

Bound-T timing analysis tool



Bound-T  
Assertion Language



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 – 2008 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: Version 6.2  
Document issue date: 2008-02-22  
Bound-T version: 3  
Last change included: BT-CH-0106  
Web location: <http://www.bound-t.com/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>.

## Preface

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.tidorum.fi/>.

If you have comments or questions on this document or the product, they are welcome via electronic mail to the address [info@tidorum.fi](mailto: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

Web: <http://www.tidorum.fi/>

E-mail: [info@tidorum.fi](mailto: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

<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
1.1	What Bound-T is.....	1
1.2	Overview of this document.....	3
1.3	Other Bound-T documentation.....	4
<b>2</b>	<b>WRITING ASSERTIONS</b>	<b>6</b>
2.1	What assertions are.....	6
2.2	Assertion = context + fact.....	7
2.3	Assertions on the repetition of loops.....	10
2.4	Assertions on the execution count of calls.....	12
2.5	Assertions on the execution time of a subprogram.....	15
2.6	Assertions on the execution time of a call.....	15
2.7	Assertions on the stack usage of a subprogram.....	17
2.8	Assertions on the final stack height of a subprogram.....	18
2.9	Assertions on the callees of a dynamic call.....	20
2.10	Assertions on variable values.....	21
2.11	Assertions on variable invariance.....	24
2.12	Assertions on target-specific properties.....	27
2.13	Special assertions on subprograms.....	28
<b>3</b>	<b>IDENTIFYING PROGRAM PARTS</b>	<b>32</b>
3.1	Scopes and qualified names.....	32
3.2	Naming subprograms.....	33
3.3	Naming variables.....	34
3.4	Identifying loops.....	35
3.5	Identifying calls.....	41
<b>4</b>	<b>ETERNAL LOOPS AND RECURSION</b>	<b>46</b>
4.1	Handling eternal loops.....	46
4.2	Handling recursion.....	48
<b>5</b>	<b>ASSERTION LANGUAGE SYNTAX AND MEANING</b>	<b>52</b>
5.1	Introduction.....	52
5.2	Assertion syntax basics.....	52
5.3	Scopes.....	55
5.4	Global bounds.....	56
5.5	Subprograms.....	56
5.6	Loops.....	58
5.7	Calls.....	60
5.8	Clauses and facts.....	62
5.9	Execution-time bounds.....	63

5.10	Stack bounds.....	64
5.11	Repetition bounds.....	64
5.12	Variable bounds.....	69
5.13	Variable invariance.....	71
5.14	Property bounds.....	72
5.15	Callee bounds.....	73
5.16	Combining Assertions.....	73
<b>6</b>	<b>TROUBLESHOOTING</b>	<b>76</b>
6.1	Warning Messages.....	76
6.2	Error Messages.....	77

## Tables

Table 1:	Meaning of loop properties.....	59
Table 2:	Meaning of call properties.....	61
Table 3:	Fact and context combinations.....	62
Table 4:	Meaning of execution time assertion.....	63
Table 5:	Meaning of repetition count assertion.....	64
Table 6:	Meaning of variable value assertion.....	70
Table 7:	Meaning of variable invariance assertion.....	72
Table 8:	Meaning of property value assertion.....	73
Table 9:	Effect of multiple assertions on the same item.....	73
Table 10:	Warning messages.....	76
Table 11:	Assertion error messages.....	78

## Figures

Figure 1:	Inputs and outputs.....	3
Figure 2:	An assertion file.....	9
Figure 3:	An irreducible flow-graph.....	30
Figure 4:	Longest call path in recursion example.....	49
Figure 5:	A loop in a flow-graph.....	66
Figure 6:	A general kind of loop asserted to repeat 6 times.....	67
Figure 7:	A middle-exit loop asserted to repeat 6 times.....	68
Figure 8:	An exit-at-end loop asserted to repeat 6 times.....	68

This page is intentionally blank except for this note.

# 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”.

### *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***

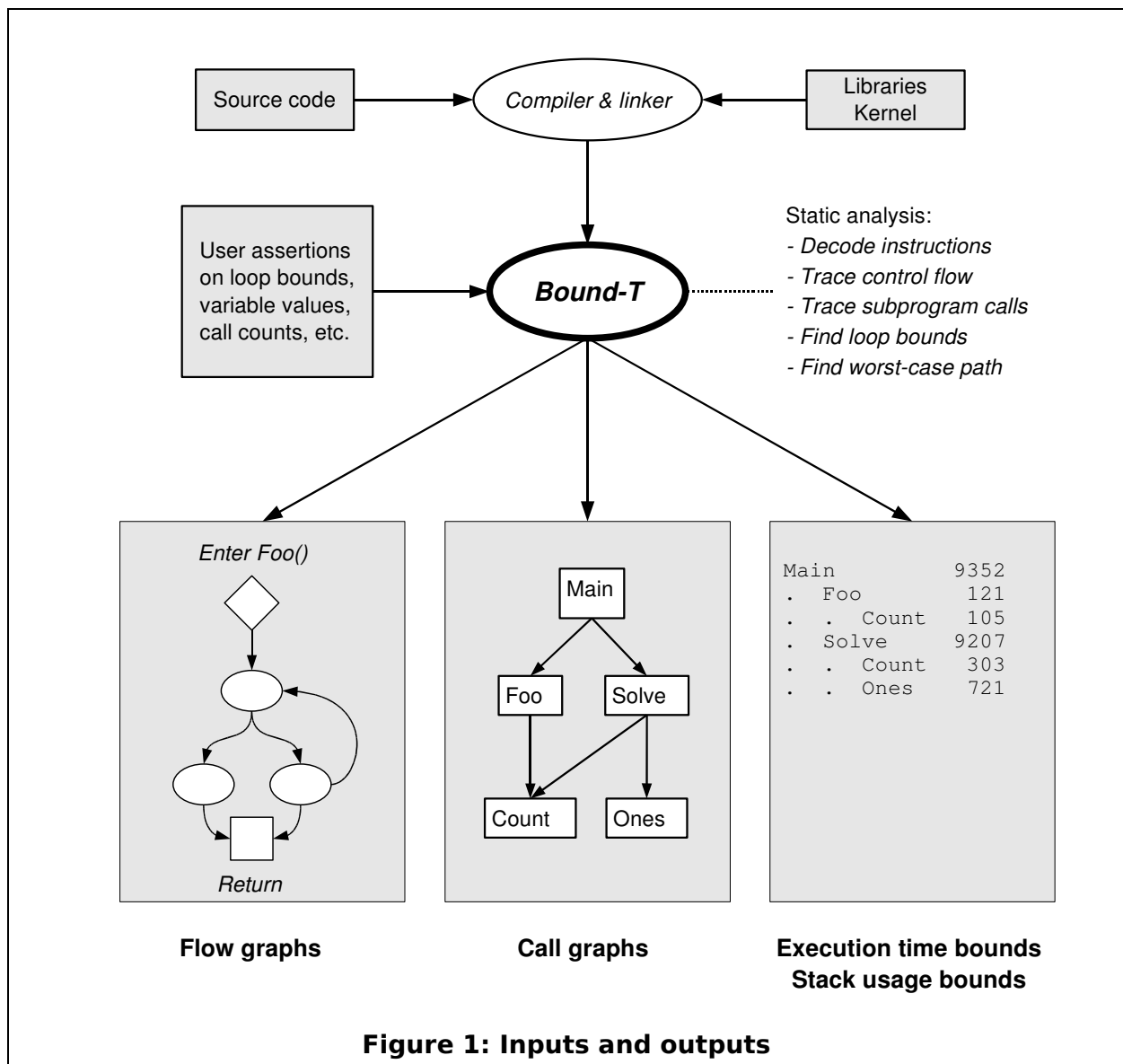
Also bear in mind that Bound-T produces an upper bound for the execution time, which may be different from the exact worst-case time. Various approximations in Bound-T's analysis algorithms may give over-estimated, too conservative bounds. However, the bounds can be sharpened by suitable assertions.

### ***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 plus command-line arguments and options (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.





## 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/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/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/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](http://www.bound-t.com/app_notes) for a list of the currently supported target processors and the available Application Notes.

### ***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/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 your 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),
- execution count of a call or loop,
- 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,
- value or range for some target-specific property in a part of the program.

### ***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.

The set of loops or calls is defined 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.

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.

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.

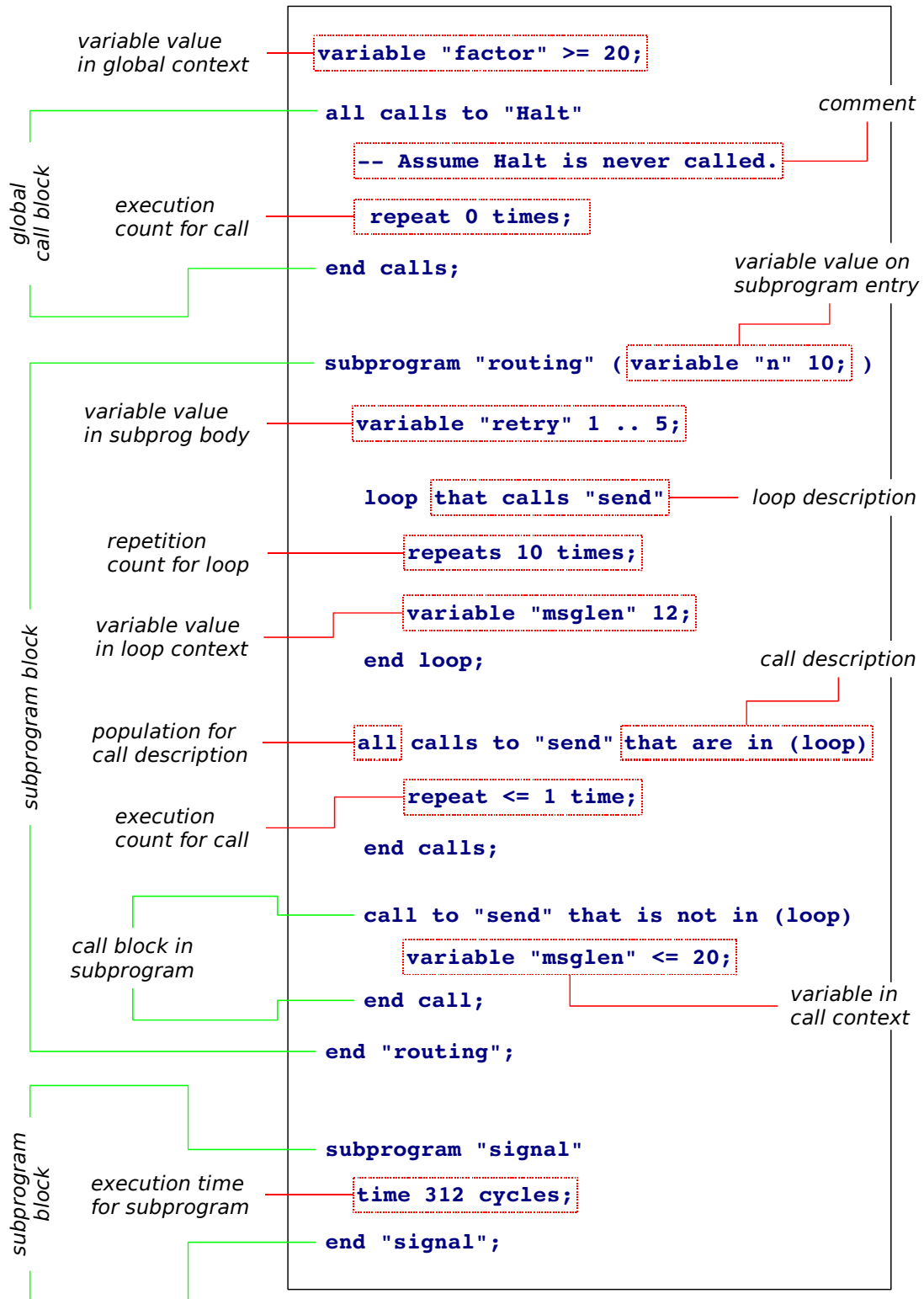


Figure 2: An assertion file

## 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 5.11 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 5.11 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 2.10 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  $100 \times 100$  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 to 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 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 2.10 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 ever 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 2.13. 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.

## 2.5 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 2.13. 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 TBA.

### *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";
```

## 2.6 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.

## **2.7 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 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. 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 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 for it, you should use the **omit** assertion. See section 2.13 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.

## **2.8 Assertions on the final stack height of a subprogram**

### *Why?*

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

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.

When Bound-T analyses a subprogram it always attempts to compute the final stack height. 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 case 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.

### *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 there 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 2.7 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";
```

## 2.9 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;
```

## 2.10 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.

## **2.11 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.

## 2.12 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.

## 2.13 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 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";
```

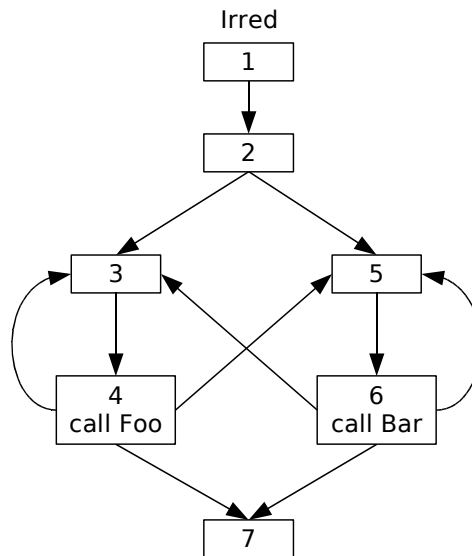
However, such an assertion does *not* omit calls to the omitted subprogram, so the calls (and callers) will be *unbounded* (have unknown execution-time and stack-usage bounds) unless you also assert bounds on time and/or stack usage for the subprogram or for each call.

If you assert sufficient time and stack usage 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, in particular on calls, 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 3: 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 Scopes and qualified names

#### *Scopes qualify names*

Assertions 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`.

## 3.2 Naming subprograms

When writing assertions you may need to write a subprogram name 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.

### *By symbolic name*

Subprograms are usually named 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 "named" 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.

### 3.3 Naming variables

When writing assertions you may need to name 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.

#### *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 3.2.

#### *By machine address*

Variables can also be “named” 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.

## 3.4 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.

### *Loop properties*

A loop can be identified firstly by the subprogram that contains the loop and secondly by specific properties of the loop itself.

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.

In addition, the following specific properties or keywords can be used to identify loops:

- |                    |                                                                          |
|--------------------|--------------------------------------------------------------------------|
| <b>is labelled</b> | The loop contains (or does not contain) a specific statement label.      |
| <b>calls</b>       | The loop calls (or does not call) a specific subprogram.                 |
| <b>uses</b>        | The loop reads (or does not read) a specific variable.                   |
| <b>defines</b>     | The loop assigns (or does not assign) to a specific variable.            |
| <b>is in</b>       | The loop is contained (nested) in another loop (or is not so contained). |
| <b>contains</b>    | 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 **is 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 3.5.

### *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**, **is labelled** and **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 to *Check\_Power*.

### **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 not calls "Fee")
  and not contains (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 **is 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)`.

## 3.5 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.

### *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***

The following 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 3.2). 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. 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, the only way to identify a *subset* of calls from the same caller to the same callee is to describe the calls by the loops that contain them. 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 3.4.

### *Non-returning subprograms are never in a loop*

A call to a subprogram that is marked "no return" (see section 2.13) 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  $2 \times 10 = 20$  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  $4 \times 10 = 40$  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.

## 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.

end;
```

```

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;

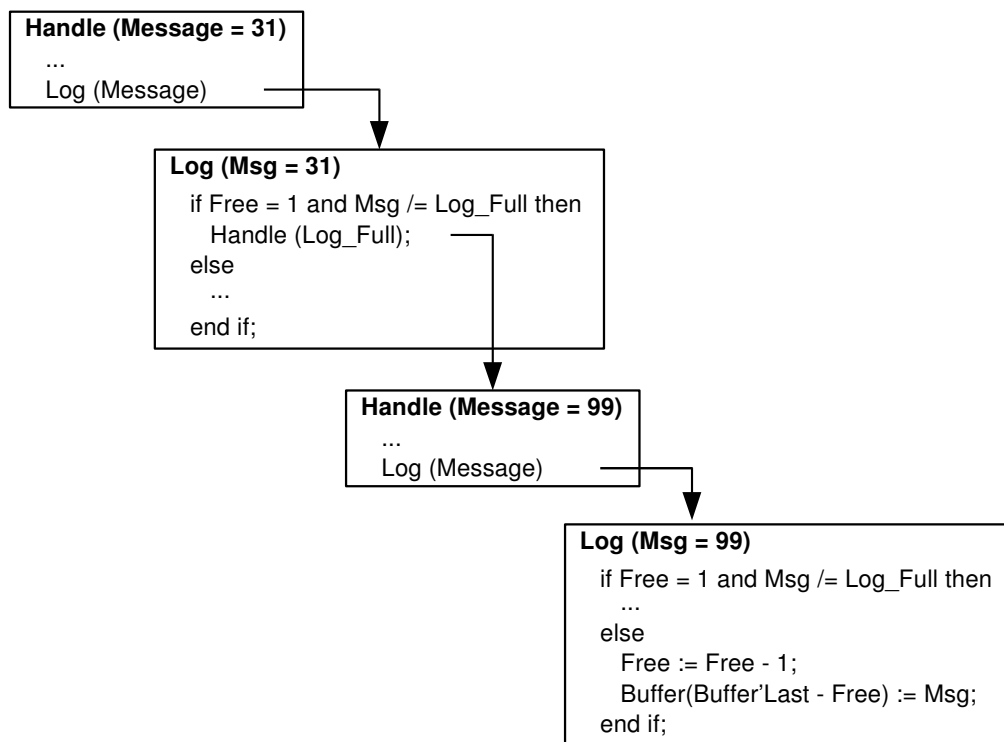
end Errors;

```

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 4: 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 characters, not including the double quote ("), and itself enclosed in double quotes. Example: "foo|memo".

### *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 3.2.

### ***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.

### ***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*

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 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*; in this case the subset is bounded only at one end.

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.

### *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. Moreover, a few keyword pairs have obsolete single-word equivalents with embedded underscores. Here are the equivalent keywords:

*First form*                      *Equivalent second form*

<b>call</b>	<b>calls</b>
<b>contain</b>	<b>contains</b>
<b>cycle</b>	<b>cycles</b>
<b>define</b>	<b>defines</b>
<b>is</b>	<b>are</b>
<b>loop</b>	<b>loops</b>
<b>repeat</b>	<b>repeats</b>
<b>time</b>	<b>times</b>
<b>use</b>	<b>uses</b>

*Keyword pair*                      *Obsolete but equivalent keyword*

<b>call to</b>	<b>call_to</b>
<b>end call</b>	<b>end_call</b>
<b>end loop</b>	<b>end_loop</b>
<b>is in</b>	<b>is_in</b>
<b>loop that</b>	<b>loop_that</b>
<b>no arithmetic</b>	<b>no_arithmetic</b>

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;
```

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

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

The meaning of various assertions has already been explained by examples in Chapter 2. Here we will give a full syntax. We proceed in a top-down order, first explaining the overall structure of the assertion file and then the details.

### *Overall assertion structure*

The start symbol is *Assertions*, representing the whole assertion file. The 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.

### *Comments*

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

## 5.3 **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 3.1 and .

### *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.

## 5.4 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 four types of global bounds, namely variable bounds, property bounds, loop blocks and call blocks:

```
Global_Bound  →  Variable_Bound ;
                |  Property_Bound ;
                |  Loop_Block ;
                |  Call_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 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 5.6) 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.

## 5.5 Subprograms

### *Subprogram blocks and subprogram names*

A subprogram block collects assertion statements that shall be applied only to the analysis of the named subprogram.

```
Sub_Block      →  subprogram Sub_Name [ ( { Parameter } ) ]
                  [ { Statement } ]
                  [ end [ subprogram ] [ Sub_Name ] ; ]

Sub_Name       →  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 ;
                |  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
                |  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).

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 application 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.

## 5.6 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 selected loops in this subprogram only. If the *Loop\_Block* occurs as a *Global\_Bound*, it applies to the selected 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

Given some initial set (universe) of loops, subsets of loops are described by conjunctions of zero or more loop properties:

Loop\_Description → **loop** [ **that** Loop\_Properties ]

Loop\_Properties → Loop\_Property [ **and** Loop\_Properties ]

If the loop description contains no properties (no **that** part), 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.

One of the important properties 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 5.7 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”.

The loop properties are then defined as follows:

Loop\_Property → **contains** Count Other\_Loop  
| **is [ not ] in** Other\_Loop  
| **contains** Count Other\_Call  
| **calls** Sub\_Name  
| **uses** Variable\_Name  
| **defines** Variable\_Name  
| **is [ not ] labelled** Label\_Name  
| **executes** code-address  
| **executes offset** code-offset  
| **not** Loop\_Property

Label\_Name → *symbol*

Table 1 below defines exactly the meaning of each type of loop property.

**Table 1: Meaning of loop properties**

Property	Loop <i>L</i> has this property if and only if:
contains Count Other_Loop	Within the set of all inner loops directly contained in <i>L</i> , the given Count of loops matching the <i>Loop_Description</i> .
is in Other_Loop	<i>L</i> is directly contained in an outer loop that matches the <i>Other_Loop</i> description.
is not in Other_Loop	Either <i>L</i> is not contained in any outer loop, or the loop that directly contains <i>L</i> does not match the <i>Other_Loop</i> description.
contains Count Other_Call	<i>L</i> (or some inner loop) contains the given Count of calls matching <i>Other_Call</i> .
calls Sub_Name	Same as “contains >= 1 calls to <i>Sub_Name</i> ”.

Property	Loop L has this property if and only if:
uses Variable_Name	<i>L</i> (or some inner loop) contains an instruction that reads (uses) the value of the variable identified by <i>Variable_Name</i> .
defines Variable_Name	<i>L</i> (or some inner loop) contains an instruction that writes (assigns a value) to the variable identified by <i>Variable_Name</i> .
is labelled Label_Name	<i>L</i> (or some inner loop) contains the instruction that has the code address assigned to <i>Label_Name</i> .
is not labelled Label_Name	Neither <i>L</i> nor any of its inner loops contains the instruction that has the code address assigned to <i>Label_Name</i> .
executes code-address	<i>L</i> (or some inner loop) contains the instruction that has the given code address.
executes offset code-offset	<i>L</i> (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 <i>Sub_Block</i> .
not Loop_Property	The negation (logical complement) of the <i>Loop_Property</i> .

The ability to write the **not** keyword after the **is** keyword is only syntactic sugar. The form “**is not ...**” means exactly the same as “**not is ...**”.

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 “not contains (loop ...)” property must be written, for example as in:

```
loop that
  calls "Foo"
  and not contains (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.

## 5.7 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 selected calls in this subprogram only. If the *Call\_Block* occurs as a *Global\_Bound*, it applies to the selected 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. The most important property is the callee subprogram, when this is statically known, that is, when the call instruction statically specifies the address of the callee. 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 [ that 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 properties (no **that** part), 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 properties are listed, a match in addition requires that all the listed properties are true for this call.

```

Call_Property     →  is [ not ] in Other_Loop
                   |  uses Variable_Name
                   |  defines Variable_Name
                   |  not Call_Property

```

Table 2 below defines the meaning of each type of call property.

**Table 2: Meaning of call properties**

<b>Property</b>	<b>Call C to Sub_Name has this property if and only if:</b>
is in Other_Loop	<i>C</i> is contained in a loop that matches the <i>Other_Loop</i> description. Note that this loop is not necessarily the innermost loop that contains <i>C</i> .
is not in Other_Loop	<i>C</i> is not contained in any loop, or no loop that contains <i>C</i> matches the <i>Other_Loop</i> Description. Note that the test applies also to outer loops, not just to the innermost loop that contains <i>C</i> .
uses Variable_Name	Not implemented. Has no effect.
defines Variable_Name	Not implemented. Has no effect.
not Call_Property	The negation (logical complement) of the <i>Call_Property</i> .

The ability to write the **not** keyword after the **is** keyword is only syntactic sugar. The form “**is not ...**” means exactly the same as “**not is ...**”.

## 5.8 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 ;
        | Repetition_Bound ;
        | Variable_Bound ;
        | Property_Bound ;
        | Variable_Invariance ;
        | Callee_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.

**Table 3: Fact and context combinations**

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

### *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.

Finally, the set of possible callees (*Callee\_Bound*) is obviously relevant only to dynamic calls and cannot be asserted in any other context.

## 5.9 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 4: 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.

## 5.10 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).

## 5.11 Repetition bounds

Bounds on the number of repetitions of a loop or the number of executions of a call 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 as a *Clause* in a *Call\_Block* (in the latter case we sometimes call it "execution count" bounds). The following table explains the meaning of asserting the repetition (or execution) count in each context where such an assertion is allowed.

**Table 5: 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).

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

### *Repetition bounds for calls*

When a repetition bound applies to a call, it constrains the worst-case execution path of the containing subprogram so that the number of executions of the call 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 can *decrease* the overall execution time, since forcing the execution to pass more often through this call 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.

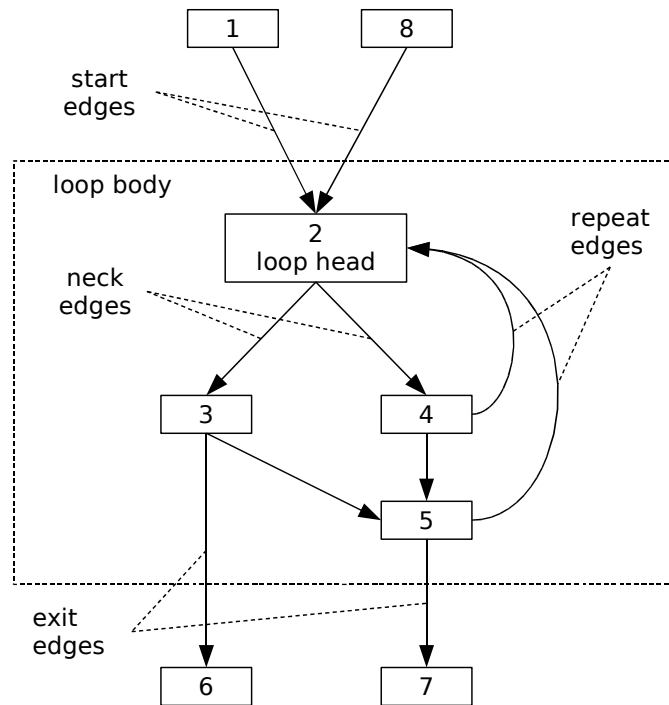
Loops correspond to cyclic paths in the flow-graph. Bound-T currently requires that the structure of the flow-graph be *reducible*, which means that two loops are either completely separate (share no nodes or edges) or one is completely nested within the other.

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 5 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.
- 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.



**Figure 5: A loop in a flow-graph**

The loop head is node 2; the loop body consists of nodes 2, 3, 4, and 5; the repeat edges are those from node 4 to node 2 and from node 5 to node 2; and the exit edges are those from node 3 to node 6 and from node 5 to node 7.

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.

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.

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.

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.

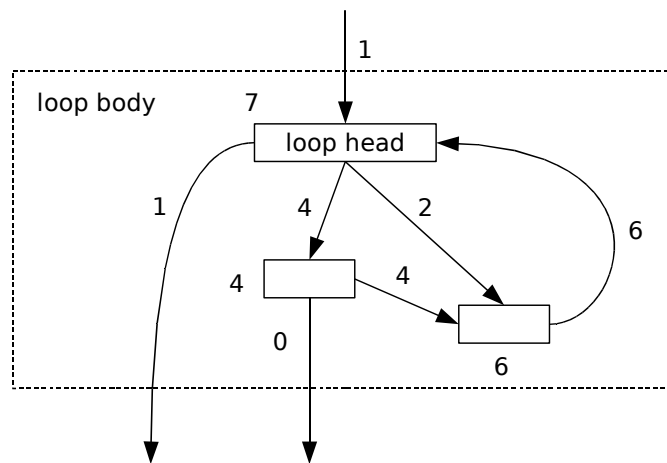


- When a repetition bound with the number  $R$  as the upper *Bound* applies to an exit-at-end loop it constrains the worst-case execution path as follows. If the start edges are executed a total of  $A$  times, then the repeat edges are executed in total at most  $(R - 1) \times A$  times.

Note that only the upper bound of of the repetition *Bound* is used in either case.

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.

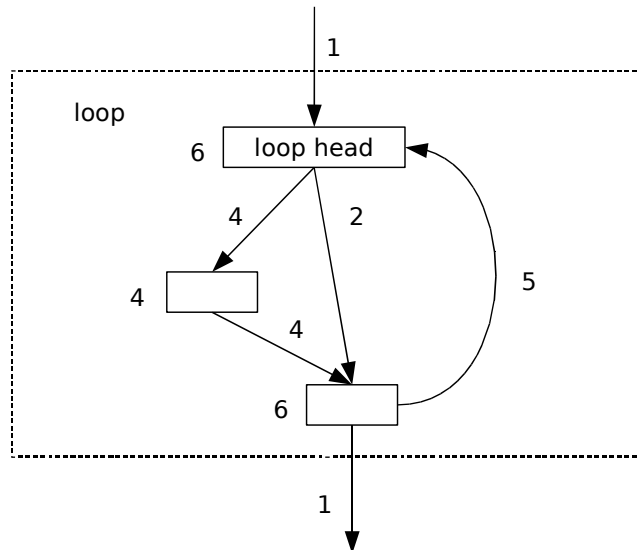
The following figure shows an example of the most general form of a loop with exit edges both from the loop head and from the loop body. The nodes and edges in the flow-graph are labelled with execution counts assuming an assertion that the loop repeats  $R = 6$  times and one start of the loop,  $A = 1$ . (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 6: 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 7 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 8 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 8: 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 and call repetitions***

When a call is in a loop, a bound on the number of executions of the call may implicitly bound the number of loop repetitions. However, Bound-T also requires an explicit bound on each loop in the subprogram before it tries to compute the WCET of the subprogram. 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 calls executed) 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 unless an explicit loop-bound is asserted, 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.

## **5.12 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 6: 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.
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 2.10 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.

## **5.13 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 2.11 explained.

The following table explains the meaning of asserting the invariance of a variable in each context where such an assertion is allowed.

**Table 7: 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.

## 5.14 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 8: 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 overridden 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.
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).

## 5.15 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.

## 5.16 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 9: 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.

Asserted fact	Effect in same context	Effect in nested context
Variable invariance	Multiple assertions have the same effect as a single assertion.	The assertion for the inner context holds.
Loop repetition count	The effective bound on repetitions is the minimum of the asserted values.	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.
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.

### ***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 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 occurs 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 the 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 10: Warning messages**

Warning Message		Meaning and Remedy
Ignoring asserted target <i>A</i> for <i>C</i>	<i>Reasons</i>	An assertion on the callees of a dynamic call listed a callee with entry address <i>A</i> , but Bound-T is unable to accept such assertions on this class, <i>C</i> , of dynamic call.
	<i>Action</i>	This warning is preceded by a <i>Fault</i> message, because this problem indicates a fault in Bound-T (unimplemented operation on class <i>C</i> ). Please contact Tidorum Ltd.
Lower bound on stack usage set to zero	<i>Reasons</i>	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.
	<i>Action</i>	Change the lower bound to a non-negative value.
Multiple assertions on stack final height	<i>Reasons</i>	The same <i>Stack_Bound</i> clause gives more than one <b>final</b> stack height bound for the same stack. This is either redundant or contradictory.
	<i>Action</i>	Use only one <b>final</b> bound.
Multiple assertions on stack usage	<i>Reasons</i>	The same <i>Stack_Bound</i> clause gives more than one stack <b>usage</b> bound for the same stack. This is either redundant or contradictory.
	<i>Action</i>	Use only one <b>usage</b> bound.
No valid assertions found in this file	<i>Reasons</i>	This assertion file seems to be empty, or to contain only comments, or only assertions that are in error.
	<i>Action</i>	Correct the erroneous assertions, if any. Check that assertions have not, by mistake, been commented out or removed.
Other address connection not used.	<i>Reasons</i>	An assertion uses a numeric address to name a subprogram or statement-label, but this address is connected to more than one symbolic (source-code) name (multiply defined). This message shows a connection that Bound-T did not use.

Warning Message		Meaning and Remedy
		This is generally harmless since the analysis uses the address rather than the symbolic name. The symbolic name appears only in the outputs.
	<i>Action</i>	Change the assertion to use the desired symbolic name.
Other identifier connection not used.	<i>Reasons</i>	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.
	<i>Action</i>	Add scope context to the name in the assertion, to make the name unambiguous. See section 3.1.
This address connection used.	<i>Reasons</i>	An assertion uses a numeric address to name a subprogram or statement-label, but this address is connected to more than one symbolic (source-code) name (multiply defined). This message shows the connection that Bound-T uses.
		This is generally harmless since the analysis uses the address rather than the symbolic name. The symbolic name appears only in the outputs.
	<i>Action</i>	Change the assertion to use the desired symbolic name.
This identifier connection used.	<i>Reasons</i>	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.
	<i>Action</i>	Add scope context to the name in the assertion, to make the name unambiguous. See section 3.1.
Subprogram not found: S	<i>Reasons</i>	In the assertion file, the subprogram named for this subprogram block (after the <b>subprogram</b> 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.
	<i>Action</i>	Correct the assertion file to use the subprogram name as it exists in the target program executable (the link-name). To find the possibly mangled name, use <i>-trace symbols</i> or dump the target program file.

## 6.2 Error Messages

When the assertion parser in Bound-T finds an error in the 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 11: Assertion error messages**

<b>Error Message</b>		<b>Meaning and Remedy</b>
"S" expected, at "T"	<i>Problem</i>	At the end of a subprogram block, for a subprogram identified by <i>S</i> , the subprogram identifier is repeated in the form <i>T</i> which does not match <i>S</i> . See also the error message that begins "Mismatch...".
"A" is not a valid cell address	<i>Problem</i>	An assertion contains the string <i>A</i> that is meant to denote a variable (storage cell) address, but is rejected by the assertion parser.
	<i>Reasons</i>	The string <i>A</i> is not written according to the rules for variable addresses that Bound-T uses for this target processor.
	<i>Solution</i>	Refer to the Application Note for this target and correct the string.
"A" is not a valid code address, at "T".	<i>Problem</i>	An assertion contains the string <i>A</i> that is meant to denote a code address, but is rejected by the assertion parser. The next token is <i>T</i> .
	<i>Reasons</i>	The string <i>A</i> is not written according to the rules for code addresses that Bound-T uses for this target processor.
	<i>Solution</i>	Refer to the Application Note for this target and correct the string.
"A" is not a valid code offset, at "T".	<i>Problem</i>	An assertion contains the string <i>A</i> that is meant to denote a code offset, but is rejected by the assertion parser. The next token is <i>T</i> .
	<i>Reasons</i>	The string <i>A</i> is not written according to the rules for code offsets that Bound-T uses for this target processor.
	<i>Solution</i>	Refer to the Application Note for this target and correct the string.
Ambiguous label name: <i>N</i> or Ambiguous subprogram name: <i>N</i> or Ambiguous variable name: <i>N</i>	<i>Problem</i>	The assertion tries to identify a label, subprogram, or variable by the name <i>N</i> but the name matches more than one label, subprogram, or variable (respectively) in the program and so is ambiguous.
	<i>Reasons</i>	The program contains more than one label, subprogram, or variable (respectively) with the name <i>N</i> but in different scopes. The name in the assertion does not specify the scope (well enough).
	<i>Solution</i>	Add scope levels to the name to make it unambiguous.
Assertion expected, at "T".	<i>Problem</i>	The current token <i>T</i> cannot be the start of an assertion, as would be expected here. See the nonterminal <i>Assertions</i> in section .  The assertion parser silently skips the following text until it finds the start of the next assertion.

<b>Error Message</b>		<b>Meaning and Remedy</b>
Assertion file contained errors.	<i>Problem</i>	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.	<i>Problem</i>	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	<i>Problem</i>	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.	<i>Problem</i>	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).
A “that” part is not allowed for this call.	<i>Problem</i>	This <i>Other_Call</i> structure is not enclosed in parentheses and therefore it cannot include a list of call properties introduced with the <b>that</b> keyword. See section 5.7.
	<i>Solution</i>	Add parentheses around the <i>Other_Call</i> , thus allowing a full <i>Call_Description</i> .
A “that” part is not allowed for this loop.	<i>Problem</i>	This <i>Other_Loop</i> structure is not enclosed in parentheses and therefore it cannot include a list of loop properties introduced with the <b>that</b> keyword. See section 5.6.
	<i>Solution</i>	Add parentheses around the <i>Other_Loop</i> , thus allowing a full <i>Loop_Description</i> .
Bound expected, at "T"	<i>Problem</i>	The assertion file should have a bound here, instead of the token <i>T</i> . See the nonterminal <i>Bound</i> in section .
"call" after "end" expected, at "T"	<i>Problem</i>	The keyword <b>call</b> should here follow the keyword <b>end</b> . See the nonterminal <i>Call_Block</i> in section 5.7.
Call properties expected, at "T"	<i>Problem</i>	The assertion file should have a call property here, instead of the token <i>T</i> . See the nonterminal <i>Call_Property</i> in section 5.7.
Calls do not have the "defines" property.	<i>Problem</i>	The assertion file tries to use the <b>defines</b> keyword to identify a call. This property is not yet supported for calls.
Calls do not have the "uses" property.	<i>Problem</i>	The assertion file tries to use the <b>uses</b> keyword to name a identify. This property is not yet supported for calls.
Cannot assert callees for a static call.	<i>Problem</i>	The assertion file tries to assert the possible callees for a static call, which is nonsense. In other words, there is a <i>Callee_Bound</i> in a <i>Call_Block</i> for a static call.
Clause expected, at "T"	<i>Problem</i>	The assertion file should have an assertion clause here, instead of the token <i>T</i> . See the nonterminal <i>Clause</i> in section 5.8.
Clause or "end call" expected, at "T"	<i>Problem</i>	The assertion file should have an assertion clause here, or the <b>end call</b> keywords, instead of the token <i>T</i> . See the nonterminals <i>Clause</i> in section 5.8 and <i>Call_Block</i> in section 5.7.

Error Message		Meaning and Remedy
Clause or "end loop" expected, at "T"	<i>Problem</i>	The assertion file should have an assertion clause here, or the <b>end loop</b> keywords, instead of the token <i>T</i> . See the nonterminals <i>Clause</i> in section 5.8 and <i>Loop_Block</i> in section 5.6.
Closing parenthesis after call expected, at "T"	<i>Problem</i>	A call description that is enclosed in parentheses should be followed by a ')', instead of the token <i>T</i> . See the nonterminal <i>Other_Call</i> in section 5.6.
Closing parenthesis after loop expected, at "T"	<i>Problem</i>	A loop description that is enclosed in parentheses should be followed by a ')', instead of the token <i>T</i> . See the nonterminal <i>Other_Loop</i> in section 5.6.
Closing parenthesis after parameters expected, at "T"	<i>Problem</i>	The assertions on subprogram parameters should be followed by a ')', instead of the token <i>T</i> . See the nonterminal <i>Sub_Block</i> in section 5.5.
Context provides no base for code offset	<i>Problem</i>	A loop description uses the property “ <b>executes offset code-offset</b> ” 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 <i>Loop_Block</i> that is a <i>Global_Bound</i> .
"cycles" expected, at "T"	<i>Problem</i>	This execution-time assertion should have the keyword <b>cycles</b> , instead of the token <i>T</i> , after the asserted execution time. It should be “ <b>time N cycles</b> ”.
Dynamic call cannot have “call to” or Dynamic call cannot have “call_to”.	<i>Problem</i>	The assertion describes a call that is <b>dynamic</b> but is also a <b>call to</b> a named callee, which is an impossible combination.
“end call” expected, at "T"	<i>Problem</i>	The present <i>Call_Block</i> should end with <b>end call</b> , instead of the token <i>T</i> .
“end loop” expected, at "T"	<i>Problem</i>	The present <i>Loop_Block</i> should end with <b>end loop</b> , instead of the token <i>T</i> .
Execution time bounds for a loop are not allowed.	<i>Problem</i>	The assertion file tries to assert the <b>time</b> for a loop. This assertion is not supported for loops, only for subprograms and calls.
Execution time bounds for the program are not allowed.	<i>Problem</i>	The assertion file tries to assert the <b>time</b> as a global fact. This assertion is not supported in the global context, only for subprograms and calls.
Final height of stable stack must be zero: <i>S</i>	<i>Problem</i>	This <i>Stack_Bound</i> clause contains a <b>final</b> stack-height assertion that allows a non-zero value. This is a contradiction because this stack, named <i>S</i> , is known to Bound- <i>T</i> as a “stable” stack, where the final height is always zero.
	<i>Reasons</i>	Perhaps a change in the compilation options for the target program, or in the command-line options for Bound- <i>T</i> , have changed the stack to be classed as “stable” instead of “unstable”.
	<i>Solution</i>	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.

Error Message		Meaning and Remedy
"for" after "enough" expected, at "T"	<i>Problem</i>	The keyword <b>for</b> is expected after the keyword <b>enough</b> , to form the assertion <b>enough for time</b> . See section 2.13.
Integer value expected, at "T"	<i>Problem</i>	The assertion file should have an integer literal here, instead of the token <i>T</i> .
Integrated analysis cannot be negated.	<i>Problem</i>	The assertion tries to negate (disable) integrated analysis of a subprogram; this is not possible.
	<i>Solution</i>	Non-integrated analysis is the default, so remove the assertion as redundant. See section 5.5.
Label not found: <i>L</i>	<i>Problem</i>	The assertion file names a label <i>L</i> but the target program's symbol table does not have a statement label named <i>L</i> in this scope.
	<i>Reasons</i>	The name <i>L</i> 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.
	<i>Solution</i>	Check for typos. Check the target program's symbol table using eg. <i>-trace symbols</i> or by dumping the file.
"loop" after "end" expected, at "T"	<i>Problem</i>	The keyword <b>loop</b> should here follow the keyword <b>end</b> . See the nonterminal <i>Loop_Block</i> in section 5.6.
Loop description expected, at "T"	<i>Problem</i>	A loop description is expected here, instead of the token <i>T</i> . See the nonterminal <i>Loop_Description</i> in section 5.6.
Loop or call expected, at "T"	<i>Problem</i>	A loop-block or call-block is expected here, instead of the token <i>T</i> . See the nonterminals <i>Loop_Block</i> in section 5.6 and <i>Call_Block</i> in section 5.7.
Loop property expected, at "T"	<i>Problem</i>	The assertion file should here have a loop property, instead of the token <i>T</i> . See the nonterminals <i>Loop_Property</i> and <i>Loop_Properties</i> in section 5.6.
Mismatched subprogram identifier " <i>S</i> " after "end" <i>followed by</i> " <i>N</i> " expected, at "T"	<i>Problem</i>	The subprogram identifier <i>S</i> after the final <b>end</b> of a <i>Sub_Block</i> does not match the identifier <i>N</i> given at the start of the block.  This message is followed by another error message that shows the expected identifier <i>N</i> .
"not", "in" or "labelled" expected, at "T"	<i>Problem</i>	The assertion file should here have one of the keywords <b>not</b> , <b>in</b> , or <b>labelled</b> , instead of the token <i>T</i> . See the nonterminal <i>Loop_Property</i> in section 5.7 and the nonterminal <i>Call_Property</i> in section 5.7.
Quoted character expected , at "T"	<i>Problem</i>	The assertion file should have a character in single quotes here ( <i>'c'</i> ), instead of the token <i>T</i> .
Quoted string expected , at "T"	<i>Problem</i>	The assertion file should have a string in double quotes here (" <i>string</i> "), instead of the token <i>T</i> .
Repetition bounds on subprograms not implemented	<i>Problem</i>	The assertion file contains an assertion on execution count ( <b>repeats <i>N</i> times</b> ) for a subprogram. This assertion is not allowed for subprograms, only for loops and calls.

<b>Error Message</b>		<b>Meaning and Remedy</b>
Semicolon after callees expected, at " <i>T</i> "	<i>Problem</i>	The list of callee subprograms in a <i>Callee_Bound</i> (section 5.15) should be followed by a semicolon, instead of the token <i>T</i> .
Semicolon after "end call" expected, at " <i>T</i> "	<i>Problem</i>	The keywords <b>end call</b> should be followed by a semicolon, instead of the token <i>T</i> .
Semicolon after "end loop" expected, at " <i>T</i> "	<i>Problem</i>	The keywords <b>end loop</b> should be followed by a semicolon, instead of the token <i>T</i> .
Semicolon after "end subprogram" expected, at " <i>T</i> ".	<i>Problem</i>	A subprogram block lacks the terminating semicolon. Instead, the next token is <i>T</i> .
Semicolon after subprogram option expected, at " <i>T</i> "	<i>Problem</i>	A subprogram option lacks the terminating semicolon. Instead, the next token is <i>T</i> .
Semicolon expected after clause, at " <i>T</i> "	<i>Problem</i>	An assertion clause lacks the terminating semicolon. Instead, the next token is <i>T</i> .
Semicolon expected after global bound, at " <i>T</i> "	<i>Problem</i>	An assertion (on the values of a variable or a property) in the global context lacks the terminating semicolon. Instead, the next token is <i>T</i> .
Stack-name required	<i>Problem</i>	The program has more than one stack, but this <i>Stack_Bound</i> clause does not name the stack to which the assertions apply.
	<i>Reasons</i>	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.
	<i>Solution</i>	Add the stack name. The Application Note for your target processor shows the applicable stack names.
Subprogram address is invalid	<i>Problem</i>	An assertion tries to identify 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.
	<i>Solution</i>	Refer to the Application Note for your target processor and write the entry address in the proper form.
Subprogram name expected, at " <i>T</i> "	<i>Problem</i>	The assertion file should have a subprogram name (or address) here, in double quotes, instead of the token <i>T</i> .
Text at or after column <i>C</i> not understood: " <i>T</i> "	<i>Problem</i>	The text at or after column <i>C</i> on the current line in the assertion file is not a valid lexical "token" of the assertion language. The string <i>T</i> contains (part of) this text.
The integer literal " <i>n</i> " is not a valid number	<i>Problem</i>	The digit string <i>n</i> 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 "omit" property cannot be negated	<i>Problem</i>	A <i>Sub_Option</i> structure (section 5.5) uses the keyword <b>omit</b> and the keyword <b>not</b> in a combination that says that the subprogram in question is "not omitted".
	<i>Solution</i>	Not being omitted is the default condition for any subprogram, so remove the assertion as redundant.
The program has no stack named <i>S</i>	<i>Problem</i>	The stack name <i>S</i> given in this <i>Stack_Bound</i> clause does not match the name of any stack in the target program.



Error Message		Meaning and Remedy
	<i>Reasons</i>	Mistyped stack name, or a change in the compilation options such that the program no longer uses this stack.
	<i>Solution</i>	Refer to the Application Notes for your target processor to check the stack name.
The program has no stacks	<i>Problem</i>	The target program has no stacks (at all), so this <i>Stack_Bound</i> clause is not applicable.
	<i>Reasons</i>	Perhaps a change in the compilation options such that the program no longer uses any stacks (that Bound-T knows of).
	<i>Solution</i>	Refer to the Application Notes for your target processor.
The “unused” property cannot be negated or The “used” property cannot be asserted	<i>Problem</i>	A <i>Sub_Option</i> structure (section 5.5) uses the keywords <b>used</b> or <b>unused</b> and perhaps also the keyword <b>not</b> in a combination that says that the subprogram in question is “not unused”.
	<i>Solution</i>	Being used is the default condition for any subprogram, so remove the assertion as redundant.
“time” expected, at "T"	<i>Problem</i>	The keyword <b>time</b> is expected after the keyword <b>for</b> , to form the assertion <b>enough for time</b> . See section 2.13.
"times" expected, at "T"	<i>Problem</i>	This execution-count assertion should have the keyword <b>times</b> , instead of the token <i>T</i> , after the asserted execution time. It should be <b>repeats N times</b> .
Unrecognized property name: <i>P</i>	<i>Problem</i>	The assertion file names a target-specific "property" <i>P</i> but there is no such property for this target processor.
	<i>Solution</i>	Check the target-specific Application Note for the names of the properties for this processor.
Variable not found: <i>V</i>	<i>Problem</i>	The assertion file names a variable <i>V</i> , but the target program’s symbol table does not have a variable named <i>V</i> (in the implicit or explicit scope).
	<i>Reasons</i>	The name <i>V</i> may be mistyped; if the default scope is used perhaps another should be named explicitly; or the target compiler may have mangled the names.
	<i>Solution</i>	Check for typos. Check the target program’s symbol table using eg. <i>-trace symbols</i> or by dumping the file.
Variable name expected, at "T"	<i>Problem</i>	The assertion file should here have the name (or address) of a variable, instead of the token <i>T</i> .
Void bounds on stack final height ignored	<i>Problem</i>	The assertion bounds the <b>final</b> stack height to a void (null, empty) range. This is a contradiction and Bound-T ignores the assertion.
Void bounds on stack usage ignored	<i>Problem</i>	The assertion bounds the stack <b>usage</b> to a void (null, empty) range. This is a contradiction and Bound-T ignores the assertion.



Tidorum Ltd

Tiirasaarentie 32  
FI-00200 Helsinki, Finland  
[www.tidorum.fi](http://www.tidorum.fi)  
Tel. +358 (0) 40 563 9186  
VAT FI 18688130