.
The cells *C* and *D* do not define a range	*Problem*	This volatility assertion on an address range is erroneous, because the two *variable_address* values given for the lower and upper bounds are not compatible (for example, they lie in different address spaces) and cannot be used to delimit a range of addresses. The cell-names *C* and *D* show the interpretation of the lower and upper address bound, respectively, using the target-specific names for storage cells.

Error message		Meaning and remedy
The integer literal "*n*" is not a valid number	*Problem*	The digit string *n* is not a valid number for some reason. Perhaps it has too many digits for the integer type that Bound-T uses for asserted numbers.
The number *n* is not a valid source-line number or offset.	*Problem*	If this *Source_Line* is of the form **line** *n*, the number *n* is zero or negative and thus invalid as a source-line number. If this *Source_Line* is of the form **line offset** *offset*, the sum *n* of the base line number (from the containing subprogram) and the *offset* is zero or negative and thus invalid as a source-line number.
The "omit" property cannot be negated	*Problem*	A *Sub_Option* structure (section 5.6) uses the keyword **omit** and the keyword **not** in a combination that says that the subprogram in question is "not omitted".
	Solution	Not being omitted is the default condition for any subprogram, so remove the assertion as redundant.
The program has no stack named *S*	*Problem*	The stack name *S* given in this *Stack_Bound* clause does not match the name of any stack in the target program. This message is followed by informative error messages of the form "Stack #n: name" to show the names of the stacks that do exist in this target program.
	Reasons	Mistyped stack name, or a change in the compilation options such that the program no longer uses this stack.
	Solution	Refer to the Application Notes for your target processor to check the stack name.
The program has no stacks	*Problem*	The target program has no stacks (at all), so this *Stack_Bound* clause is not applicable.
	Reasons	Perhaps a change in the compilation options such that the program no longer uses any stacks (that Bound-T knows of).
	Solution	Refer to the Application Notes for your target processor.
The "unused" property cannot be negated *or* The "used" property cannot be asserted	*Problem*	A *Sub_Option* structure (section 5.6) uses the keywords **used** or **unused** and perhaps also the keyword **not** in a combination that says that the subprogram in question is "not unused".
	Solution	Being used is the default condition for any subprogram, so remove the assertion as redundant.
"time" expected, at "*T*"	*Problem*	The keyword **time** is expected after the keyword **for**, to form the assertion **enough for time**. See section 2.17.
"times" expected, at "*T*"	*Problem*	This execution-count assertion should have the key-word **times** (or **time**) instead of the token *T*, after the asserted execution time. It should be **repeats** *N* **times**.
Unrecognized property name: *P*	*Problem*	The assertion file names a target-specific "property" *P* but there is no such property for this target processor.
	Solution	Check the target-specific Application Note for the names of the properties for this processor.
"usage" or "final" expected, at "*T*"	*Problem*	One of the keywords **usage** or **final** is expected in this *Stack_Value*, instead of the token *T*.

Error message		Meaning and remedy
Variable bounds for an instruction are not implemented.	*Problem*	The assertion tries to assert bounds on a variable at an instruction (in an *Instruction_Block*). This assertion is not implemented for instructions.
“variable” expected, at “*T*”	*Problem*	At this point in the syntax, the keyword **variable** should appear instead of the token *T*.
Variable invariance for an instruction is not implemented.	*Problem*	The assertion tries to assert that a variable is invariant at an instruction (in an *Instruction_Block*). This assertion is not implemented for instructions.
Variable not found: *V*	*Problem*	The assertion file names a variable *V*, but the target program’s symbol table does not have a variable named *V* (in the implicit or explicit scope).
	Reasons	The name *V* may be mistyped; if the default scope is used perhaps another should be named explicitly; or the target compiler may have mangled the names.
	Solution	Check for typos. Check the target program’s symbol table using eg. *-trace symbols* or by dumping the file.
Variable “*V*” is ambiguous	*Problem*	The assertion file names a variable *V*, but the target program’s symbol table lists several variables named *V* (in the implicit or explicit scope).
	Reasons	Perhaps some of these variable are local variables (declared in local scopes) but match the name *V* because the assertion does not define the scope of the variable.
	Solution	Add scope to the variable name in the assertion. Check the target program’s symbol table using eg. *-trace symbols* or by dumping the file.
Variable name expected, at “*T*”	*Problem*	The assertion file should here have the name (or address) of a variable, instead of the token *T*.
Variable name or "range" expected, at “*T*”	*Problem*	At this point in this volatility assertion, there should be the name (or address) of a variable, or the keyword **range**, instead of the token *T*.
Void bounds on stack final height ignored	*Problem*	The assertion bounds the **final** stack height to a void (null, empty) range. This is a contradiction. Bound-T ignores the assertion.
Void bounds on stack usage ignored	*Problem*	The assertion bounds the stack **usage** to a void (null, empty) range. This is a contradiction. Bound-T ignores the assertion.
“volatile” expected, at “*T*”	*Problem*	At this point in the syntax, the keyword **volatile** should appear instead of the token *T*.

Tidorum Ltd

Tiirasaarentie 32
FI-00200 Helsinki, Finland
www.tidorum.fi
Tel. +358 (0) 40 563 9186
Fax +358 (0) 42 563 9186
VAT FI 18688130