Bound-T timing analysis tool

User Manual

Version 6.1
2007-10-31

Tidorum Ltd.

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

This document 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.

Document reference:	TR-UM-001
Document issue:	Version 6.1
Document issue date:	2007-10-31
Bound-T version:	3
Web location:	*http://www.bound-t.com/user-manual.pdf*

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

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

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.

Contents

Tables

Figures

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 Bound-T 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.

These cautions and remedies are discussed in more detail later in this manual.

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).

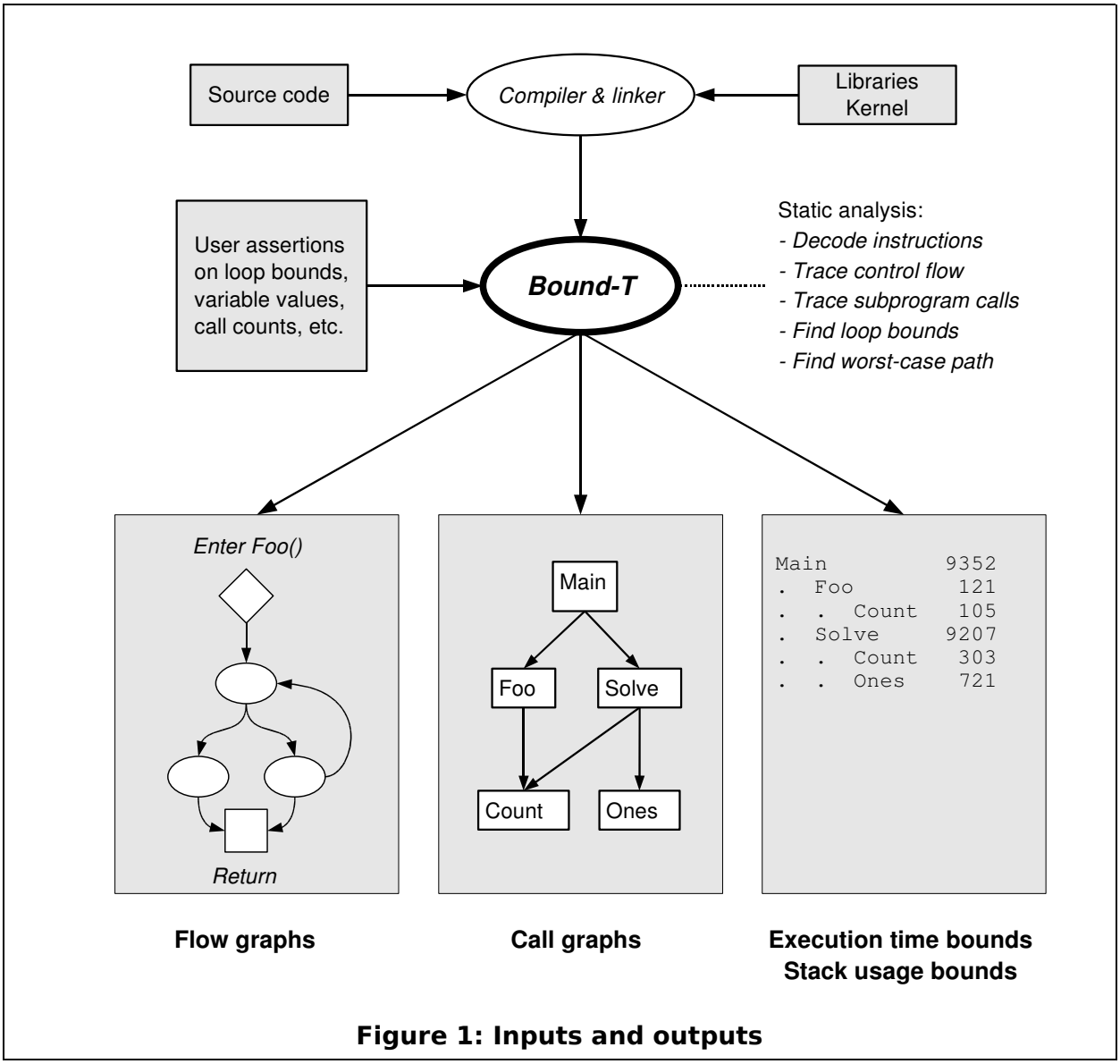

Figure 1: Inputs and outputs

1.2 Overview of this User Manual

What the reader should know

This User Manual is intended to explain what Bound-T can do and how Bound-T is used. The reader is assumed 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.

User guide

The rest of this manual consists of two parts: a *user guide* that introduces Bound-T in a tutorial and informal way, and a *reference manual* that explains all the details.

The user guide is organised into chapters as follows:

- Installing Bound-T is explained in Chapter 2. The computer on which Bound-T is installed and used is known as the *host* computer, and it may be quite different from the *target* computer on which the target program runs or will run.
- Using Bound-T is explained in Chapter 3, in a tutorial style that introduces the basic methods, options and results. The next-to-last section tells how to get started with the analysis of a real program and the last section explains stack-usage analysis.
- Chapter 4 suggests how to write programs that Bound-T can analyse.
- Chapter 5 shows how to write assertions to control and support Bound-T.

You may also have use for the glossary in Chapter 10.

Reference manual

The reference manual part of this document contains the following chapters:

- Chapter 6 lists and explains all command-line options and arguments for Bound-T.
- Chapter 7 explains all the outputs from Bound-T.
- Chapter 8 defines the syntax and meaning of the assertion language.
- Chapter 9 lists all warning messages and error messages, with explanations and advice on solving the problems.
- A glossary of terms in Chapter 10 concludes the manual.

1.3 Target-Specific Application Notes

Target processors

Bound-T is available for several target processors, with a specific version of Bound-T for each processor. All Bound-T versions are used in the same general way as explained in this User Manual. Additional information for specific targets is provided in separate Bound-T *Application Notes*.

Please refer to *http://www.bound-t.com/app_notes* for a list of the currently supported target processors and Application Notes.

Languages, compilers, kernels

Bound-T is largely independent of the programming language and execution environment of the target program. When necessary, separate Bound-T Application Notes advise on using Bound-T with specific target languages, compilers, real-time kernels or target operating systems.

Please refer to *http://www.bound-t.com/app_notes* for the currently available Application Notes.

1.4 Hard Real Time Programming Model

Execution Skeletons to order

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.

For an HRT program, Bound-T can generate so-called *execution skeletons* with detailed worst-case execution-time information as required by HRT schedulability analysis.

It's optional

Using Bound-T's HRT functions is quite optional. Bound-T can be used for non-HRT applications without knowing anything about the HRT model and how Bound-T supports this model.

It's described elsewhere

This manual describes how Bound-T is used in its basic mode, without the special HRT feaures. There is a separate manual (Tidorum document ref. TR-UM-002) that explains how to use Bound-T in HRT mode. See *http://www.bound-t.com/hrt-manual.pdf*.

2 INSTALLING BOUND-T

2.1 Delivery Medium and Instructions

Installation instructions are provided on the delivery medium or as hard-copy enclosed with the medium. Specific installation instructions are usually provided for each type of host platform (workstation and operating system).

On a typical platform, an installation of Bound-T for one target processor consists of a "bin" directory folder with three executable programs: Bound-T itself and two auxiliary programs. On Unix-like platforms the folder also contains two short executable shell-scripts that assist the auxiliary programs.

Support for each additional target processor type adds one executable of Bound-T itself to this folder. The auxiliary programs are the same for all targets.

The *dot* program for handling the graphical output is not included on the delivery medium. Install it from *http://www.graphviz.org*.

2.2 Disk Space Requirements

The disk space consumed by Bound-T depends on the host platform but allowing 20 MB should be ample for one type of target processor. Each additional target processor type needs approximately 10 MB more disk space.

2.3 Processor and Memory Requirements

Bound-T usually places about the same demands on the host workstation's processor speed and memory size as a typical compiler does.

For complex target programs the arithmetic analysis of loop-bounds may require a great deal of time and memory. However, experience shows that a normal desktop PC can handle many arithmetic analysis problems, and the larger problems are better dealt with by disabling arithmetic analysis with the option *-no_arithmetic* and asserting loop-bounds manually, or by simplifying the target program.

2.4 Host-Specific Usage Instructions

Advice on using Bound-T on various host platforms is given in separate platform-specific Application Notes included on the installation medium or enclosed as hard-copy.

2.5 Verifying the Installation

The installation instructions and host notes show how to get started by using Bound-T on examples provided with the installation. These examples are sufficient to verify that all components of Bound-T are functional.

3 USING BOUND-T

3.1 Preliminaries

Bound-T is used as an additional tool in a software development environment. It is not a stand-alone development tool, so you will need the usual kit of program editors, compilers and linkers.

To use Bound-T, you need:

1. The *target program* for which execution time bounds are wanted. The program must be provided in an executable form, compiled and linked for the target processor. Source code for the target program is not absolutely necessary but makes it easier to control Bound-T.
2. A *version of Bound-T* that supports the specific target processor and executable file format (e.g. COFF or ELF) and runs on your host platform (e.g. Linux or MS Windows).
3. Usually, knowledge about the *processing load* of the target-program, such as the maximum size of data structures, is also required.

If your linker produces a binary format that Bound-T does not support, note that the freely available GNU tool *objcopy*, a component of the GNU *binutils* tool-set, can be used to convert between various binary formats.

If the target program is concurrent (multi-threaded), a scheduling-analysis tool will also be useful, but is not required for using Bound-T (nor is one included). Typical scheduling analysis tools use Rate-Monotonic Analysis (RMA) or Deadline-Monotonic Analysis.

3.2 Running Bound-T

Bound-T is started with a command of the form

```
boundt <options> <target exe file> <subprogram names>
```

This command requests Bound-T to compute upper bounds for the worst-case execution time of the named subprograms within the given executable target program file. See chapter 6 for a full list of the command-line options.

This computation can either succeed fully automatically, or succeed only after some additional assertions are given. Chapter 5 shows how to write assertions when needed.

For an HRT-oriented analysis, Bound-T is started with a command of the form

```
boundt -hrt <more options> <target exe file> <TPOF name>
```

See section 1.4 for more about the HRT analysis mode.

The command name, written just `boundt` above, usually includes a suffix to indicate the target processor, for example `boundt_avr` names the Bound-T version for the Atmel AVR processor. Please refer to the relevant Application Note for the exact name.

3.3 Easy Examples : Loops

At last, some code!

To show what Bound-T can do, consider the following C function that computes the sum of the elements of a vector of floating-point numbers:

```
#define VECTOR_LENGTH 100

float sum_vector (float vector[])
{  int i;
   float sum;
   sum = 0.0;
   for (i = 0; i < VECTOR_LENGTH; i++)
      sum += vector[i];
   return sum;
}
```

Assume that this function is stored in the file *sum.c* and compiled and linked (together with some main function, not shown) into an executable program called *summer.exe*. Then, the Bound-T command

```
boundt summer.exe sum_vector
```

will display the worst-case execution time (WCET) of *sum_vector* as the last field of the output line:

```
Wcet:summer.exe:sum.c:sum_vector:3-10:1532
```

The WCET is given as the number of instruction cycles (1532 in this example); the corresponding number of seconds or microseconds of real time depends on the particular target processor and its clock frequency, as explained in the Application Note for this processor. The numbers in the preceding field, 3-10, are the source-code line-numbers of the subprogram.

How did it do that?

How does Bound-T compute the worst-case execution time? To use Bound-T effectively, it helps to know the general method, although it is not necessary to understand the details.

First, Bound-T reads in the executable program and uses the symbolic debugging information to find the entry point of the code of the *sum_vector* function. Then, Bound-T decodes the machine instructions to generate the control-flow graph of *sum_vector* and to locate the loop. Bound-T analyses the arithmetic of the looping code and infers that the loop is executed 100 times. Bound-T reports this by printing

```
Loop_Bound:summer.exe:sum.c:sum_vector:7-8:99
```

(The loop-bound is reported as 99 instead of 100 because Bound-T computes the number of times the looping code goes back to the start of the loop.) This defines the exact sequence of machine instructions that are executed in a call of *sum_vector*, and Bound-T simply adds up their execution time to give the WCET.

How does it know the execution time of the instructions?

For simple target processors each type of instruction has a fixed execution time – so many clock ticks. Sometimes the execution time depends on the sort of operands the instruction uses, for example memory operands taking longer than register operands, and Bound-T takes this into account.

On pipelined processors the execution time can depend on what other instructions are in the pipeline. Bound-T models the pipeline state to include this effect.

For some complex instructions, such as multiplication, division or floating point instructions, the execution time can vary depending on the values of the operands – the numbers being multiplied, for example. Bound-T usually assumes a worst-case execution time for such instructions.

Some target processors have several kinds of memory, at different memory addresses and with different access times. For example, on-chip memory is usually faster than off-chip external memory. For such processors, Bound-T analyses the address in each memory-accessing instruction to find the memory areas it can access and thus the access time. If the memory area remains unknown, Bound-T uses the access time for the slowest type of memory.

Some target processors have cache memory or branch prediction units or other types of acceleration mechanisms that store execution history and have a large effect on instruction execution time. Some target processors have several internal functional units that work in parallel, more or less asynchronously, also affecting the execution time. In its present form, Bound-T does not support such target processors.

Syntax is only sugar

Since Bound-T works on the binary, executable code and not on the source code, it's not picky about the way loops are written: **for**-loops, **while**-loops, **do-while**-loops or even **goto**-loops are all acceptable, as long as the loop is counter-based. For example, here is *sum_vector* with a **do-while**-loop:

```
#define VECTOR_LENGTH 100

float sum_vector (float vector[])
{  int i = 0;
   float sum = 0.0;
   do {
      sum += vector[i];
   } while (++i < VECTOR_LENGTH);
   return sum;
}
```

Goto is not harmful

Not only can loops be written with the **goto** statement, but the **goto** can also be used in other ways, for example to exit from a loop in the middle. The same holds for other control-flow statements such as the C statements **continue** and **break** and the Ada **exit** statement.

Any control structures in the programming language can be used, as long as the loops are counter-based and nicely nested within each other (in technical terms, the control-flow graph must be *reducible*).

3.4 Larger Examples : Calls

The root calls its children

In the above examples, the target subprogram did not call any other subprograms. Such calls are of course allowed, and Bound-T will automatically analyse the call graph and compute WCET bounds for all called subprograms, and finally for the "root" subprogram named as the argument on the command line.

If the WCET of a subprogram depends on the actual value of a parameter, Bound-T tries to compute the WCET separately for each call of a subprogram. This can extend progressively to calls within this call, and so on. An example follows.

What if vector-length is a parameter?

A flexible vector-summing function should have the vector-length as a parameter, for example called *n*:

```
float sum_vec_n (float *vector, int n)
{  int i;
   float sum;
   sum = 0.0;
   for (i = 0; i < n; i++)
     sum += vector[i];
   return sum;
}
```

Now the command

```
boundt summer.exe sum_vec_n
```

will report that *sum_vec_n* "could not be fully bounded". The reason is that Bound-T found no (reasonable) upper bound on the loop-counter *i*, because there is no (reasonable) upper bound on the parameter *n*. Bound-T points to the source of the problem as follows:

```
sum_vec_n
   Loop unbounded at sum.c:5-6
```

However, when the target program calls the *sum_vec_n* function, the call gives an actual value for the parameter, for example thus:

```
float sum_two (void)
{
   float v1[40], v2[1234];
   float sum1, sum2;
   ...
   sum1 = sum_vec_n (v1, 40);
   sum2 = sum_vec_n (v2, 1234);
   ...
   return sum1 + sum2;
}
```

If the above function is stored in *sum_two.c*, then compiled and linked into *summer.exe*, the command

```
boundt summer.exe sum_two
```

will compute a WCET, for example:

```
Wcet:summer.exe:sum_two.c:sum_two:4-12:30607
```

Although Bound-T again failed to bound the loop in *sum_vec_n* as such, it repeated the analysis for each of the two calls of *sum_vec_n* in *sum_two*. With *n* known to be 40 or 1234, respectively, Bound-T could compute loop-bounds and WCET for each call and thus also the total WCET for *sum_two*.

The results for each call are reported in the following form, slightly abbreviated here for reasons of line-length:

```
Loop_Bound:summer.exe:sum.c:sum_two=>sum_vec_n:5-6:39
Loop_Bound:summer.exe:sum.c:sum_two=>sum_vec_n:5-6:1233
Wcet_Call:summer.exe:sum.c:sum_two=>sum_vec_n:4-10:628
Wcet_Call:summer.exe:sum.c:sum_two=>sum_vec_n:4-10:18904
```

Loops within loops

Nested loops are handled in the same fashion, for example:

```
float sum_matrix (float *matrix[], int m, int n)
{  int i, j;
   float sum;
   sum = 0.0;
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         sum += (matrix[i])[j];
   }
   return sum;
}
```

Since the loop-limits *m* and *n* are again parameters, Bound-T cannot compute a WCET for this subprogram as such. Once *m* and *n* are given values in a call of *sum_matrix*, the WCET can be computed just as in the earlier example with *sum_vec_n*.

3.5 Harder Examples : Assertions

While-loops may be confusing

Next, consider the more complex C function *binary_search* that looks up a value in a sorted vector using a divide-and-conquer approach:

```
int binary_search (int *vector, int val)
{  int low, high, mid;
   low = 0;
   high = VECTOR_LENGTH - 1;
   while (low <= high)
   {
      mid = low + (high - low) / 2;
      if (vector[mid] == val)
         return mid;
      else if (vector[mid] < val)
         low = mid + 1;
```

```
      else
        high = mid - 1;
    }
    return -1;
  }
```

If this function is stored in the file *bins.c* and compiled and linked into the executable program file *bins*, the Bound-T command

```
boundt bins binary_search
```

reports that the subprogram could not be fully bounded because no bounds were found for the **while**-loop.

When Bound-T analysed the control and data flow of *binary_search*, it could not find any variables that act as loop counters with simple initial and final values and simple increments. For a non-trivial algorithm such as binary search this is not very surprising.

Assertions make it clear

What can be done to work around this problem? The Bound-T user must help out by *telling* Bound-T the maximum number of times the loop can repeat. This is done with an assertion placed in a separate assertion file. Chapters 5 and 8 will explain this fully, but here is how to do it in this example.

Assuming that *VECTOR_LENGTH* is 100, the maximum number of iterations is 7. The assertion takes the form

```
subprogram "binary_search"
  loop repeats <= 7 times; end loop;
end "binary_search";
```

If this assertion is placed in a file *prog.bta* and Bound-T is run with the command

```
boundt -assert prog.bta bins binary_search
```

the analysis succeeds and Bound-T displays the WCET bound, for example as

```
Wcet:bins:bins.c:binary_search:4-18:129
```

Counters make it clear, too

Another way to help Bound-T find bounds on loops is to add loop-counters when there are none to start with. For example, the *binary_search* function could be changed as follows:

```
  int count = 0;
  ...
  while (low <= high)
  {
    mid = low + (high - low) / 2;
    ...
    count += 1;
    if (count > 7) break;
  }
```

Now the loop-repeat condition contains an explicit limit on a counter value, and Bound-T can compute a WCET bound without any help from assertions. The limit can be made a parameter of the subprogram, of course, instead of a constant (7) as in this example.

Eternal loops take a little longer

Much has been said about finding bounds on the number of iterations of loops. But what if the program contains an eternal loop? We call a loop *eternal* if it 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. To analyse the execution time of an eternal loop you must assert an assumed number of repetitions of the loop. This will be explained in section 5.17.

3.6 What Bound-T Can Do

This section outlines Bound-T's current abilities, which are, of course, constantly being extended. This is generic information, applicable to all target processors. Target-specific Application Notes define more precisely Bound-T's abilities and limits for each target.

Control flow tracing

Bound-T can decode all target processor instructions. Bound-T can analyse the control-flow and call-graph of any subprogram that follows the processor's calling standard and where the destination addresses of each branch are statically defined in the code.

For normal branches, which the compiler generates for conditional or looping statements, or for calls that give the real name of the callee, the destination address is usually an immediate literal in the branch instruction, and so is statically defined.

Note that for a conditional branch, although the *possible* destinations must be known statically, the *condition* (boolean expression) that selects the actual destination can be dynamically computed at run-time, and usually is.

Switches and cases

Large and dense switch/case statements often use a simple form of dynamic destination addressing in which the case-selector is used as an index to a table that gives the starting address of the corresponding branch of the switch/case statement. Bound-T contains specific data-flow analysis to derive static bounds on the value of the case-selector and thus find the case-address table and the possible destinations of such a branch.

However, the current version of Bound-T is limited in the kind of switch/case code it can analyse. Refer to the Application Notes for particular target processors and target compilers for details.

Counter-based loops

As the two examples in section 3.3 showed, sometimes Bound-T can analyse the target subprogram well enough to arrive at a WCET bound automatically, but sometimes the user must provide assertions to guide and constrain the analysis. The most important factor that decides the need for user assertions is the complexity of the loop-termination conditions.

The data-flow and loop-analysis algorithms currently implemented in Bound-T are designed to handle *counter-based* loops automatically. A counter-based loop is a loop that always increments (or decrements) a counter variable on each iteration, and terminates when the counter becomes greater than (or less than) a limit.

It is not necessary that a counter-based loop is actually written as a **for**-loop in the target program source code; what matters is the way the logical exit condition is written, and the way the counter variable is updated.

The counter's step (increment or decrement) can be positive or negative, but it must have the same sign for all loop iterations. The absolute value of the step can vary from one iteration to another, as long its lower bound is nonzero. The initial and final values of the counter need not be known exactly; as long as they are bounded in suitable directions, Bound-T will compute the worst-case number of loop iterations, using the value of the step that has minimum absolute value, and those extreme values of the initial and final values that are farthest from each other.

To determine a loop-counter's initial value, step, and limit value, Bound-T can follow any computation in the target program that uses integer addition, subtraction and multiplication by a constant. Values of parameters are propagated into calls, but not in the other direction (from callees to callers).

When Bound-T stumbles

As already said, deducing the worst-case execution time of an arbitrary program is unsolvable in principle, so any tool like Bound-T must fail when the target program is complex enough. When Bound-T cannot handle a problem automatically, it is usually possible to write assertions that let Bound-T solve the problem. Of course, in this case the validity of Bound-T's results depends on the validity of the assertions, which is the user's responsibility.

An alternative solution is to change the target program to make it easier to analyse. For example, if the target program contains a **while**-loop that Bound-T cannot find bounds for, simply adding an iteration counter and limit to the loop will solve the problem (and perhaps make the target program more robust, too).

Chapter 4 advises on programming styles that help Bound-T analyse the program.

3.7 What Bound-T Cannot Do

The analysis algorithms in Bound-T have been chosen and tuned to handle many forms of loops and other program structures automatically. However, sometimes the target program is too complex or inscrutable for these algorithms. Here is a list of things Bound-T cannot currently do, ordered approximately from the most common to the least common problems. Fortunately, most problems can be worked around as explained below.

Uncounted loops

Bound-T cannot infer the maximum number of loop repetitions for loops that do not have explicit counters and limits on the counters. As in the examples in section 3.5, this can be worked around with assertions or with source-code changes.

Multiplication, division etc

The method Bound-T uses to analyse the loop-counter computations handles only addition and subtraction of variables and multiplication by constants. If the computation of a loop-counter's initial value, step or limit-value involves any arithmetic operation beyond this, such as the multiplication of two variables, the value becomes "unknown" to Bound-T.

The same work-arounds apply as for loops without explicit counters.

Multiple-precision arithmetic

Bound-T bases its analysis on a model of the native instructions in the target processor, using the native number of bits per value (word length). For processors with a small word length, such as 8 bits, the compiler (or assembly-language programmer) has to construct wider arithmetic from two or more 8-bit parts and two or more 8-bit instructions connected by some form of carry flag. Bound-T usually does not model such multiple-precision arithmetic which means that it usually cannot bound loops that use multiple-precision counters, for example 16-bit counters on an 8-bit machine.

The target-specific Application Notes explain the types of arithmetic and operands that Bound-T supports for the target processor.

Aliasing and pointer chasing

If a variable is not directly assigned within a loop, Bound-T assumes that it is unchanged (invariant) throughout the loop. This assumption may be wrong if the variable is accessed indirectly through pointers, that is, via a memory reference with a dynamically computed address. The pointer-access may be explicit in the source program, or it can result from implicit aliasing between parameters that are call-by-reference.

Bound-T records which global variables are directly assigned in each subprogram. Each call of the subprogram is then considered to assign unknown values to these global variables, which is important if the call is in a loop that uses this global variable in its loop-counter computation. However, if the called subprogram assigns the global variable via a pointer, Bound-T does not include the assignment in the analysis of the loop-counter arithmetic, which may lead to wrong results.

Although Bound-T has an option (*-warn access*) to emit a warning messages for all dynamic, indirect memory accesses that it cannot resolve, most of these are just array accesses and are usually irrelevant to loop counters. Thus, with the present version of Bound-T, the user is responsible for avoiding aliasing that could distort the analysis of loop bounds.

Overflow in the target program

The method Bound-T uses to analyse the loop-counter computations assumes that these computations do not overflow when the target program is executed. If overflow occurs, the bounds computed by Bound-T may be incorrect, or Bound-T may fail to find bounds at all.

Note that this refers to overflow in some future execution of the target program, not to overflow in Bound-T's own computations, which are checked against overflow. We believe that the component tools, *oc* and *lp_solve*, also contain internal overflow checking.

The only work-around is to change the target program to guard against overflow, and to not use overflow on purpose in loop-counter computations. It is feasible to extend Bound-T to consider overflow, and we are studying how to do it efficiently.

Unsigned arithmetic

The method Bound-T uses to analyse the loop-counter computations assumes that the variables can take both positive and negative values and that there are no wrap-around effects from unsigned arithmetic. For example, in common programming languages decrementing an unsigned integer variable that has the value zero gives a large positive value of the formand not the value -1. Such wrap-arounds are similar to overflow and are currently not handled by Bound-T.

Usually, the work-around is to use only signed variables and signed arithmetic instructions in the target program's loop counters. However, check with the Application Note for the target processor as there may be target-specific solutions. It is quite feasible to extend Bound-T to include unsigned arithmetic and this is planned for future versions. It may already be implemented for specific target processors; again, please check the Application Note for your target.

Jumps and calls via pointers

Except for some switch/case statements and some locally dynamic calls, Bound-T cannot handle a branch to an address that is not known until run-time. The most common cause of such dynamic branches is calling a subprogram via a pointer. This restriction also excludes object-oriented programming with dynamically bound methods such as C++ virtual functions.

When Bound-T finds a dynamic call that it cannot resolve, it issues an error message and handles the call as if it took no time and had no effect. If you know which subprograms can actually be called by this call, you can give Bound-T this information as an assertion, or you can use Bound-T to find the maximum WCET of these potential callees and add it to the WCET that Bound-T reports for the caller.

When Bound-T finds a dynamic jump that it cannot resolve, it issues an error message and handles the jump as if it were a return instruction. That is, the WCET reported for the subprogram that contains the jump does not include the execution after the jump. If you know the possible targets of the jump, you may be able to use Bound-T to find the maximum WCET of the code after the jump and add it to the WCET that Bound-T reports for the subprogram that contains the jump. However, it is probably easier to change the target program to get rid of the dynamic jump.

For dynamic calls and jumps, the closest alternative program structure is to write a switch/case statement or a nest of if-then-else statements in which the various branches contain static calls or jumps to all the possible callees or jump targets.

Exceptions and traps

Many target processors and some programming languages perform automatic run-time checks on the computation, for example for numerical errors such as division by zero or for logical errors such as array index out of bounds. When a check fails the normal program flow is interrupted and execution continues at the address of the handler routine for the exception or trap. Execution may or may not return to the original program flow. Obviously this changes the execution time, perhaps radically.

There are basically two kinds of traps: *hardware* traps and *software* traps.

For hardware traps the check and possible branch to a trap handler are an implicit part of normal instructions. For example, the processor could be designed so that all addition instructions check and trap on overflow.

For software traps the check or the branch to the trap handler are programmed by specific instructions. For example, most processors are designed to that an addition overflow just sets an overflow flag. To take an overflow trap the addition instruction must be followed by a conditional branch instruction that branches to the trap handler if the overflow flag is set.

Bound-T generally assumes that no hardware traps occur in the execution under analysis. Thus, the WCET bound does not include hardware traps.

Software traps, in contrast, appear to Bound-T as normal program flow and are thus included in the analysis. However, the address of the trap handler is usually not given statically but in some kind of "trap table" or "vector table" which means that the trap handler is located via a pointer and Bound-T may be unable to find the handler for analysis.

Irreducible flow graphs

Bound-T can handle only control-flow graphs that are *reducible*. A reducible control-flow graph is one in which each loop is entered at a single point and any two loops are either nested one within the other or are entirely separate (no shared instructions). It is commonly observed that nearly all programs are reducible in the source code form, but sometimes the compilers emit irreducible object code, perhaps due to optimisation. Assembly-language subprograms such as run-time library routines are sometimes also irreducible, perhaps due to manual optimisation.

There is currently no work-around for this limitation, other than reducing the level of optimisation or changing the way the offending loops are coded in the target program. Any subprogram with an irreducible flow-graph must be excluded from the analysis by assserting its WCET as shown in section 5.5.

Recursion

Bound-T cannot analyse recursive calls. Bound-T builds WCET bounds in a bottom-up way from the lowest-level subprograms (leaf subprograms) towards higher-level subprograms (root subprograms). If the subprograms are recursive this bottom-up method does not work and Bound-T reports an error. However, you can analyse recursive programs by using assertions to slice the call graphs into non-recursive parts. This will be explained in section 5.18 in connection with the assertion language.

3.8 Approximations

When Bound-T computes upper bounds on worst-case execution time, it uses three types of approximation, corresponding to three sources of unknown variation in execution time.

Instruction-level approximations

The execution time of some instructions in the target processor may be inherently variable. For example, the time can depend on the data being processed, or on the history of recently executed instructions and memory accesses. For each instruction Bound-T uses an upper bound on the execution time that takes into account some of this variation for the context of this instruction. The details depend on the target processor but in general the analysis includes pipeline effects but not cache effects.

Although these dynamic features are increasing strongly in high-end processors, many smaller, embedded processors are still quite deterministic, with fixed instruction-execution times. The Application Note for a particular target processor will describe the instruction-level approximations in detail.

Loop-count approximations

The bounds on loop iterations computed by Bound-T are upper bounds. Early exits (breaks) from loops can make the real number of iterations smaller.

A similar approximation occurs for "non-rectangular" nested loops where the limits of the inner loop depend on the index of the outer loop. A typical example is a pair of loops that process the upper (or lower) triangle of an $N \times N$ square matrix. Here the current version of Bound-T can only give, automatically, an N^2 bound on the number of executions of the inner loop-body. However, the real bound, $N(N+1)/2$, can be asserted, if the inner loop-body contains a feature that can be used to identify it.

Feasible path approximations

In a sequence of conditional statements, loops or other control structures, the several conditional expressions are sometimes correlated so that only a subset of paths can actually occur. For example, this happens if a conditional of the form "**if** *A*" is followed by a conditional of the form "**if not** *A*", where the value of *A* is unchanged.

Bound-T is generally not able to correlate the conditions, but will compute the WCET over all apparently possible paths, allowing any combination of condition values, including logically impossible combinations. If the branches have very different execution times, a considerable over-estimate in WCET can result.

In some cases the approximation can be corrected with assertions. For example, if the code is in a loop, and the branches can be identified by some of their features (such as the calls they contain), one can assert an execution-count bound on certain branches that is less than the number of iterations of the loop. This forces Bound-T to "by-pass" these branches for a selectable fraction of the loop iterations. Section 5.4 shows some examples.

3.9 Context-Dependent Execution Bounds

The inputs of a subprogram

Most of your subprograms probably have parameters and the execution time usually depends more or less strongly on the actual values of those parameters. Perhaps the subprogram also uses global variables that influence the execution time.

For brevity, we use the term *input variables* or simply *inputs* for all the parameters and global variables that influence the execution bounds of a given subprogram: the bounds on execution time or stack usage (stack analysis is described in section 3.11). Some subprograms have no inputs and thus have constant execution bounds, but most do have some input variables. Take the following Ada subprogram as an example:

```
procedure Nundo (X : Integer; N : Integer) is
begin

   if X > 10 then
     Start_Engine;
   end if;

   for I in 1 .. N loop
     Mark_Point (I);
   end loop;

end Nundo;
```

The value of the parameter *X* influences the execution time of *Nundo* because the *Start_Engine* subprogram is called only for some values of *X*. The value of the parameter *N* influences the execution time because it sets the number of iterations of the loop that calls *Mark_Point*.

Each call of the subprogram may have different input values and may thus have a different execution time and stack usage. Can Bound-T take this into account? Yes, in some ways, but it depends a lot on how your program computes and passes parameter values and how the subprograms use parameters. This section tries to explain how and when Bound-T can find such input-dependent execution bounds.

Essential inputs

Some inputs to a subprogram are *essential* for the analysis in the sense that their values *must* be known in order to compute execution bounds. Consider again the example subprogram *Nundo* above. If the value of *N* is unknown then the number of loop iterations is unknown and there is no upper bound on the execution time. (One could argue that *N* is at most *Integer*'*Last*, the largest possible value of type *Integer*, so the loop can repeat at most *Integer*'*Last* times. But this upper bound is probably a huge overestimate and we ignore it.)

A value (or an upper bound) on *N* is thus needed to get an upper bound on the execution time of *Nundo*. Therefore *N* is an *essential* input variable for *Nundo*.

The same cannot be said for the *X* parameter. If the value of *X* is unknown we can simply assume the worst case, include the call of *Start_Engine* in the analysis, and get an upper bound on the execution time of *Nundo* that is valid for all values of *X*, even if it is overestimated for values of *X* less or equal to 10. Thus *X* is not an essential input for *Nundo*.

Bound-T tries to find input-dependent execution bounds for a subprogram only when the subprogram has some essential inputs. More on this later, also to show that the classification of inputs into essential or non-essential is not always so clear-cut as in the *Nundo* example.

Calls, call sites, and call paths

To explain how Bound-T does input-dependent analysis we have to define some terms that separate the static and dynamic aspects of subprogram calls.

- A *call site* is point (an instruction) in the target program that calls a subprogram (the *callee*) from within another subprogram (the *caller*). When there is no risk of confusion we will use the shorter term *call*.

Call sites are a static concept; we are not yet talking of the dynamic execution of the call when the program is running. A call site is identified by the address of the instruction that transfers control from the caller to the callee. Each call site has a *return poin*t that is usually the next instruction in the caller.

Why talk about calls and call sites here? Because the calls pass parameter values – inputs – to the callee subprogram. Different calls (call sites) can pass different values. We can hope that an analysis of the call, and of the code that leads to the call, will reveal bounds on parameter values that we can use to find bounds on the execution of the callee. However, different executions of the same call site can pass different parameter values, so some over-estimation may remain even for call-site-specific execution bounds.

Sometimes parameter values are set in some high-level subprogram and then passed through several levels of calls until they reach the subprogram for which they are esential inputs. This motivates:

- A *call path* is a list of zero or more call sites such that the callee of a call in the list is the caller of the next call in the list (if any).

- The *depth* of a call path is the number of call sites in the list.

A call path is still a static concept; we are not yet talking of the dynamic executions of these calls. In particular, even if some call on the path lies within a loop and so can be executed many times the call path does not distinguish between the iterations of the loop.

For example, assume that subprogram *A* contains a call to subprogram *B* which contains three calls, one to subprogram *C* and two to subprogram *D*. Assume further that *A* also calls *C* and *C* also calls *D* so that the call-graph looks like the diagram in Figure 2. The calls are numbered *call1* to *call6* for identification.

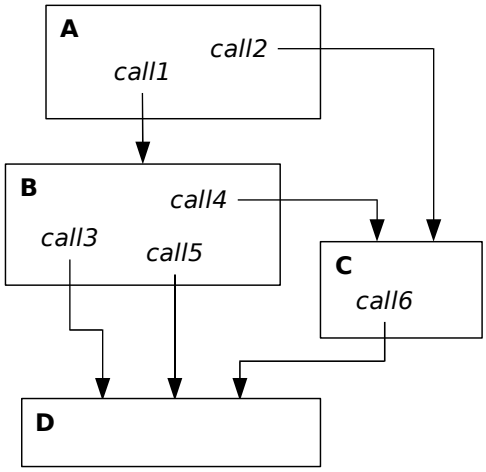

Figure 2: Example of calls and call paths

As the figure shows there are four ways to reach subprogram *D* from subprogram *A*, depending on which calls are used:

– from *A* through *call1* to *B*, then through *call3* to *D*,

– from *A* through *call1* to *B*, then through *call5* to *D*,

– from *A* through *call1* to *B*, then through *call4* to *C*, then through *call6* to *D*,

– from *A* through *call2* to *C*, then through *call6* to *D*.

Note that a call path is certainly not a complete definition of how execution goes from the first caller to the last callee – from *A* to *D* in the example. As already said, the call path concept ignores looping. It also ignores the fact that there are often several ways (execution paths) to go from the entry point of a subprogram to a given call within the subprogram – all such ways give the same call path.

Executions and contexts

In contrast to calls and call paths, the *execution of a subprogram* is a dynamic concept: during the execution of the program, control reaches this subprogram, the code in the subprogram is executed, and the subprogram (usually) returns to its caller. The *execution of a call* means that control reaches the call and then passes to the callee which is executed. The execution of a call path means that control reaches the first call on the path, passes to the callee, and in the callee to the second call on the path, and so on until control reaches and executes the last call on the path.

- The *full context of a subprogram execution* is the call path that was executed to reach the subprogram, starting from a given root subprogram.

For the example in Figure 2, if subprogram *A* is taken as the root subprogram then subprogram *D* can be executed in four different full contexts:

– (*call1*, *call3*)

– (*call1*, *call5*)

– (*call1*, *call4*, *call6*)

– (*call2*, *call6*).

The main point here is that Bound-T groups subprogram executions by their context. Thus we can find different execution bounds for each call path leading to the subprogram. However, we hope to find execution bounds without having to consider the *full* call path from the root, and so we define:

• A *suffix context of a subprogram execution* is any suffix of the full context. That is, any call path that ends with a call to this subprogram, or the null call path (a list of no calls).

For the example in Figure 2 the subprogram *D* has the nine suffix contexts listed in the table below in order of increasing depth. The table also shows depth of the suffix context and the full contexts that match the suffix context. Real programs usually have more full contexts per suffix context than this small example has.

Table 1: Example of suffix contexts and full contexts

Suffix context for *D*	**Depth**	**Matching full contexts for *D***
null call path	0	all full contexts (all executions of *D*)
call3	1	(*call1*, *call3*)
call5	1	(*call1*, *call5*)
call6	1	(*call2*, *call6*) (*call1*, *call4*, *call6*)
(*call1*, *call3*)	2	(*call1*, *call3*)
(*call1*, *call5*)	2	(*call1*, *call5*)
(*call2*, *call6*)	2	(*call2*, *call6*)
(*call4*, *call6*)	2	(*call1*, *call4*, *call6*)
(*call1*, *call4*, *call6*)	3	(*call1*, *call4*, *call6*)

The execution bounds that Bound-T computes for a subprogram always come with a suffix context such that the bounds are valid for all executions of the subprogram in this context.

Thus, if Bound-T computes execution bounds for subprogram *D* in the example above then the bounds apply as follows, depending on the suffix context of the bounds:

– If the context is null, the bounds are valid for all executions of *D* whatever the full context of the execution. For this reason the null context is also called the *universal* context and such bounds are *universal* bounds or *context-free* bounds.

– If the context is *call3*, the bounds are valid for any execution of *D* from *call3*. In the example the only full context that matches this suffix context is (*call1*, *call3*).

– If the context is *call6*, the bounds are valid for any execution of *D* from *call6*. In the example there are two full contexts that match this suffix context: (*call1*, *call4*, *call6*) and (*call2*, *call6*).

– If the context is (*call4*, *call6*) the bounds are valid for any execution of *D* from *call6* within an execution of *C* from *call4*. In the example the only full context that matches this suffix context is (*call1*, *call4*, *call6*).

– And so on for the other possible contexts of *D*.

First we ignore inputs

For each subprogram Bound-T first tries to find execution bounds in the null context – context-free bounds that apply universally to all executions of the subprogram.

The subprogram is analysed in isolation, not in the context of any particular call or call path. The values of the inputs are then generally unknown. However, the analysis uses all assertions that apply to input variables or to variables defined and used within the subprogram, as long as the assertions apply universally (to all executions of the subprogram and not only to a particular call).

If that fails, we look deeper ... and deeper ...

When Bound-T cannot find context-free execution bounds on a subprogram it analyses the subprogram in ever deeper suffix contexts until it finds execution bounds, and then it stops. In other words, when the subprogram has some essential inputs (with unknown values in the null context) Bound-T tries deeper contexts until the context defines values (or sufficient bounds) for the essential inputs at each (feasible) call of the subprogram.

The context-free analysis in Bound-T traverses the call-graph in *bottom-up* order. That is, we first analyse the leaf subprograms – those subprograms that do not call other subprograms – then subprograms that only call leaf subprograms, and so on to higher levels in the call-graph up to the root subprogram(s). If a callee subprogram has context-free execution bounds these bounds are thus known when the caller is analysed.

When Bound-T analyses a caller subprogram it finds all calls from this subprogram to callees that do *not* have execution bounds (whether context-free bounds or bounds in the context of this call); these are known as *unbounded calls*. For each unbounded call Bound-T uses the analysis of the caller to find bounds on the inputs to the callee. If some such bounds exist Bound-T re-analyses the callee in the context of these bounds, that is, using these bounds as the initial state on entry to the callee.

When an unbounded call leads to the re-analysis of the callee Bound-T may find further unbounded calls in the callee, leading to re-analysis of their callees, and so on. Thus context-dependent re-analysis spreads *top-down* in the call-graph.

Call-specific assertions on variable values can help context-specific analysis by directly defining input values for the subprogram being analysed (when the assertion applies to a call of this subprogram, the last call in the context) or indirectly by defining values on other variables that enter the computation of the input values (when the assertion applies to some other call in the context).

The command-line parameter *-max_par_depth* defines the largest context depth that Bound-T tries. If a subprogram has some full context such that Bound-T finds no execution bounds for a suffix context of depth *max_par_depth* then Bound-T emits an error message and considers the subprogram's execution unbounded in this context. In other words, *max_par_depth* sets an upper bound on the number of call levels through which Bound-T tries to find values or bounds on essential inputs.

How it works in the example

This subsection shows step by step how the context-free and context-dependent analysis works for the example program shown in Figure 2. The description is long; if you feel that you understand the idea you can skip this subsection.

The analysis of the program in Figure 2 proceeds as follows, assuming that the root subprogram is *A*. Subprograms are analysed in bottom-up order in the call-graph. Thus *D* is the first subprogram to be analysed, followed by *C*, *B*, and finally *A*.

1. First Bound-T looks for context-free execution bounds on *D*. If this analysis succeeds Bound-T uses these bounds for all calls of *D*. That is, it uses these context-free bounds on *D*'s execution time (and/or stack usage) for *call3* and *call5* in the analysis of subprogram *B* and for *call6* in the analysis of subprogram *C*. In this case the analysis of *D* stops here and Bound-T goes on to analyse *C*, *B*, and *A* in that order. But the other case is more interesting and the analysis then proceeds as follows.
2. If Bound-T does not find context-free bounds on *D* it postpones further analysis of *D* until the analysis of the direct callers of *D*: subprograms *B* and *C*. That is, Bound-T will re-analyse *D* in the suffix contexts *call3*, *call5* and *call6*, all of depth one.
3. Next Bound-T looks for context-free bounds on *C*. *C* contains *call6* which is an unbounded call (because the callee, *D*, has no context-free bounds). Therefore Bound-T tries to find bounds on the inputs to *D* at *call6*. If it finds some such bounds:
   - Bound-T re-analyses *D* in the context of *call6* (a depth-one context). If it finds execution bounds they become the definitive bounds for *call6*; Bound-T uses these bounds for all executions of *call6*. Assume otherwise, that *call6* remains an unbounded call. This means, firstly, that Bound-T will try deeper contexts for this call (if *max_par_depth* permits) and secondly that the context-free analysis of *C* fails. Accordingly Bound-T postpones further analysis of *C* (and thus further analysis of *call6*) until the analysis of the direct callers of *C*: subprograms *A* and *B*.
4. Next Bound-T looks for context-free bounds on *B*. Here the unbounded calls are *call3* and *call5* to *D* and *call4* to *C*. Assuming that some callee inputs are bounded at each call, then:
   - Bound-T re-analyses *D* in the contexts of *call3* and *call5* (both are depth-one contexts). Assume that *call3* gets execution bounds but *call5* does not. Bound-T then uses these bounds on *call3* as the bounds on all executions of *call3*, that is, it does not try to analyse deeper contexts that lead to *call3*. Since *call5* remains unbounded the context-free analysis of *B* fails.
   - Bound-T re-analyses *C* in the context of *call4*. *C* contains *call6* which is still unbounded, so Bound-T again tries to find bounds on the inputs for *D* at *call6* within this new analysis of *C*; if some bounds are found, Bound-T re-analyses *D* with these bounds, that is, in the depth-two context (*call4*, *call6*). If this analysis finds execution bounds Bound-T will use these bounds on *D* for every execution of *D* that has the suffix context (*call4*, *call6*). If Bound-T also finds execution bounds on all other parts of *C* it will use these bounds on *C* for every execution of *C* that has the suffix content *call4*. But let us again assume the harder case where *call6* remains unbounded which also means that *C* remains unbounded in the context *call4*.

To summarise the situation at this point in the analysis: We found no context-free execution bounds on *B*, *C*, or *D*. We found execution bounds on *D* in the context *call3* but not in the contexts *call5* and *call6*. We found no execution bounds on *C* in the context *call4* and have not yet analysed *C* in its other depth-one context, *call2*. We have not yet tried to analyse *A*, but *A* is next in the bottom-up order of the call-graph, so here we go:

5. The next subprogram to be analysed is *A*, the root subprogram. Root subprograms can only have universal, context-free execution bounds (unless there are several root subprograms and some root calls another root, which is unusual). The unbounded calls within *A* are *call1* and *call2*. Assuming that Bound-T finds bounds on the inputs for *B* and *C*, respectively, at these calls Bound-T re-analyses the callees in these contexts, so:

   - *B* is re-analysed in the context of *call1*. The unbounded calls in *B* are *call5* and *call4*. Assuming that Bound-T finds bounds on the inputs for *D* and *C*, respectively, at these calls, then:

     - Bound-T re-analyses *D* in the context (*call1*, *call5*). Assume that it finds execution bounds on *D* in this context.

     - Bound-T re-analyses *C* in the context (*call1*, *call4*). *C* contains *call6* which is still unbounded in this context so Bound-T tries to find more bounds on the inputs for *call6*, now from the deeper context (*call1*, *call4*). Assuming that such bounds are found:

       - Bound-T re-analyses *D* in the context (*call1*, *call4*, *call6*). Assume that it finds execution bounds on *D* in this context. This makes *call6* bounded in the context (*call1*, *call4*).

       Assume that all other parts of *C* are also bounded in the context (*call1*, *call4*).

     Thus all calls in *B* are bounded in this context (*call3* was bounded earlier, in the null context for *B*, and *call4* and *call5* were bounded in the present context, *call1*).

     Assume that all other parts of *B* are also bounded in this analysis. Thus we have execution bounds on *B* in the context *call1* and so *call1* is bounded in *A*.

   - *C* is re-analysed in the context of *call2*. *C* contains *call6* which is still unbounded in this context (the execution bounds that we found, above, on *call6* apply to a different context (*call1*, *call4*) for *C*). Assuming that Bound-T finds bounds on the inputs for *D* at this call:

     - Bound-T re-analyses *D* with all the input bounds collected from the context (*call2*, *call6*). Assume that it finds execution bounds on *D* in this context.

     Assume that all other parts of *C* are also bounded in this analysis. Thus we have execution bounds on C in the context *call2* and so *call2* is bounded in *A*.

   Thus, all calls within *A* have execution bounds. Assuming that all other parts of *A* are also bounded we finally get execution bounds on *A*, the root subprogram.

This method of re-analysing subprograms in ever deeper contexts is evidently inefficient if many subprograms need a deep context for their analysis. The method works well enough when most subprograms get context-free bounds or need only shallow context for their analysis. But the main importance of the method for you, as a user of Bound-T, is not its computational efficiency but how the results of the analysis depend on the properties of the target program under analysis. The rest of this section focusses on that question.

What does this mean?

The main things to remember from the above discussion are:

- Bound-T only tries to find context-dependent bounds when it fails to find context-free bounds, that is, when some essential input values are unknown without context.
- Bound-T explores contexts only as far (as deeply) as is necessary to find execution bounds, that is, until the context defines the essential input values.

- However, when Bound-T does make a context-specific analysis it tries to find context-specific values or bounds on *all* inputs to the subprogram, not only on the essential inputs, and uses all such values or bounds in the analysis.

The rest of this section tries to explain what this means in terms of the design of the target program, using examples.

Examples of essential and non-essential inputs

First some examples to illustrate when inputs are essential and when not. There are some complex cases in which even this decision may depend on context so the classification of inputs into essential and non-essential is a simplification of reality.

- The conditions in **if**-**then**(-**else**) statements are usually not essential.

An input that appears in an **if**-**then**(-**else**) condition can have a large effect on the execution time when one branch of the conditional statement has a much larger execution time than the other branch, but this does not make the input essential. For an example, see the *Nundo* subprogram, above, and its parameter *X*.

- Similarly, the selector (index) of a **switch**-**case** statement is usually not essential.

A switch-case selector can have a large effect on the execution time when the different cases have very different execution times. But this does not make the selector essential.

- An input that controls a conditional statement or a switch-case statement can determine which other inputs are essential.

For example, consider the following variation of the *Nundo* procedure where the changes are shown in **bold** style:

```
procedure Nundo2 (X : Integer; N : Integer) is
  K : Integer := N;
begin

  if X > 10 then
    Start_Engine;
    K := 55;
  end if;

  for I in 1 .. K loop
    Mark_Point (I);
  end loop;

end Nundo2;
```

Here the loop is controlled by the local variable *K* which is initialized to the parameter *N* but changed to 55 when *X* is greater than 10. In a context-free analysis *X* is unknown, thus the loop may have the upper bound *N*, so *N* seems essential. However, if we analyse *Nundo2* in a context that provides no bounds on *N* but implies that $X = 13$, say, then the analysis may show that *K* is necessarily set to 55 which means that the loop is bounded although *N* is still unknown.

Another variation could use a conditional statement to choose which of several parameters defines the loop bound; if a context defines the choice condition only the chosen parameter is essential. In yet another variation the parameter-dependent loop is contained within the **if**-**then** statement; if a context makes the choice condition false then the loop cannot be reached in this context and the parameter that sets the loop-bound is not necessary in this context.

- Inputs that seem essential can be dominated by constants that make them non-essential (but can cause large over-estimates).

For example, consider this modified form of the *Nundo* subprogram, again with changes in **bold** style:

```
procedure Nundo3 (X : Integer; N : Integer) is
begin

  if X > 10 then
    Start_Engine;
  end if;

  for I in 1 .. Integer'Min (N, 1000) loop
    Mark_Point (I);
  end loop;

end Nundo3;
```

The only difference with respect to the original *Nundo* is that the upper bound on the loop counter *I* is now defined as the smaller of *N* and 1000. This means that Bound-T finds an upper bound of 1000 loop iterations even when the value of *N* is unknown. Thus *N* is no longer an essential input. However, the context-free execution bounds (1000 iterations) may be greatly over-estimated compared to context-dependent bounds for smaller values of *N*.

Non-essential inputs can matter

Consider again the *Nundo* subprogram that was introduced at the start of this section with its two inputs *X* (not essential) and *N* (essential). Assume that the program contains the following call where *Nundo* is called with *X* equal to 7 and *N* to 31:

```
Nundo (X => 7, N => 31);
```

When Bound-T analyses *Nundo* in the context of this call it uses the essential *N* value to bound the loop. However, it also uses the non-essential *X* value and finds that the condition $X > 10$ is false and thus that execution cannot reach the call to *Start_Engine*. This should give very good execution bounds that apply to the case $X = 7$, $N = 31$.

In fact, since *Nundo* does not use *X* for any other purpose these bounds apply when $N = 31$ for *any* value of *X* less or equal to 10, but Bound-T does not make use of this fact. If there is another call with such *X* and *N* values, for example *Nundo* ($X => 5$, $N => 31$), Bound-T will make a new analysis of this call and will not reuse the execution bounds from the first call.

Now assume that the call defines the value of *N* but not that of *X*, as in:

```
Nundo (X => Y, N => 31);
```

where *Y* is some input to the caller and is thus unknown in the context of just this call. When Bound-T analyses *Nundo* in the context of this call it uses the essential *N* value to bound the loop. It has no bounds on the value of *X* so it includes a possible call to *Start_Engine*. The execution bounds thus apply to the case $N = 31$, for any value of *X*, and are overestimated for values of *X* less or equal to 10. Since *X* is not an essential input for *Nundo* Bound-T is satisfied with these bounds for this call even if an analysis in a deeper context might set bounds on *Y*, thus on *X*, and thus give tighter execution bounds.

Forcing context-dependent analysis

To repeat: if Bound-T finds context-free execution bounds on a subprogram it uses these bounds for all calls of this subprogram, even if context-dependent analysis could give better (sharper) bounds. Similarly, if Bound-T finds execution bounds on a subprogram for a certain suffix context, it uses these bounds for all calls of this subprogram in matching contexts even if an analysis in some deeper context could give better (sharper) bounds. At present there is no way to force Bound-T to look for (deeper) context-dependent bounds in such cases.

The only work-around currently available is to analyse the subprogram separately for each desired context under assertions that define the inputs for the context. You then feed the resulting WCET bound for each context into the analysis of the caller as an assertion on the execution time of the call to the context-dependent subprogram. This is admittedly cumbersome.

3.10 Getting Started With a Real Program

Suppose you have a real target program and want to use Bound-T to find out something about the program's real-time performance, where do you start? Here is a suggestion.

The suggested sequence of steps, below, assumes that the target program has not been written with Bound-T in mind, so it does not try a fully automatic analysis. This will also help you understand how the final WCET values are computed and the assumptions or approximations that are used.

Here are the suggested steps:

1. Decide which parts of the target program are of interest. The parts could be individual subprograms, interrupt handlers, threads or tasks. Make a list of the subprograms that will be used as *root subprograms* for Bound-T.

2. To get an overview of each root subprogram, run Bound-T on this subprogram with the option *-no_arithmetic*. This option prevents Bound-T from trying to find loop-bounds automatically, so Bound-T will give you a listing of all the loops in the root subprogram and any callees.

As an alternative to *-no_arithmetic*, use the option *-max_par_depth 0* to let Bound-T try to find loop-bounds that depend only on local computatios (context-free bounds). This is often quick enough for a first look. You will get a listing of the loop-bounds that were found and a listing of the so-far unbounded loops.

Note, however, that the option *-no_arithmetic* may prevent the proper analysis of **switch**-**case** statements. If this happens Bound-T should warn you that certain subprograms contain “unresolved dynamic jumps”. You must then enable arithmetic analysis at least for these subprograms. Section 8.5 explains how to use assertions for that.

When you are studying a particular subprogram the option *-alone* is useful. This option restricts the analysis to the subprogram(s) you name on the command line, without going deeper in the call graph.

3. Inspect the so-far unbounded loops in the source-code of the target program. For each loop, decide whether to bound it automatically or by an assertion.

If you are familiar with the assembly language of your target processor you can use the option *-trace decode* to view the disassembled instructions as Bound-T analyses them.

4. Take each subprogram that has so-far unbounded loops, starting at the leaves of the call-tree and going on to higher-level subprograms. Write the necessary assertions and run Bound-T on the subprogram, using the assertions for this subprogram and also the

assertions you wrote earlier for the lower-level subprograms. Verify that the assertions are sufficient to bound the targeted loops and that Bound-T finds bounds for the other loops automatically. Change or add assertions when necessary. Possibly write alternative assertions for different scenarios, for example nominal cases, error cases or different application "modes" as often occur in embedded programs.

When step 4 reaches the root subprograms, you will have the WCET bound for each root subprogram and a call-graph that shows how this WCET is built up from the WCETs of the lower-level subprograms.

For large target programs, it is convenient to implement step 4 as a separate shell-script or batch command file for each subprogram and perhaps collect these into a Makefile. The shell-script should combine the necessary assertion files, run Bound-T with the chosen options, and store the output in a file for browsing. By setting up such shell-scripts, the whole analysis or any part of it can be re-run easily if the target program or the assertions are changed.

The assertion files and analysis scripts are also useful as a record of *how* you determined the time and space bounds for your application. This record can be used as "performance case" documentation, for example to show to certification authorities as part of the "safety case" documentation for a critical system.

3.11 Stack Usage Analysis

Stack analysis to avoid stack overflow

Like execution time, *stack memory usage* is also a *dynamic* attribute of a program. A program that exceeds its allocated stack memory will often fail (for example, through a *stack-overflow trap*) or behave randomly when non-stack memory is wrongly *overwritten* and its content destroyed. However, small embedded processors may not have memory to waste on safely oversized stacks, especially since each thread usually needs its own stack area.

For programs that use the stack in a suitable way, Bound-T can compute a *worst-case upper bound* on the stack usage. You can remove the risk of stack overflow by allocating at least this amount of stack space.

This section explains how stack usage analysis works in Bound-T, focusing on general aspects and leaving the target-specific aspects to the Application Notes for each target processor.

Command line

Stack usage analysis is optional and is activated by a specific command-line option (*-stack* or *-stack_path*). Stack usage can be analysed together with execution time (*-time*, the default) or separately (*-no_time*).

Results

Bound-T reports the stack-usage bound in a basic output line that starts with *Stack*, as in:

```
Stack:summer.exe:summer.c:sum_vector:3-10:HW_stack:26
```

This output line reports that the subprogram *sum_vector*, together with its callees, needs at most 26 units of space on the stack called *HW_stack*. The unit of stack usage depends on the target processor but is usally the natural unit for memory size on that processor. For example, on an octet-oriented processor this could mean 26 octets of stack space, while on a processor built around 32-bit words it could mean 26 words = 104 octets. The unit is of course defined in the Application Note for the target.

Stack mechanisms

Different target processors have very different stack mechanisms. The most common mechanism are these:

- no hardware stack at all,
- a fixed-size stack that is used only for return addresses and cannot be used for data,
- a stack in main memory that is used both for return addresses and for data and has a software-defined size.

The processor may have special instructions for accessing the stack, for example "push" and "pop" instructions. The processor may have general-purpose instructions that are also suitable for stack operations, for example load and store instructions with register-indirect addressing and auto-increment or auto-decrement of the pointer register.

For small hardware stacks that only hold return addresses the software usually has no choice in how it uses the stack. When the processor hardware is not so constraining, the processor manufacturer sometimes defines software rules for passing parameters and using the stack for local variables. This definition is called the *procedure calling standard* or the *application binary interface* definition (ABI) for this processor. Otherwise, the choice of stack-pointer register and stack-usage conventions are left to the programmers and compiler developers. When different compilers follow different conventions it may be impossible to link together their object files into a working program.

Multiple stacks and stack names

When the processor's hardware stack holds only return addresses, but the programming language provides subprograms that may be reentrant or recursive, it is common for the compiler to define its own *software stack* for parameters and local variables. Thus, some target programs use *two* stacks, hardware and software, or perhaps even several software stacks for different purposes. Software stacks are of course used also when the processor has no hardware stack at all.

The Application Note for each target explains which stacks are used on this target; this may also depend on which compiler is used and even on which compiler options are used. Bound-T analyses the usage of each stack separately.

Each stack has a name, for example "HW-stack" or "SW-stack". Bound-T includes the stack name in the output from stack usage analysis. The Application Notes explain the stack names for each target processor.

Local stack height and total stack usage

For stack usage analysis, Bound-T generally is not concerned with the details of parameter-passing mechanisms and stack lay-out (although those are important for context-sensitive analyses). Instead, the important factor is the *amount* of stack space that a subprogram allocates, called the *local stack height* of the subprogram, and how these locally allocated stack areas add up along a call-path to give the *total stack usage* of the root subprogram.

We generally assume that stack-space is *local* to a subprogram: when a subprogram returns, it must deallocate all stack space that is has allocated. However, some target processors or compilers may behave differently; please see the Application Notes. During its execution a subprogram may allocate more stack space, or release some or all of its stack space, or release some stack space allocated in the caller. For example, in many processors a call-subroutine instruction (in the caller) pushes the return address on the stack and the return-from-subroutine instruction (in the callee) pops it; in Bound-T such a return instruction makes the

local stack height negative. Likewise, in some software calling conventions the caller pushes stack-located parameters and the callee pops them; this makes the local stack height negative in the callee after the parameters are popped.

Bound-T models the local stack height in the same way as it models the values of registers and variables in memory. For example, a "push" instruction will increase the local stack height by the size of the pushed data. Arithmetic analysis (or, if we are lucky, just constant propagation analysis – see section 6.5) can give bounds on the local stack height in a subprogram. This gives a bound on the stack-space that a subprogram uses for *itself*.

Take-off height and stack usage of a call

When subprogram *A* calls subprogram *B*, the total stack-space used by *A* and *B together* for this call is the sum of

- the local stack height in *A* when the call occurs, and
- the total stack usage of *B*.

The local stack height in the caller (*A*) when a call occurs is here termed the *take-off height* for the call. The sum is called the *stack usage* of the call.

Thus, if we have an upper bound on the stack usage of *B* (either a general bound, or specific to the context of this call) and an upper bound on the take-off height for the call, the sum of these bounds is an upper bound on the stack usage of the call.

With these definitions in hand, we can explain how Bound-T computes an upper bound on the total stack usage in a subprogram *S*, including all its callees, as follows:

- If *S* is a leaf subprogram (that is, *S* calls no other subprograms), we take the upper bound on the local stack height of *S*.
- If *S* is not a leaf subprogram, we take the maximum of the upper bound on the local stack height of *S* and the upper bounds on stack usage of all calls in *S* (which, as defined above, adds the call's take-off height to the callee's stack usage).

This definition is the same as saying that Bound-T considers all call paths rooted at *S* and takes the maximum upper bound on the stack usage of any such call path. However, the upper bound on the local stack height in *S* may be larger than that of any call path, in which case the bound on local stack height is also the upper bound on total stack usage.

Worst-case stack path

The (bound on the) total stack usage is defined by the (bound on the) call-path that uses the most stack space; this call path is called the *worst-case stack path*. There may of course be several call paths with the same stack usage; they are all called worst-case stack paths but Bound-T shows only one of them in its output.

The stack usage analysis always finds a worst-case stack path, but Bound-T displays this path only if the option *-stack_path* is chosen. In this case the path is displayed by a sequence of output lines starting with *Stack_Path*. There will be one *Stack_Path* line for each subprogram in the worst-case stack path and these lines traverse the path in top-down order. For example, the path from the root subprogram *main* via *fnoo* to *emak* would be shown as:

```
Stack_Path:prg.exe:prg.c:main:34-42:SP:5:15
Stack_Path:prg.exe:prg.c:fnoo:11-32:SP:6:10
Stack_Path:prg.exe:prg.c:emak:43-66:SP:4:4
```

These output lines show that *main* needs at most 15 units of space on the stack called *SP*. Of this space, *main* itself uses 5 units (that is, the local stack height in *main* is at most 5) and the call to *fnoo* uses the remaining 10 units. The subprogram *fnoo* itself uses 6 units and *emak* uses the remaining 4 units.

The worst-case stack path is often also the longest call path, that is, the one with the largest number of nested calls. Still, a short call path can use a lot of stack space if the subprograms on the path have many parameters or many or large local variables.

It can also turn out that some subprogram *S* on the worst-case path uses only a little stack when it calls other subprograms, but at other times uses so much stack for its own purposes that its total stack usage is dominated by its local stack height. In this case, the worst-case stack path ends at *S* although *S* contains calls to other subprograms, giving longer call paths.

When the target program uses several stacks, the upper bound on stack usage and the worst-case stack path is analysed separately for each stack. Some stacks may have the same worst-case path, others may have a different worst-case paths.

The worst-case stack path may or may not be the worst-case execution path in terms of execution time. That is, an execution that reaches the worst-case stack usage may be much faster than the WCET; vice versa, an execution that reaches the WCET may use much less stack than the worst-case stack path.

Safe, unsafe or unknown stack bounds

If the local stack height of a subprogram is defined by a computation that Bound-T cannot analyse, the stack usage analysis may fail to find an upper bound on the local stack height in this subprogram and thus for any call path that includes this subprogram.

However, even if the analysis fails to find a safe upper bound on local stack height, it may still find some values for the local stack height and it will then report the largest value found as a kind of "lower bound" on the worst-case local stack usage. For example, assume that the analysis finds an upper bound of 20 units on the stack usage of subprogram *B*, but fails on the subprogram *A* which calls *B*. It is then likely that *A* and *B* together will use at least 20 units of stack. Moreover, the analysis may be able to bound the take-off height of the call $A \rightarrow B$ even if it cannot compute the local stack height everywhere in *A*. If the analysis finds an upper bound of 11 units for the take-off height, it is likely that *A* and *B* together use at least 11 + 20 = 31 units of stack.

Bound-T reports such unknown or unsafe stack usage bounds in several ways: first, as a message to say that the (maximum) local stack height is unknown for a subprogram; second, as an error message to say that the subprogram could not be fully bounded; third, as *Stack* and (if chosen) *Stack_Path* lines with a special form; and finally as an entry in the list of unbounded program elements.

In the special *Stack* and *Stack_Path* lines an unsafe stack usage has the form "*number* + ?", where the *number* is the "lower bound" on the worst-case stack usage. For the example above, the *Stack* line for *A* would be as follows (assuming that this stack is named *SP*):

```
Stack:ab.exe:ab.c:A:101-114:SP:31+?
```

With *-stack_path* and assuming that the single call $A \rightarrow B$ is the worst-case stack path for *A*, the *Stack_Path* lines would be:

```
Stack_Path:ab.exe:ab.c:A:101-114:SP:11+?:31+?
Stack_Path:ab.exe:ab.c:B:207-221:SP:20:20
```

This shows that the stack usage of *B* is at most 20 units (all local consumption) while that of *A* is more complex: *A* is likely to use at least 11 units locally, which together with *B*'s bound means that the upper bound on the total stack usage is likely to be at least 31 units.

Context-dependent stack usage

The amount of stack space that a subprogram uses may depend on the input parameters and thus on the context (the call path). For example, an integer parameter may determine the size of a local array that the subprogram stores on the stack.

Bound-T supports context-dependent stack-usage bounds in the same way as it supports context-dependent loop bounds. Bound-T first tries to bound the stack usage without context information; if this succeeds, this generic bound is used for all calls of the subprogram. If the context-independent analysis fails, Bound-T tries context-dependent analysis for ever longer call paths; when stack usage bounds are found for some context, these bounds are used for all matching contexts. That is, if Bound-T finds stack bounds for the subprogram *B*, in the context $A \rightarrow B$, it will use the same bounds for *B* in all call-paths that end with this call: $\ldots \rightarrow A \rightarrow B$.

When Bound-T fails to find stack usage bounds for some context (null or otherwise), it issues an error message.

Assertions for stack usage missing

For the execution-time analysis, Bound-T offers assertions to specify the execution time of a subprogram or a given call of a subprogram. At present, Bound-T does not offer such assertions for stack usage analysis. If you want to exclude a subprogram from the stack usage analysis you can do so by asserting its execution time, but this will leave the subprogram's stack usage undefined (unbounded). Bound-T will report this as warnings and errors and will report the final stack usage bound as an unsafe bound of the form "*number* + ?" where the *number* is the stack usage bound that omits the variable part of the stack usage of the excluded subprogram but includes the fixed part (the part that is imposed by the calling protocol as the initial value of the local stack height).

4 WRITING ANALYSABLE PROGRAMS

4.1 Why and How

To get the best results from Bound-T, you should write your programs to make them analysable by Bound-T, by using suitable styles of design and coding. As you do so, you may well find that the program becomes clearer also to human readers, and also more robust and predictable.

These design and coding styles (or rules, if you will) have nothing to do with the layout of source code, or the naming of variables and functions; since Bound-T works on the machine code, all those source-level issues have no effect. The important points are, rather:

- All loops should have counters, at least "last resort" counters.
- The initial value, increment, and final value of the counter should be simple (at most first degree) expressions, and should be passed as single parameters rather than in structures or arrays.
- Dynamically computed jumps, such as switch/case statements, should be avoided, or limited to forms for which your compiler creates code that Bound-T can analyse.
- Dynamically computed calls, such as calls through function pointers, should be avoided as much as possible.

We will show examples as we go along.

4.2 Count the Loops

A *loop counter* is a variable that grows on each iteration of the loop, such that the loop terminates when the counter reaches or exceeds some value. Of course, the counter may as well be decreased on each iteration, and terminate the loop when it reaches or falls below some value. The former is an *up-counter* and the latter a *down-counter*.

An up-counter example in Ada, with *i* a loop counter:

```
for i in 1 .. 17 loop
  Foo (A, B(i));
end loop;
```

A down-counter example in C, with *j* a loop counter:

```
j = 17;
do {
  Foo (A, B[j]);
  j -= 2;
} while (j > 0);
```

During the arithmetic analysis of a subprogram, Bound-T finds the potential loop counter variables for each loop and tries to bound the initial value, the step value (increment or decrement), and the limit (loop-terminating) value of each potential loop counter. If it succeeds, it bounds the number of repetitions of the loop. If there are several loop counters for the same loop, Bound-T uses the one that gives the least number of repetitions.

To be avoided are simple **while**-loops such as polling loops, for example waiting on an A/D converter:

```
Start_AD_Conversion (channel);
while AD_Is_Busy loop
  null;
end loop;
```

Obviously, Bound-T cannot know how many times this loop runs. On the other hand, can you? For a robust, fault-tolerant program, surely it would be better to place an upper bound, say 100, on the number of polls:

```
Start_AD_Conversion (channel);
polls := 0;
while AD_Is_Busy and polls < 100 loop
  polls := polls + 1;
end loop;
```

Now *polls* is an up-counter and Bound-T determines that the loop runs at most 100 times. Note that the same effect can be had in different ways, one alternative being

```
Start_AD_Conversion (channel);
for polls in 0 .. 99 loop
  exit when not AD_Is_Busy;
end loop;
```

In nested loops, each level should have its own counter variable. For example, assume that the program is processing a rectangular image stored as an array *pix* indexed 0 .. *pixels* – 1, containing a certain number of image rows (scan lines), each with *cols* pixels. The image could be scanned by two nested loops in this way:

```
i := 0;
while i < pixels loop
  -- Here pix(i) is the start of a row.
  next_row := i + cols;  -- Start of next row.
  while i < next_row loop
    process pix(i);
    i := i + 1;
  end loop;
end loop;
```

Bound-T cannot find loop-bounds in the above code because the same counter (*i*) is used in the inner loop and the outer loop, and moreover the counter range for the inner loop is different on each iteration of the outer loop. Instead, use a different counter for the inner loop, and make its initial and final values independent of the counter of the outer loop:

```
i := 0;
while i < pixels loop
  for j in 0 .. cols – 1 loop
    process pix(i + j);
  end loop;
  i := i + cols;
end loop;
```

In this form of the code Bound-T has a good chance of finding the loop-bounds if it can find bounds on the values of the variables *pixels* and *cols*.

4.3 Simple Steps and Limits

A loop counter is useful for Bound-T only if Bound-T can compute static bounds on the initial value, the step and the limit value. With the current analysis algorithms, this means that each of these values should be of one of the following forms:

- a literal value, such as 123,
- a simple expression (see below) computed from such values, or
- an independent input parameter (not a component of an array or a record/struct) which is given such a simple actual parameter value at some call of the subprogram under analysis.

Bound-T propagates literal integer values along the program flow and into calls (for one or more levels), but not back up from callees to callers.

4.4 First Degree Formulas

While propagating values for loop-counter analysis, Bound-T can only evaluate formulas of degree 1 in any variable. The reason for this is that Bound-T uses a formalism called Presburger Arithmetic, which is a solvable subset of integer arithmetic but does not allow multiplication of variables (which essentially is the reason why it is solvable).

In practice, this means that you should avoid using multiplication in your loop-counting formulas, except when one or both of the factors are compile-time literals. For example, assume that you are implementing a C subprogram *Sum* to the following specification:

```
float Sum (float image[], int rows, int cols);
/* Computes the sum of the floating point image which is */
/* stored in image[] row-wise with no gaps between rows. */
```

An optimizing C programmer would probably write this body for *Sum*:

```
{  int pixels, i;
   float total = 0.0;
   pixels = rows*cols;
   for (i = 0; i < pixels; i++) {
     total += image[i];
   }
   return total;
}
```

Bound-T may be unable to bound this loop because it does not know the value of *rows*cols*, even if *rows* and *cols* are known (from a call). The loop should be written in the nested form:

```
{  int i, row_start, j;
   float total = 0.0;
   for (i = 0; i < rows; i++) {
     row_start = cols*i;
     for (j = 0; j < cols; j++) {
       total += image[row_start + j];
     }
   }
   return total;
}
```

Although this code also contains a multiplication to compute *row_start*, this does not influence the loop counters and so does not hinder the analysis.

For target processors that have a native multiplication instruction Bound-T's constant-propagation analysis may be able to compute the value of *rows*cols* when the values of the factors are known, and then Bound-T should be able to bound the *Sum* loop in its original unnested form.

4.5 Sign Your Variables

When a program variable has an unsigned type (C) or modular type (Ada), special arithmetic wrap-around rules apply if the variable is assigned an expression with a negative value. For example, if an unsigned 16-bit variable is decremented starting from the value zero, it will get the value $2^{16} - 1$. These rules are similar to overflow rules, and Bound-T currently cannot handle them in its arithmetic analysis. Thus, loops that use unsigned counter variables or unsigned counter arithmetic usually cannot be automatically bounded.

Therefore we recommend that all loop-counter variables should be declared as signed variables and only instructions meant for signed arithmetic and signed comparisons should be applied to them, as detailed in the target Application Notes. However, for some target processors it may be better to use unsigned counters, so please refer to the relevant Application Note for your target.

In the Ada language, loop counters are often of an enumerated type or a non-negative integer type (type natural or positive) for which the compiler may use unsigned-arithmetic instructions. We are working to extend Bound-T to handle such code.

4.6 Go Native by Bits

Most programming languages provide integer types of different widths, that is, different number of bits and different numerical ranges. For example, the C language provides *char*, *short*, *int*, *long* and perhaps more types, while the Ada language lets the programmer define application-specific integer types by stating the required range, as in *type counter_type is range 0 .. 670*. For both languages the compiler chooses the actual number of bits in the physical representation of the type, following some rules laid down in the language standard and taking into account the word size of the target processor.

Bound-T's arithmetic analysis models the native instructions of the target program which means that it models arithmetic on the native word size. The analysis of loop bounds thus works best if the loop counters also use the native word size. When possible you should declare the loop-counter variables and related quantities (initial and final values and counter steps) to have the native number of bits. For example, on an 8-bit processor such as the Intel-8051 architecture loop counters should be 8 bits (usually *char* in C), while on a 32-bit processor such as the ARM they should be 32 bits (usually *int* or *long* in C).

- If a loop counter is declared to be *narrower* than the native word size, the compiler may have to insert masking operations to make the code work as the language requires. These masking operations are usually bitwise logical **and** instructions and may confuse Bound-T's analysis of the loop counter.
- If a loop counter is declared to be *wider* than the native word size, the compiler has to use two or more words (registers) to store the variable and has to generate instruction sequences for each arithmetic operation. For example, a 16-bit addition on an 8-bit machine is usually implemented by an 8-bit add of the lower octets followed by an 8-bit add-with-carry of the higher octets. Bound-T is generally unable to deduce that such instruction sequences represent a 16-bit addition and thus will fail to bound the loop.

Using the native word size may be impossible; for example, a loop that repeats 1000 times cannot use an 8-bit counter. You must then use a counter that is wide enough and assert the loop bound.

4.7 Aliasing, Indirection, Pointers

Most programming languages support the concept of *pointers* or *access variables*. Thus, there can be an integer variable *n*, say, and a pointer *p* that can point to some integer variable. Assuming that *p* currently points to *n*, the value of *n* can be changed either by a direct assignment to *n*, such as *n* := 5, or by an indirect assignment via *p*, such as *p*.**all** := 5 (Ada) or **p* = 5 (C). The two names, *n* and *p*.**all** or **p*, are then *aliases* for the same integer variable.

Aliasing can also result from parameters that are passed by reference, since the same variable may then be accessible via several different parameters, and perhaps also directly (as a global).

Bound-T currently does not analyse aliasing (also called "points-to analysis"). Thus, if your program modifies loop-counting variables (either counters, or limits, or steps) via aliased references or pointers, Bound-T may give incorrect loop-bounds. It is therefore most important to avoid such coding practices if you wish to rely on automatic loop bounding.

Bound-T attempts to resolve the true address of all dynamic (indirect, indexed) memory accesses, and this may reveal some aliases which are then handled correctly. If an address is not resolved, Bound-T by default does *not* emit a warning because the failure is usually harmless: the unresolved addresses usually access arrays and not loop counters. To be quite certain they should be manually inspected. Use the option *-warn access* to make Bound-T issue warnings for all unresolved dynamic memory accesses.

Of course we intend to improve Bound-T in this area.

4.8 Switch to Ifs

To implement **switch**/**case** statements, many compilers use complex code that involves indexed or sorted tables of addresses. While we try to make Bound-T understand such code, it may be safer to avoid switch/case statements and instead use a cascade of conditionals (**if** - **else if** – **else**). Moreover, for many forms of **switch**/**case** statements Bound-T must use its most powerful form of analysis, called “arithmetic” analysis, and this can take a long time.

The Application Note for a specific target processor or target compiler will explain which switch/case forms are supported. Sometimes the right compiler options can make the compiler emit analysable switch/case code.

4.9 No Pointing at Functions

A *static call* is a subprogram call that defines the callee subprogram statically and directly by giving the actual name or address of the callee. However, many programming languages also support *dynamic calls* – subprogram calls where the callee is defined by some form of dynamic run-time value. The C language provides function pointer variables; the Ada language provides access-to-subprogram variables; object-oriented languages provide late-bound or “virtual” methods.

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.

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.

Bound-T needs to know the callee(s) of each call in order to construct the call graph of the root subprogram to be analysed. This is obviously much easier for static calls. For dynamic calls Bound-T can find the callees automatically only in some special cases and only if the computation of the callee or callees depends only on statically known data in the calling subprogram (not, for example, on parameters of the calling subprogram or on global variables). Therefore you should avoid dynamic calls in the target program. The main alternative is to replace each dynamic call by a control structure that selects between the equivalent set of static calls – a **switch**/**case** structure, or an **if**-**then**-**else** cascade.

If you must use dynamic calls you can use assertions to list the possible callees for each dynamic call. See section 5.7 for examples and section 8.14 for the assertion syntax.

5 WRITING ASSERTIONS

5.1 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 use this option several times to use assertions from several files. Examples of assertions were already shown in Chapter 3. The present chapter introduces the full assertion language by description and more examples. Chapter 8 defines the formal syntax and meaning of the assertion language.

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

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

Target-specific issues

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

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

While the assertion language is generic, the target processor and the target programming tools define how assertions should refer to subprograms and variables by name or by machine-level address. The target-specific Application Notes explain the naming rules.

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.

5.2 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,
- 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 3 below shows an example assertion file and points out the different kinds of facts and contexts in the file.

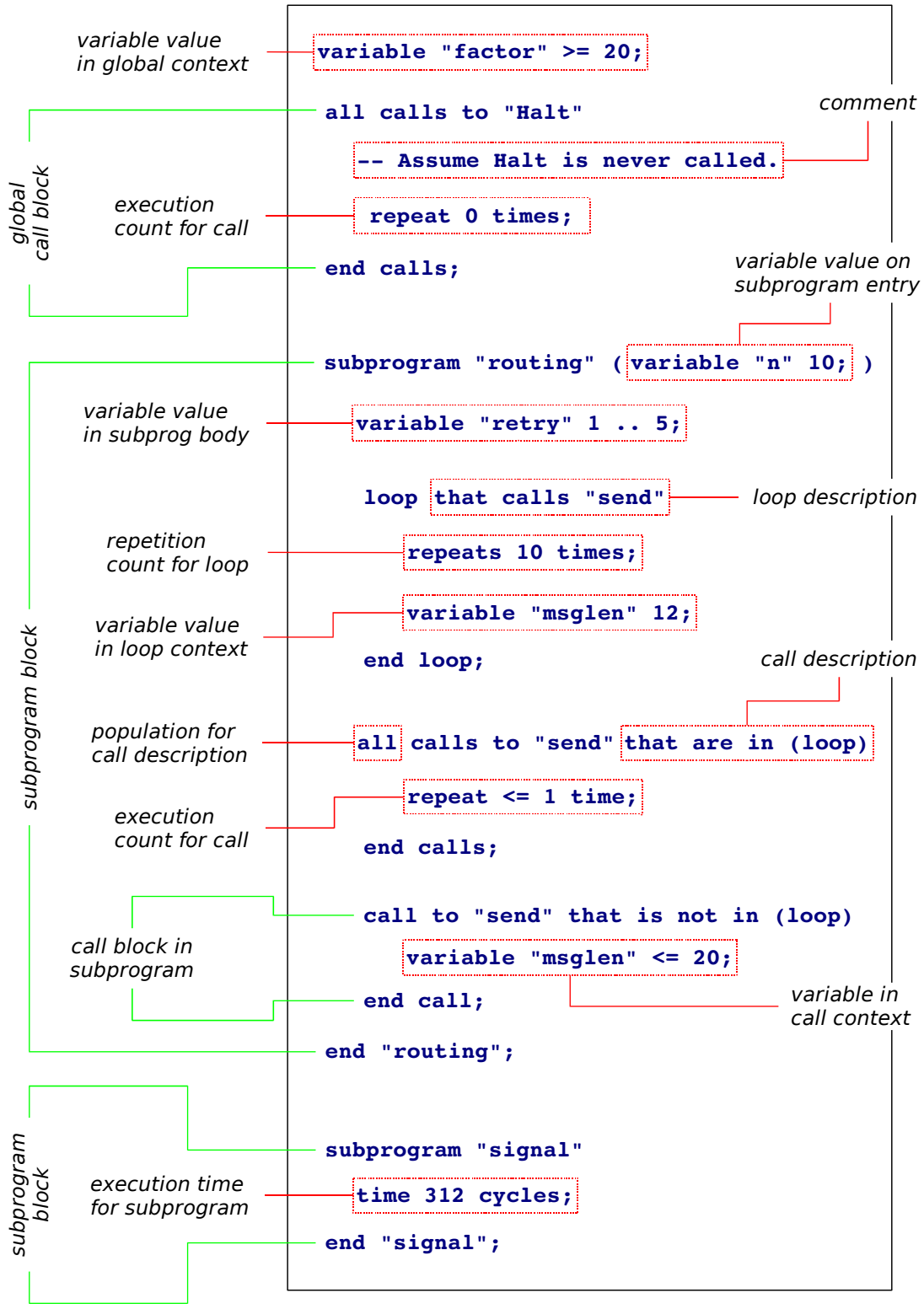

Figure 3: An assertion file

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.

5.3 Assertions on the Repetition of Loops

Why?

An 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 8.10 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 8.10 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 5.8 below.

Looping in a subprogram

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

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

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

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

Looping in any subprogram

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

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

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

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

Nested loops

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

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

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

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

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

Non-rectangular loops

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

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

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

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

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

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

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

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

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

5.4 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 5.8 below.

Don't take that path in that subprogram

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

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

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

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

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

Don't take that path in any subprogram

When all calls to a certain callee subprogram should be excluded everywhere (from all caller subprograms) the easiest way is to mark the callee subprogram *unused* as explained in section 5.11. 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.

5.5 Assertions on the Execution Time of a Subprogram

Why?

If you assert an execution time for a subprogram Bound-T will not analyze the subprogram at all. 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 5.11.

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";
```

5.6 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.

5.7 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;
```

5.8 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 5.3. 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.

5.9 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.

5.10 Assertions on 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.

5.11 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 show the subprogram in the call-graph drawing

Bound-T can help you understand your program by drawing the call-graph as explained in section 7.6. 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.

5.12 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*.

5.13 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 section 6.2, 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.

5.14 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 5.13.

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.

5.15 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 innner 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:

is labelled The loop contains (or does not contain) a specific statement label.

calls The loop calls (or does not call) a specific subprogram.

uses The loop reads (or does not read) a specific variable.

defines The loop assigns (or does not assign) to a specific variable.

is in The loop is contained (nested) in another loop (or is not so contained).

contains The loop contains (or does not contain) another loop.

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

The properties **contains** and **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 5.16.

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 (see section 7.3) 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 5.3 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 folllowing 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)`.

5.16 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 5.13). 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 5.15.

Non-returning subprograms are never in a loop

A call to a subprogram that is marked "no return" (see section 5.11) 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.

5.17 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 Bound-T detects that the loop is eternal).

5.18 Handling Recursion

The perils of recursion

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

Trivial recursions: an example

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

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

```
package Errors is

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

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

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

end Errors;
```

This module could be implemented as follows:

```
package body Errors is

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

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

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

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

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

end Errors;
```

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

Handle → *Log* → *Handle* → *Log*

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

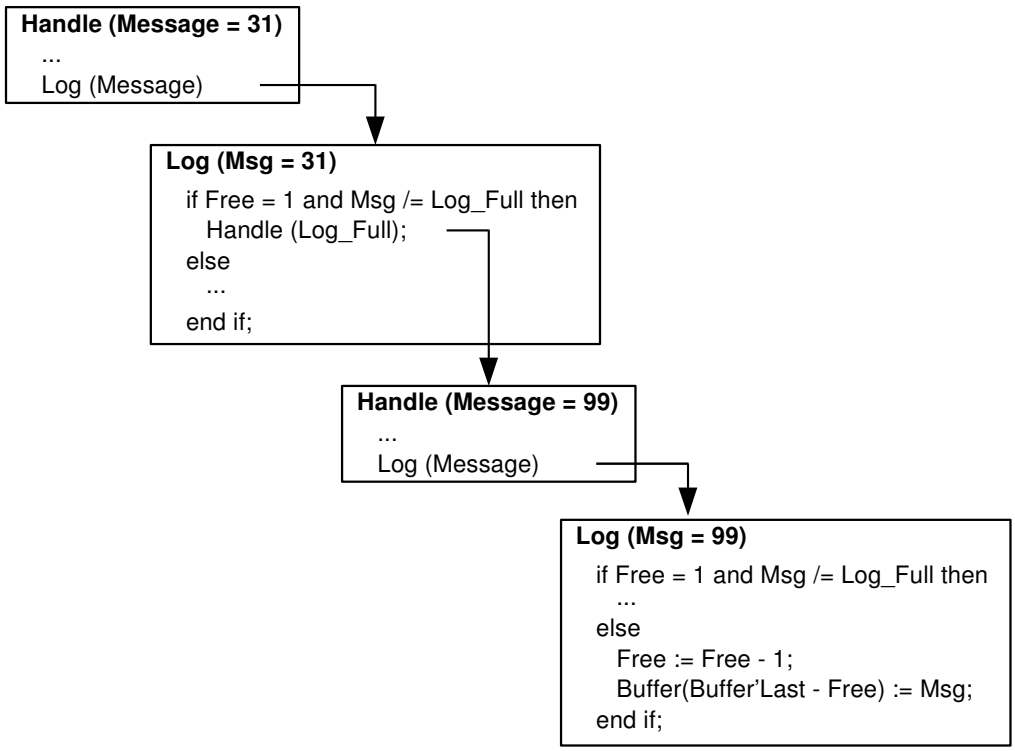

Figure 4: Longest call path in recursion example

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.

6 THE BOUND-T COMMAND LINE

6.1 Basic Form

The Bound-T command has two forms, one for the *basic mode* of operation and one for the *HRT mode* of operation. This manual discusses only the basic mode, where the command has the form

```
boundt <options> <target exe file> <root-subprogram names>
```

The command name, written just *boundt* above, usually includes a suffix to indicate the target processor, for example *boundt_avr* names the Bound-T version for the Atmel AVR processor. Please refer to the relevant Application Note for the exact name.

<options>

The options choose the analyses to be done, control optional features, select the outputs to be produced, and specify the assertions to be used, if any.

The options are described in detail below in section 6.4. For the basic mode of operation, the option *-hrt* must not be present (see section 1.4 for information on the HRT mode).

<target exe file>

The first argument after the options is the name of the file that contains the target program in linked, executable form.

Many different file formats (data encodings, file structures) exist for executable files: COFF, ELF, AOMF, S-record files, hex files and others. Sometimes the programming tools for a given target processor support only one format; sometimes the linker provides a choice of formats for the executable file. The Bound-T version for a given target processor should support the executable formats that are commonly used with this processor; please refer to the relevant Application Note.

<root-subprogram names>

The rest of the arguments are the names (identifiers) of the subprograms for which time bounds and/or stack bounds are wanted. These subprograms are called *roots*. Their order is not important. Bound-T will analyse each of them, and all the subprograms they call.

The name for a subprogram must be given in the form used by the linker, the same form as in assertions (see section 5.13). The link-name is often slightly different from the identifier used in a high-level source language; for example, C programming systems often add an underscore to the function identifier so that the C function "foo" becomes the link name "_foo".

For most target processors a root subprogram can also be identified by giving its entry address in the code, usually in hexadecimal form. The Application Notes for specific targets and cross-compilers explain the form of link-names and entry addresses.

6.2 Special Forms

If Bound-T is invoked with no arguments, it will report an error.

If Bound-T is invoked with the option *-help*, it will print out some help on the command format and the options.

If Bound-T is invoked with the option *-version*, it will print out its version identification (target processor and version number).

If Bound-T is invoked with the option *-license*, it will print out a description of the license under which it runs. This can be useful for evaluation licenses that are of limited duration.

The options *-help*, *-version* and *-license* can also be used in a normal execution of Bound-T. For example, *-version* can be useful documentation for analysis results.

If Bound-T is invoked with the name of a target program file but no root subprogram names, it will read the target program and display it on standard output, including a dump of the memory image and the symbolic debugging information. The form of the output is target-specific.

There may be other special command forms for some targets; please refer to the Application Note for your target.

6.3 Options Grouped by Function

Command-line options are used to:

- select what to analyse: execution time, stack usage or both;
- control optional parts and parameters of the analysis;
- choose what results should be produced;
- control the form and detail of the output; and
- possibly alter (patch) the target program before analysis.

Finally, there are some options that are used rarely and only for troubleshooting.

This subsection lists the options compactly, grouped in this way. The next subsection describes the options in detail in alphabetical order. The target-specific options are explained in the relevant Application Notes.

Selecting the analysis

The following options select the kind of analysis that Bound-T will do. The default is *-time*.

Option	Meaning
-hrt	HRT mode analysis. The default is basic mode. See section 1.4.
-stack	Stack usage analysis. By default not selected. See section 3.11.
-stack_path	Stack usage analysis with display of the worst-case stack path.
-time -no_time	Selects or omits execution-time analysis. Selected by default.

Controlling the analysis

The following options control details of the selected analyses. The defaults are as follows:

```
-no_alone
-arithmetic
-arith_ref relevant
-bold
-calc_max 40_000_000
-const_iter 10
-const_refine effect
-const_refine cond
-flow_iter 5
-max_par_depth 3
```

Option	Meaning
-alone -no_alone	The *-alone* option analyses only the root subprograms, not the subprograms that the roots call. All non-root subprograms are considered to have zero execution time and zero stack usage.
-arithmetic -no_arithmetic	Enables or disables analysis of the arithmetic computations. This analysis is necessary for automatic loop analysis and for analysis of many forms of switch/case statements. It is enabled by default but can be slow for complex or large subprograms.
-arith_ref *choice*	Chooses the subset of dynamic data-memory references that will be subjected to arithmetic analysis (unless all such analysis is disabled by *-no_arithmetic*). By default all references that read (load) relevant data are analysed but references that write (store) data are not.
-assert *filename*	Names a file of assertions to guide the analysis. This option can be repeated to name all the necessary assertion files.
-bold	Makes the analysis continue even if some subprogram could not be bounded.
-timid	Makes the analysis stop when some subprogram can't be bounded.
-calc_max *N*	Sets an upper bound *N* on the magnitude of literal constants that are given as such to the Omega auxiliary program for the arithmetic analysis. Larger literals are translated to unknown (unconstrained) values.
-const_iter *N*	Limits the number of iterations of constant propagation followed by resolution of dynamic data references.
-const_refine *item*	Selects the kind of refinements (partial evaluations) that are applied as a result of the constant-propagation analysis. All refinements are enabled by default.
-no_const	Disables constant propagation analysis.
-flow_iter *N*	Limits the number of iterations for dynamic control-flow analysis.
-max_anatime *duration*	Sets an upper limit on the *duration* of the analysis, in seconds.
-max_par_depth *N*	Limits the number of parameter-passing levels (contexts) analysed.
-model_iter *number*	Limits the number of iterative updates of a computation model.
-no_join	Forces arithmetic analysis to model each instruction as an Omega relation.
-no_bitwise_bounds	Disables the arithmetic analysis of bit-wise logical and/or instructions.
-no_orig	Disables the value-origin (copy propagation) analysis.
-no_prune	Disables the pruning of infeasible parts of control-flow graphs.
-virtual *item*	Controls the analysis of virtual function calls for processor/compiler combinations that implement this concept.

Choice of outputs

The following options choose what Bound-T will produce as output. The default is *-quiet*.

Option	Meaning
-anatime	Shows the total elapsed analysis time.
-dot *filename*	Generates drawings of the control-flow graphs and call graphs in a single DOT file with the given *filename*. See section 7.6.
-dot_dir *dirname*	Generates a drawing of each control-flow graph and call graph as separate DOT files within the directory of the given *dirname*. See section 7.6.
-dot_page *size*	Adds a page-size definition to each generated DOT file. See section 7.6.
-dot_size *size*	Adds a drawing-size definition to each generated DOT file. See section 7.6.
-draw *item*	Chooses the *items* to be shown in the DOT drawings. See section 7.6.
-help	Lists the command-line options (both generic and target-specific).
-license	Displays information about the Bound-T license.
-q -quiet	Disables the output of verbose messages ("notes").
-show *item*	Chooses the detailed *items* to be included in the detailed output.
-stack_path	Displays the worst-case stack path for each root subprogram. See section 3.11.
-table	Creates a table that shows how the WCET of a root subprogram is built up from the WCETs of lower-level subprograms. See section 7.4.
-v -verbose	Enables the output of a lot of verbose messages ("notes").
-version	Displays the Bound-T version: the target processor and the version number.
-warn *item*	Chooses which types of warnings will be output.

Control over output format

The following options control details of the output from Bound-T. The default options are as follows:

```
-lines around
-output_sep ':'
-source base
```

Option	Meaning
-address	Shows also the code addresses, not just source-line numbers.
-no_address	Shows code addresses only when no source-line numbers are known.
-scope	Shows also the scope of each subprogram, for example the name of the module that contains the subprogram, not just the subprogram name. Implies the option *-draw scope*.
-lines around	Shows source-line numbers close to the code address, if no exact match.
-lines exact	Shows only source-line numbers that match code addresses exactly.
-output_sep *C*	Defines the field-separator character *C* for the basic output lines.
-source base	Source-files omit the directory (folder) path: `foo.c`.
-source full	Source-file names include the directory (folder) path: `/home/bill/src/foo.c`.
-split	Splits the *Wcet* and *Wcet_Call* outputs into “self” and “callees” parts.

Patching the target program

In some special cases it is convenient to apply small “patches” to the target program before analysis, that is, small changes to the program's memory image as loaded from the executable file. Bound-T provides the *-patch* option for this.

Option	Meaning
-patch *filename*	Names a file of patches (changes) to be applied to the loaded memory image of the target program, before analysis begins. This option can be repeated to name all the necessary patch files, which will be applied in the same order (thus, the later files can override patches defined in earlier files).

Troubleshooting and diagnostic options

The following options are useful for diagnosing problems in the analysis but may require more insight into Bound-T's internal workings than is given in this manual.

Option	Meaning
-imp *item*	Enables the internal (implementation) option named by *item*.
-keep_files	Retains certain temporary files instead of deleting them.
-trace *item*	Enables on-the-fly tracing output for various things.

6.4 Options in Alphabetic Order

The following table lists the target-independent options in alphabetical order. Note that some options must be followed by an argument, which is the next argument on the command line (there must white-space between the option and its argument). Target-specific options may exist, and are then explained in the Application Note for the target. Target-specific options may use different conventions for separating an option and its argument.

Numeric arguments can be written in base 10 (decimal) or in some other base using the Ada notation for based literals. For example, the hexadecimal literal 16#20# equals the decimal literal 32, or 10#32# in based notation. Underscores can be used to separate digit groups for clarity, for example 1_200_320 is the same as 1200320.

An *italic* word in the “Option” column stands for some specific word, number or other choice. For example, in “-assert *filename*” the *filename* part stands for the name of a file. The main table is followed by sub-tables that give the possible values for such arguments where this value set is small and fixed.

The notation “[*no_*] *item*” means a choice of “*item*” or “*no_item*”. That is, the *item* is either included or excluded from some optional function. For example, the option *-warn no_sign* disables warnings about literals with uncertain sign, while *-warn sign* enables them.

Table 2: Command-line options

Option	Meaning and default value	
-address	*Function*	Include machine-code addresses in the basic output to indicate the location of subprograms, loops, calls or other program parts.

Option	Meaning and default value	
	Default	The default is *-no_address* which see.
-alone	*Function*	Analyse only the root subprograms listed in the command line, not any of the subprograms that the roots call. All non-root subprograms are considered to have zero execution time and zero stack usage.
	Default	The default is *-no_alone* which see.
-anatime	*Function*	Show the total elapsed time of the analysis, as an output line with the keyword Analysis_Time. See chapter 7.
	Default	The analysis time is not shown.
-arithmetic	*Function*	Enforce Presburger arithmetic analysis even when not needed. This can be overridden with assertions for subprograms as explained in section 5.11. See also *-no_arithmetic*.
	Default	Arithmetic analysis is applied only when needed to bound a subprogram.
-arith_ref none -arith_ref relevant -arith_ref all	*Function*	Chooses the subset of dynamic data-memory references that will be subjected to arithmetic analysis (unless all such analysis is disabled by *-no_arithmetic*). The *none* choice prevents all arithmetic analysis of such references; the *relevant* choice restricts analysis to references that read relevant data, but omits references that write data; the *all* choice applies arithmetic analysis to all dynamic data references, whether or not they seem relevant to other analyses, for example to loop bounds. At present arithmetic analysis of dynamic data references is useful only when it can resolve the reference to a single possible data address (possibly depending on subprogram calling context). In most cases constant-propagation analysis is sufficient to resolve such references and it is seldom necessary to apply the more time-consuming arithmetic analysis.
	Default	*-arith_ref relevant*
-assert *filename*	*Function*	Use assertions from the named file. This option can be repeated to name several assertion files; all the files are used.
	Default	No assertions are used.
-bold	*Function*	Even if the path-bounding phase cannot bound all subprograms, analysis continues to find the worst-case measures of the bounded subprograms. Otherwise (see *-timid*) the analysis stops after reporting the unbounded parts.
	Default	This is the default.
-calc_max *number*	*Function*	Specifies the maximum literal value to be included as such in the arithmetic analysis. Values with a larger magnitude are considered opaque (unknown). In some cases, the limit may have to be reduced to avoid overflow in the Omega calculator.
	Default	The default limit is 40_000_000.
-const_iter *number*	*Function*	Sets the maximum *number* of iterations of constant propagation alternated with resolution of dynamic data accesses. See section 6.5.

Option	Meaning and default value	
	Default	The default *number* is 10.
-const_refine [no_] *item*	*Function*	Controls how the constant-propagation analysis is used to refine (simplify) the model of the target program. The possible *items* are listed in a table below. The "*no_*" prefix disables refinements of this kind, otherwise the option enables them. See section 6.5.
	Default	The default is to apply all possible refinements.
-dot *filename*	*Function*	Generates drawings of the control-flow graphs and call graphs in a single DOT format file with the given *filename*. The file is created if it does not already exist and overwritten if it exists. This option overrides the *-dot_dir* option. See section 7.6.
	Default	Drawings are not generated.
-dot_dir *dirname*	*Function*	Generates a drawing of each control-flow graph and call graph as separate DOT files within the directory of the given *dirname*. This directory must already exist; Bound-T will not create it. This option overrides the *-dot* option. See section 7.6.
	Default	Drawings are not generated.
-dot_page *size*	*Function*	Adds the command "page=*size*" in each generated DOT file to define the page size that DOT should assume. The size is given as two decimal numbers separated by a comma; the first number is the page width in inches and the second number is the page height in inches. See section 7.6.
	Default	No page size is defined in the DOT file.
-dot_size *size*	*Function*	Adds the command "size=*size*" in each generated DOT file to define the drawing size that DOT should aim at (bounding box). The size is given as two decimal numbers separated by a comma; the first number is the drawing width in inches and the second number is the drawing height in inches. See section 7.6.
	Default	No drawing size is defined in the DOT file.
-draw [*no_*] *item*	*Function*	Controls the number and form of the drawings, if some control-flow graphs are drawn (see option *-dot*). The possible *items* are listed in a table below. If the "*no_*" prefix is included, the item is omitted from the drawing, otherwise it is included.
	Default	See table below.
-flow_iter *number*	*Function*	Set the maximum *number* of iterations of alternating flow-analysis and dynamic data/flow resolution.
	Default	The default number is 5.
-help	*Function*	Displays a list of all command-line options, both general options and target-specific options.
	Default	None.
-hrt	*Function*	Chooses the HRT mode of Bound-T operation. See section 1.4.
	Default	The basic mode, without HRT features.
-imp *item*	*Function*	Enables the internal implementation option *item*. We do not document the possible *items* here because it would require a detailed description of Bound-T internal structures.

Option	Meaning and default value	
	Default	Bound-T works as described in this manual.
-keep_files	*Function*	A diagnostic option that makes Bound-T keep as files the input and output data streams to and from the auxiliary programs for arithmetic analysis (*Omega*) and Integer Linear Programming (*lp_solve*). Normally these data are not stored. See below for file naming rules. This option currently works only on Linux hosts.
	Default	These data streams are not stored in files.
-license	*Function*	Displays Bound-T license information.
	Default	Not displayed.
-lines exact -lines around	*Function*	Selects how target code addresses are connected to source-line numbers for display purposes: whether an exact connection is required or if the closest source-line number around the code address can be shown instead.
	Default	*-lines around*
-max_anatime *duration*	*Function*	Aborts the analysis if it has not finished within the given *duration*. The duration is measured in seconds of wall-clock time (not processor time) and possibly with a decimal part. For example, *-max_anatime 3.5* sets a maximum duration of three and a half seconds.
	Default	No limit on the duration of the analysis.
-max_par_depth *number*	*Function*	The bounds of a loop may depend on actual parameter values passed in from the caller(s), perhaps across many call levels. This option defines the maximum *number* of call levels across which parameter values are analysed to find such call-dependent loop-bounds. To disable call-dependent analysis, set the *number* to zero.
	Default	The default value is 3.
-model_iter *number*	*Function*	Sets the maximum number of iterations of updates to the "computation model" of a subprogram. Iterations may be necessary when analysis resolves dynamic references to identify new storage cells that take part in the computation.
	Default	The default value is 5.
-no_address	*Function*	Locations are indicated by source-line numbers. Machine-code addresses are used only if source-line numbers are not available or no source-line numbers are associated with this location.
	Default	This is the default.
-no_alone	*Function*	Analyse the root subprograms and all subprograms that are called from the root subprograms, directly or indirectly. The whole call-graph below the roots is analysed except as limited by assertions.
	Default	This is the default.

Option	Meaning and default value	
-no_arithmetic	*Function*	Disables Presburger arithmetic analysis. Warnings are emitted if arithmetic analysis is needed to bound a subprogram. This option can be overridden with assertions for subprograms as explained in section 5.11. See also *-arithmetic*.
	Default	Arithmetic analysis is enabled.
-no_bitwise_bounds	*Function*	Disables arithmetic analysis of bitwise and-or operators. See section 6.5.
	Default	Analysis of bitwise operators is enabled.
-no_const	*Function*	Disables constant-propagation analysis. See section 6.5 and *-const_refine*.
	Default	Constant-propagation analysis is enabled.
-no_orig	*Function*	Disables value-origin (copy propagation) analysis. See section 6.5.
	Default	Value-origin analysis is enabled.
-no_prune	*Function*	Disables the pruning (removal) of dead, unreachable parts from the control-flow graphs. See section 6.5.
	Default	Pruning is enabled.
-no_time	*Function*	Disables the analysis of worst-case execution time. See *-time*.
	Default	Execution time is analysed.
-output_sep *character*	*Function*	Defines the *character* that is used to separate fields in the basic output lines. See section 7.2.
	Default	The colon character, ':'.
-patch *filename*	*Function*	Names a file of patches (changes) to be applied to the loaded memory image of the target program, before analysis begins. This option can be repeated to name all the necessary patch files, which will be applied in the same order. Thus, the later files can override patches defined in earlier files.
	Default	No patches are used. The executable file is used as it stands.
-q -quiet	*Function*	Do not display remarks and progress messages (basic output classified as "notes"). The two forms *-q* and *-quiet* are equivalent. See also *-v* and its synonym *-verbose*.
	Default	Quiet. "Notes" are suppressed.
-scope	*Function*	Qualify subprogram names with the scope, in all output. Thus subprogram *foo* defined in module *Mod* will be identified as *Mod|foo*. Useful when subprogram names are often over-loaded. Implies the option *-draw scope*.
	Default	Scopes are not shown; only the basic name (*foo*) is shown.
-show *item*	*Function*	Requests the detailed output of the analysis results identified by *item*. Section 7.5 explains the detailed outputs. The possible *items* are listed in a table below. The option *-show callers* has an additional role: it adds inverse call-tree information to the list of unbounded program parts (see section 7.3).
	Default	No detailed output is emitted.

Option	Meaning and default value	
-source base -source full	*Function*	Controls the presentation of the names of source-code files and executable files in the output. The *full* choice displays the whole file-name including the path of folder names: `/home/bill/src/foo.c`. The *base* choice displays only the file-name, no folders: `foo.c`.
	Default	*-source base*
-split	*Function*	Modifies the form of output lines with the keyword *Wcet* or *Wcet_Call* by splitting the time bound into "self" and "callees" parts. See chapter 7.
	Default	Only the total WCET is shown, including “self” and “callees”.
-stack	*Function*	Enables the stack-usage analysis for each root subprogram named in the arguments. See section 3.11.
	Default	Stack usage is not analysed.
-stack_path	*Function*	Enables stack-usage analysis and also displays the worst-case stack path – the call-path that accounts for the maximal stack usage – for each root subprogram. See section 3.11.
	Default	Stack usage is not analysed. Under *-stack* the worst-case stack path is not shown.
-synonym	*Function*	Lists all synonyms for all identified subprograms in the program, at the end of the analysis. A synonym is another identifier (subprogram or label name) that is connected to the same code address. This may help you relate the names tha Bound-T uses (linkage names) to the names in the source-code of the program under analysis.
	Default	Synonyms are not listed.
-table	*Function*	Generates a table showing how the WCET bounds for each root subprogram are made up from bounds on the lower-level callee subprograms. See section 7.4.
	Default	No tabular output.
-time	*Function*	Enables the analysis of worst-case execution time for each root subprogram named in the arguments. See also *-no_time*.
	Default	Time is analysed.
-timid	*Function*	Stops the analysis after the path-bounding phase, if some subprograms are unbounded. Otherwise (see *-bold*) the analysis continues and tries to find the worst-case measures of the bounded subprograms.
	Default	Analysis continues (*-bold*).
-trace *item*	*Function*	Requests on-the-fly tracing of a certain *item* (an event or stage within the analysis). The possible *items* are listed in a table below.
	Default	All tracing is turned off.

Option	Meaning and default value	
-v -verbose	*Function*	Displays remarks and progress messages (basic output classified as "notes"). The two forms *-v* and *-verbose* are equivalent. See also *-q* and its synonym *-quiet*.
	Default	This output is suppressed (quiet).
-version	*Function*	Displays the version of Bound-T: the name of the target processor and the version number of Bound-T itself.
	Default	The version is displayed only when the *-help* option is used.
-virtual *item*	*Function*	Controls the analysis of virtual function calls for target processors and programming languages where this concept is implemented. The possible *items* are listed in a table below.
	Default	Static analysis of the set of callees (*-virtual static*).
-warn [*no_*] *item*	*Function*	Enables or disables the specific type of warnings named by the *item*. The possible items are listed in a table below.
	Default	See table below.

Drawing options (-draw)

The following table lists the *item* values that can be used with the *-draw* option. Multiple *-draw* options can be given, with cumulative effect. For example, the command

```
boundt –draw step –draw cond –dot drawing.dot ...
```

turns on drawing of both the step-addresses and the edge conditions and names the output file `drawing.dot`.

The *-draw* items fall in five groups that control respectively 1) some properties of all drawings, 2) the form of the call-graph drawing, 3) the choice of subprograms for which flow-graphs are drawn, 4) which flow graphs to draw for each chosen subprogram, and 5) the information to be shown in the flow-graph drawings. These groups are explained in the corresponding thre tables below. The rightmost column in these tables shows the default options which are used if only the *-dot* or *-dot_dir* option is given (and no *-draw* options). By using items with the *no_* prefix you can cancel these defaults. Section 7.6 explains the *-dot* output.

There is one *-draw* item that applies to all drawings:

Table 3: Options for all drawings

-draw item	Effect	Default?
scope	Shows also the scope of each subprogram, for example the name of the module that contains the subprogram, not just the subprogram name.	Only if the option *-scope* is used

There are a numer of *-draw* items that control the call-graph drawing:

Table 4: Options for call-graph drawings

-draw item	Effect	Default?
bounds	The nodes in the call-graph drawing represent execution bounds for a subprogram rather than the subprogram itself. If a subprogram has only one set of execution bounds (context independent bounds), it appears as one node; if it has several (context dependent) execution bounds, it appears as several nodes, one for each set of execution bounds.	
no_bounds	For subprograms with context-dependent execution bounds, all bounds are summarised into one node in the call-graph, so the call-graph drawing has one node per subprogram.	Yes
scope	Shows also the scope of each subprogram, for example the name of the module that contains the subprogram, not just the subprogram name. Implies the option *-draw scope*.	

There are some *-draw* items that control the choice of subprograms for which flow-graphs are drawn:

Table 5: Options for choosing subprograms for flow-graph drawings

-draw item	Effect	Default?
deeply	Draw flow graphs for all subprograms in the call tree.	Yes
no_deeply	Draw flow graphs only for the root subprograms named on the command line, but not for the subprograms they call.	

There are several *-draw* items that define which flow-graphs will be drawn for the chosen subprograms. In fact each subprogram has only one flow-graph, but when the subprogram has different context-dependent execution bounds it may be interesting to make a separate drawing of the flow-graph for each set of execution bounds, to see the different worst-case execution paths in the flow graph. Any combination of the items in the following table can be specified, but the items *used*, *min* and *max* are irrelevant if the item *all* is specified since *all* includes all execution bounds. The default is to draw no flow-graphs at all.

Table 6: Options for choosing the flow-graphs to be drawn

-draw item	Effect	Default?
all	Draw a separate flow-graph for each set of execution bounds for the subprogram.	
used	Like *all* but include only execution bounds that take part in the worst-case execution path of some root subprogram.	
min	Draw a flow-graph that shows the execution bounds that have the smallest (minimum) worst-case time bound for this subprogram. Note that this is not a best-case time bound!	
max	Draw a flow-graph that shows the execution bounds that have the largest (maximum) worst-case time bound for this subprogram.	
total	Draw a flow-graph that shows the execution counts and times for the subprogram, within the bounds for the root subprogram.	

There are several *-draw* items that control the information to be shown in the flow-graph drawings:

Table 7: Options for flow-graph drawings

-draw item	Effect (what is shown in the drawing)	Default?
address	The code address range [first, last] of each flow-graph node.	
cond	The arithmetic precondition of each edge in a flow graph.	
count	The execution count of each node and edge, in the execution path that defines the worst-case time bound.	Yes
effect	The arithmetic effect of each node.	
line	Source-line numbers corresponding to code addresses.	Yes
step	The machine addresses of each node.	
step_graph	Draw each flow step (machine instruction) as a node. By default each node in the flow-graph represents a basic block.	
symbol	Symbols (identifiers, labels) connected to each node.	
time	The execution time of each node and edge.	Yes

Options for constant propagation refinements (-const_refine)

The following table lists the *item* values that can be used with the *-const_refine* option. Multiple *-const_refine* options can be given, with cumulative effect. The rightmost column in the table below shows the default options. By using items with the *no_* prefix you can cancel these defaults.

Table 8: Options for the constant-propagation phase

-const_refine item	Refined element	Default?
effect	The arithmetic effects of flow-graph steps (corresponding to target program instructions).	Yes
cond	The arithmetic conditions of flow-graph edges (corresponding to conditional branches in the target program).	Yes

Auxiliary program file names (-keep_files)

The *-keep_files* option makes Bound-T create text files that record the data streams to and from the auxiliary programs `Omega` (for the arithmetic analysis phase) and `lp_solve` (for the Integer Linear Programming phase). The files are placed in the working directory and are named as shown in the table below. The part "`_N`" is a sequential number that separates the several runs of the auxiliary programs. The number starts from 1 for each run of Bound-T. For example, the files for the first execution of `lp_solve` within an execution of Bound-T are named `lp_in_1` and `lp_out_1`. Existing files with these names are overwritten without warning.

Table 9: File names for auxiliary program files

Auxiliary program	Input file for run *N*	Output file for run *N*
Omega	`omega_in_`*N*	`omega_out_`*N*
lp_solve	`lp_in_`*N*	`lp_out_`*N*

Detailed output options (-show)

The *-show* option enables the detailed output of analysis results. Section 7.5 explains the form and content of this output, which depends on the items selected with *-show item*. The following table lists the *item* values that can be used with the *-show* option. Multiple *-show* options can be given, with cumulative effect. For example, the command

```
boundt -show loops -show times ...
```

turns on detailed output of both the loop-bounds and the execution time of each flow-graph node.

Table 10: Options for detailed output

-show item	What is shown in the detailed output
general	General information, including the full name of the subprogram, the call-path for context-dependent analysis, and whether the analysis succeeded.
bounds	Computed or asserted bounds on execution time and/or stack usage of the subprogram.
callers	All call-paths to the subprogram (the inverse call tree).
cells	Input and output cells (variables and registers) for the subprogram.
counts	Execution counts of flow-graph elements (nodes, edges).
deeply	Detailed results (as selected by other items) for all subprograms and calls in the whole call tree, not just for root subprograms.
full	All other items except *callers* and *deeply*.
loops	Loop-bounds for each loop in the subprogram.
model	Final "computation model" for the subprogam, after all analyses and consequent refinements and solutions of dynamic accesses. Also shows which parts of the flow-graph are considered feasible, which infeasible.
spaces	Local stack height at significant flow-graph elements. In particular, the take-off height for all calls from the subprogram.
stacks	The final stack height for each subprogram, that is, the net push or pop effect of the subprogram on the stack.
times	Execution times of flow-graph elements.

The option *-show callers* has an additional role: it adds inverse call-tree information to the list of unbounded program parts. See section 7.3.

Tracing options (-trace)

The following table lists the *item* values that can be used with the *-trace* option for all target processors. Further *item* values may be defined for some target processors as explained in the Application Notes for those processors. Multiple *-trace* options can be given, with cumulative effect. For example, the command

```
boundt –trace decode –trace loops ...
```

turns on tracing of both the decoding process and loop structures.

There may also be further, processor-specific *-trace* items. If so, they are described in the relevant Application Note.

This tracing information is intended for troubleshooting and may not be easy to understand without some insight into the design of Bound-T. If necessary, Tidorum Ltd will help you interpret the information.

Table 11: Options for tracing

-trace item	What is traced
additional	Additional processor-specific analysis steps and results. See the processor-specific Application Notes.
arith	Start and progress of arithmetic analysis for each subprogram and each analysis context.
bounds	Building execution-bounds objects.
calc	Calculation of data-flow relations, briefly.
calc_full	Calculation of data-flow relations, fully.
calls	Call instructions found.
call_eff	The arithmetic effect of calls, as and when defined.
cells	Subprogram input, output and basis cell-sets.
const	Constant propagation results.
const_fixp	Constant propagation iterations.
context	Context data for context-specific analysis.
counters	Analysis of loop counters, showing which variables are tested and the results.
data	The simulation of the data state of the program, as part of the flow-graph construction. Such simulation is an optional and somewhat experimental analysis phase, used for some target processors, and not yet documented.
dead	Dead assignments.
decode	Decoding of program instructions. Includes disassembly listing but not the arithmetic effect (Presburger equations); for this see the item *effect*.
effect	Decoding of program instructions, with disassembly and display of the Presburger equations that model the arithmetic effect of the instruction.
flow	Constructing the control-flow graph element by element.
ilp	ILP/IPET calculations; all communication with *lp_solve*.
inbounds	Bounds on the values of input parameters and globals for calls, when set.
join	The joint artihmetic effect of a sequence of consecutive steps (instructions) in a flow graph (the result of the *-imp join* optimization).

-trace item	What is traced
joining	The process of joining the effects of consecutive steps (see *join*) in detail.
live	The arithmetic assignments that are live (effective) in each basic block in a flow graph. For more detail, see *live_step*.
live_fixp	Least-fixpoint iteration for live cells.
live_stat	Number of live vs. dead assignments.
live_step	The arithmetic assignments that are live (effective) in each step (instruction) in a flow graph.
locus	Forming the code location of a program element, for output purposes.
loops	Loop structures from control-flow.
map	Mapping assertions to code elements.
models	Managing computation models (context-dependent refinements of the arithmetic effects of nodes in the flow-graph of a subprogram).
nodes	Completed control-flow graphs by basic blocks.
nubs	The steps (instructions) that require arithmetic analysis in a flow graph.
orig	Value-origin (copy propagation) analysis results.
orig_fixp	Least-fixpoint iteration for value-origin analysis.
orig_inv	Invariant cells (variables) from value-origin analysis.
params	Parameter-bounds for calls. Mapping parameters from caller to callee.
parse	Parsing the assertion file, in detail.
phase	Progress through analysis phases: constant propagation, value-origin analysis, arithmetic analysis, iterations of the same.
proto	Analysis of dynamic calling protocols.
prune	Pruning dead (infeasible) parts from control-flow graphs.
refine	Refinements due to constant propagation.
resolve	Resolving dynamic code and data addresses.
scopes	Creating lexical scopes from the symbol tables of the target program.
stack	All results of stack height and usage analysis.
steps	Completed control-flow graphs step by step.
subopt	Applying subprogram “option” assertions such as “not used”.
subs	The set of subprograms under analysis, when it changes.
summary	The summary (total) arithmetic effect of each loop.
symbols	Symbols found in the target program.
symins	Inserting symbol-table entries, in detail.
unused	Subprograms and calls that are asserted to be “unused”.

Warning options (-warn)

The following table lists the item values that can be used with the *-warn* option. Multiple *-warn* options can be given, with cumulative effect. For example, the command

```
boundt -warn access -warn symbol ...
```

turns on warnings for unresolved dynamic memory accesses and for multiply defined symbols. The rightmost column in the table shows the default warning options. By using items with the *no_* prefix you can cancel these defaults. Note also that there are many kinds of warnings that cannot be controlled with this option and are always enabled.

Table 12: Options for warnings

-warn item	Warnings affected	Default?
access	Instructions that use unresolved dynamic data access (pointers).	
computed_return	Calls with dynamically computed return addresses.	
eternal	Eternal loops that have no exit and thus cannot terminate.	Yes
flow	Jumps and calls with dynamically computed target address.	Yes
large	Instructions that contain or involve literal values too large to be analysed as defined by the option *-calc_max*.	
reach	Instructions, loops or calls that become unreachable (infeasible), in part or in whole, through analysis or assertions. See the discussion of flow-graph pruning in section 6.5.	Yes
return	Calls to subprograms that never return.	
sign	Instructions that contain literal values with an uncertain sign, where the value can interpreted as an unsigned or signed (two's complement) value.	Yes
symbol	Symbols that have multiple definitions in the target-program symbol-table, that is, symbols that are ambiguous even when fully qualified by scope (see section 5.12).	Yes

Virtual function call options (-virtual)

The following table lists the item values that can be used with the *-virtual* option for the analysis of virtual function calls. Virtual function calls are those call instructions that are classified as a call of a virtual (late bound, dispatching) function (method) as defined in the object-oriented programming domain. Typically this means that the target of the call – the callee subprogram – is not statically defined, but depends on the dynamically defined class of the object to which the call is applied. Whether and how Bound-T detects virtual function calls depends on the target processor and the cross-compiler and is explained in the relevant Application Notes. Typically, virtual function calls can be detected only when the cross-compiler creates a description of the class inheritance structure in the symbol-table of the target program.

Table 13: Options for virtual function calls

-virtual item	Meaning	Default?
dynamic	A virtual function call is modelled as a dynamic call, that is, the callee address is the result of a computation that Bound-T will try to analyse but will probably fail to resolve. You may and probably have to assert the possible callees using a **dynamic call** assertion.	
static	A virtual function call is modelled as a set of alternative static calls to each possible implementation of the virtual function (like a switch-case statement). As no dynamic calls (in the Bound-T sense) are created you cannot assert the possible callees using a **dynamic call** assertion, but you can use other kinds of assertions to control which of the alternative static calls can be executed, and how many times.	Yes

6.5 Optional Analysis Parts

What are they?

The options *-no_bitwise_bounds*, *-no_const*, *-no_orig*, and *-no_prune* disable some optional parts of the analysis that Bound-T uses to model the arithmetic computations of the target program. The options exist to let us experiment with different sets of analyses. Normally you do not have to understand what these optional analysis parts are; just leave them enabled. Still, this section explains them briefly, to make this user manual more complete.

Bit-wise Boolean operations

Sometimes compilers apply the bit-wise Boolean operations to loop counters or other data used in loop counting. Most common is the “and” operation which is used to mask off some unwanted bits in the datum. By default, Bound-T models the bit-wise **and** and **or** operations by translating them to Presburger constraints on the integer values of the operands and the result as shown in the table below. The option *-no_bitwise_bounds* makes Bound-T instead model these operations as yielding unknown (opaque) values.

Operation	Constraint in default model	Effect under *-no_bitwise_bounds*
T := A **and** B	(0 ≤ T) and (T ≤ A) and (T ≤B)	T := unknown
T := A **or** B	(0 ≤ T) and (T ≤ (A + B))	T := unknown

Note that the symbol “and” in the constraint column means the logical “and” (conjunction of Presburger conditions), not the bit-wise **and** as in the operation column. The constraint is inserted in the arithmetic effect of the instruction that executes the bit-wise operation.

Constant propagation

Before launching the full Presburger Arithmetic analysis of a subprogram, Bound-T tries to simplify its model of the subprogram's arithmetic by propagating constant values from definitions to uses. For example, if an instruction assigns the constant value 307 to register R3, and this is the only value of R3 that can flow to a later instruction that adds 5 to R3 and stores the sum in R6, Bound-T propagates the constant along this flow and simplifies its model of the second instruction to add 5 to 307, giving 312, which is stored in register R6. Since this instruction now assigns a constant value to R6, the propagation can continue to instructions that use this value of R6, and so on.

Compilers usually apply constant propagation in their code optimization, so why should further constant propagation in Bound-T be useful? There are three reasons:

- the instruction set may limit the compiler’s use of constants,
- a context-dependent analysis may know more constants than the compiler did, and
- the local stack height may be constant, but not explicit in the instructions.

The *instruction set* of the target processor may not allow immediate (literal, constant) operands that are large enough to hold constants known to the compiler. The compiler must then generate code that computes the constant operand into a register. For example, there is no SPARC V7 instruction to load a 32-bit constant into a register, so the compiler must use two instructions: a **sethi** instruction that loads the high bits followed by an **or** instruction that loads the low bits. Nor is it possible to use a constant 32-bit address to access memory, so to access a statically allocated variable the compiler must generally use three instructions: **sethi** and **or** to load the address into a register and a third instruction to access the variable via this register. The model in Bound-T is more flexible, so constant propagation in Bound-T can combine the **sethi** and **or** instructions into a single constant load, and further combine that with the register-indirect memory access into an access with a static address.

Context-dependent analysis in Bound-T means that a subprogram S is analysed in the context of a call path, that is, under the assumption that the subprogram has been reached via a specific sequence of calls $A \rightarrow \mathrm{B} \rightarrow \ldots \rightarrow S$. Bound-T analyses the arithmetic of the call-path to find bounds on the inputs (parameters, globals) for S. If an input is bounded to a single value, this value is a static constant in this context and can be propagated over S. Constant propagation can handle more operations than the normal (Presburger) arithmetic analysis, including multiplication and bit-wise logical operations. Thus, constant propagation may make the arithmetic in S analysable for Bound-T where the original arithmetic is not analysable, for example because the original arithmetic multiplies variables.

The *local stack height* is similar to a variable (register) for Bound-T. As explained in section 3.11, for stack-usage analysis Bound-T tries to find the maximum value that this variable may have in the execution of the subprogram under analysis. The instructions that change the local stack height are usually of two kinds: (1) adding or subtracting a constant to or from the stack pointer register, and (2) pushing or popping a constant amount of data to or from the stack. Both translate into adding or subtracting a constant to or from the local stack height. Moreover, the local stack height generally has a constant initial value on entry to the subprogram. This means that constant propagation usually simplifies each expression assigned to the local stack height into a constant, which makes it very easy and fast to find the maximum local stack height. Thus, stack-usage analysis can often rely only on constant propagation and avoid the expensive Presburger analysis.

By finding the local stack height, constant propagation also helps to resolve accesses to local variables or parameters. Such accesses are often coded using offsets relative to the dynamic value of the stack pointer. The local stack height must be known in order to translate this offset to a static offset in the subprogram's stack frame. The static offset identifies the (stacked) parameter or local variable that is accessed.

The option *-no_const* makes Bound-T skip constant propagation. When constant propagation is enabled, some of its effects can be disabled or enabled separately by means of the *-const_refine* option.

Value-origin analysis (copy propagation)

Several analyses in Bound-T track the values of variables such as registers or memory locations along execution paths. These analyses must take into account all assignments to variables. The more assignments there are, the harder the analysis becomes.

In typical target programs some assignments can be ignored because they are surrounded by code that saves and restores the original value of the variable. This occurs especially when the calling protocol requires some registers to be preserved across any call of a subprogram ("callee-save" registers). Value-origin analysis is designed to detect this and thus to simplify the other data-flow analyses in Bound-T.

Value-origin analysis is similar to analyses called "copy propagation", "value numbering" and "static single assignment" (SSA). The analysis applies to one subprogram at a time, in bottom-up order in the call graph, and works as follows. The arithmetic assignments in the subprogram are divided into two groups:

1. Copy assignments of the form $x := y$ where the right-hand side (y) is a single variable.
2. Non-copy assignments of the form $x := expr$ where the right-hand side (*expr*) is an expression and not a single variable.

Note that instructions that save and restore registers (push, pop or the like) are copy assignments.

Each non-copy assignment $x := expr$ is taken as the *origin* of a new value (the value computed by *expr*) that becomes the value of x at this point. The value-origin analysis does not try to compute what this new value actually is; it just keeps track of where the value ends up, that is, where this origin of x is used.

A copy assignment $x := y$ is not the origin of a value but propagates the origin of y to be the origin of x.

Special value-origins are defined for the initial values of all variables on entry to the subprogram.

The analysis propagates these value-origins over the control-flow graph. When the control flow joins different origins for the same variable, the join point is taken as a new origin of the "merged" value (corresponding to "phi functions" in SSA). After the analysis we know the origin of the value of each variable at each point in the flow graph.

Bound-T uses the value-origin analysis to find variables that are invariant across the call of a subprogram: a variable must be invariant if the variable's value at all return points originates from its initial value. Knowing such invariant variables simplifies the analysis of the callers of the subprogram, for example when the caller uses the variable as a loop counter and the call is in the loop.

The option *-no_orig* disables value-origin analysis. The invariance of a variable across a subprogram call is then decided based on the calling protocol and the (static) presence or absence of assignments to the variable. The calling protocol for the subprogram can specify that certain variables (usually callee-save registers) are invariant across the call. Otherwise, if

the subprogram has an instruction that can change the variable, the variable is not considered invariant across a call. Instructions that can change a variable include assignments to the variable and calls of lower-level subprograms that can change the variable.

Flow-graph pruning

Subprograms usually contain conditional branches. The condition is a Boolean expression and often has a form that Bound-T can analyse, in part or in whole. This means that Bound-T can sometimes deduce that a branch condition must be *false*, either generally or in the context of a context-dependent analysis. A false condition means that the conditional branch cannot be taken, which means that some parts of the control-flow graph may be *unreachable*, either generally or in the current context. Such parts, and any execution paths that traverse them, are also called *infeasible*.

To simplify the analysis Bound-T will remove or *prune* the unreachable parts (nodes and edges) from the control-flow graph. The pruned parts are excluded from the analysis; they do not contribute to the arithmetic model, nor to the execution time bound, nor to the stack usage bound.

Pruning is an iterative process: when one element (node or edge) of the flow-graph is found to be unreachable this may imply that successor elements are also unreachable. When a node is unreachable, so are all the edges leaving the node. When all edges that enter a node are unreachable, so is the node.

Bound-T does not deliberately search for unreachable flow-graph parts. Rather, unreachable parts are discovered as a side effect of some analysis, as follows:

- Constant propagation may find that a branch condition has the constant value *false*.
- Arithmetic analysis of the data that reaches a loop, a dynamic memory access or a dynamic jump may show a *null* data set, meaning that the loop, access or jump is unreachable.
- An assertion may state, or Bound-T may itself discover, that the calleee of a call does not return to the caller, meaning that any control-flow edge from the call is unreachable.
- An assertion may state that a loop repeats zero times, meaning that the edges from the loop head to the loop body (including edges back to the loop head itself) are unreachable. If the loop is an eternal loop or a loop that can exit only at the end of the loop body then the whole loop (including the loop head) is unreachable.
- An assertion may state, or Bound-T may itself discover, that a loop cannot repeat even once, meaning that the “backward” or “repeat” edges from the loop body to the loop head are unreachable.
- An assertion may state that a call is executed zero times, meaning that the call is unreachable.

As an extreme case, if all return points in a subprogram become unreachable then the search for the worst-case execution path will fail and lead to a fault message from the *lp_solve* program. In future Bound-T versions this case will instead make Bound-T consider the whole subprogram unreachable and all calls to the subprogram unreachable.

Unreachability may change the looping structure of a control-flow graph in several ways:

- If the loop head becomes unreachable then the whole loop is unreachable and is pruned.
- If all the paths that can repeat the loop become unreachable then the loop is no longer a loop and is not reported as a loop in the output. However, the loop head and perhaps some parts of the loop body remain feasible and stay in the flow-graph.
- If all the paths that can exit (terminate) the loop become unreachable then the loop is an eternal loop. See section 5.17.

The option *-no_prune* disables pruning. However, Bound-T will still mark as unrecahable those flow-graph edges that have false conditions, which may cause problems in the search for the worst-case path. Operation with *-no_prune* has not been well tested and may not work.

6.6 Patch files

Patching: why and how

In some (rare) cases it is convenient to patch (that is, slightly alter) the target program for analysis purposes. Bound-T provides the command-line option *-patch* for this. This option may not be supported for all target processors; please check the Application Note for your target.

As an example of a case where patching is useful, consider a SPARC program where the addresses in the trap vector table are not defined statically (at load time) but dynamically by the boot code. Thus, Bound-T sees the traps as dynamic calls and is probably unable to find the callees (the trap-handler subprograms). If these subprograms are nevertheless statically known, you can patch their addresses into the trap vector table and then Bound-T can find and analyse the trap handlers, too.

When *-patch* is supported, the necessary patches should be written in a *patch file* or possibly several patch files and these files should then be named in *-patch* options. Patch files are text files with a generic (target-independent) surface syntax but where the detailed syntax and meaning depend on the target processor. The rest of this subsection defines the generic surface syntax; see the relevant Application Note for the target-specific syntax and meaning.

Generic patch file syntax

A patch file is a text file that is interpreted line by line as follows.

- Leading whitespace is ignored.
- A line starting with "--" (two hyphens, possibly with leading whitespace) is ignored (considered a comment line).
- Blank and null lines are ignored.
- Meaningful lines contain the following fields, in order, separated by whitespace:
  - a code address in a target-specific form (usually a hexadecimal number), denoting the starting address of the patch;
  - a string without embedded whitespace, denoting the main content of the patch in a target-specific form;
  - zero or more strings that represent code addresses or symbols connected to code addresses, with a target-specific form and interpretation.

The patching process in Bound-T reads patch lines one by one, parses them as defined above, and applies them in a target-specific way to the loaded memory image of the target program to be analysed.

Example

Here is an example of a patch file for a SPARC processor. The file changes the SPARC target program at address 40000810 (hex) by changing the 32-bit word at this address to A1480000 (hex) which encodes the SPARC instruction "rd %psr,%l0". The first two lines are comments; the third line defines the change by giving the address and the new content.

```
-- The following puts an “rd %psr,%l0” instruction
-- at the trap location:
40000810 a1_48_00_00
```

The form and meaning of SPARC patch files are further explained in the Application Note for the SPARC version of Bound-T.

7 UNDERSTANDING BOUND-T OUTPUTS

7.1 Choice of Outputs

Bound-T provides a choice of several output formats. The basic format, which is the default and was illustrated by examples in Chapter 3, is designed to be compact and easy to post-process by filters or higher-level tools, such as scheduling analysers. Section 7.2 below explains this format.

When Bound-T fails to find bounds on some parts of the target program, it lists the unbounded parts in a specific format that is explained in section 7.3.

Specific command-line options enable other forms of output as follows:

- The *-table* option gives a table of all subprograms included in the WCET bound, showing how many calls of each subprogram are included and how much time each subprogram contributes to the WCET bound. See section 7.4.
- The *-show* option gives a hierarchical, indented representation of the call graph and selected information about each subprogram and the analysis of that subprogram. See section 7.5.
- The *-dot* option creates drawings of the control-flow graphs and call graphs in DOT form. See section 7.6.

The *-trace* option can give a lot of detailed outputs about the progress of the analysis, on the fly, but this is meant for troubleshooting and the format is not explained here. Please contact Tidorum Ltd if you need to understand *-trace* output.

7.2 Basic Output Format

The fields

The basic output format consists of lines with fields separated by colon characters (or the character defined with the *-output_sep* option). The first field is a keyword such as *Note*, *Wcet*, or *Loop_Bound* that shows the type of the line. The second through fifth fields contain the name of the target-program executable, the source file, the subprogram or call being analysed, and the code location, respectively.

The remaining fields, starting from the sixth field, depend on the type of the line, as does the significance of the code location. Thus, the form is:

```
key:exe-name:source-name:sub-or-call:code-location:more
```

Fields that are undefined or not applicable are empty. For example, if Bound-T reports an error in the format of the program executable, but that does not pertain to any particular source-code file, subprogram, or code location, it emits a line of the form

```
Error:exe-name::::message
```

Subprograms and call paths

If a basic output line refers just to a subprogram, for example if it reports that the WCET of the subprogram has been bounded without considering its parameters, the sub-or-call field (field 4) contains the subprogram name alone.

If the basic output line reports on the analysis of a call path, the sub-or-call field lists the call locations in top-down order, separated by "=>". For example, the string

```
main@71[040A]=>A@[0451]=>B
```

indicates a call path starting in the subprogram *main*, where the instruction at source-line number 71 and address 040A calls the subprogram *A*, where the instruction at address 0451 but unknown source-line number calls the subprogram *B*, where the call path ends. The code addresses are usually displayed as hexadecimal numbers.

Normally, the bracketed code addresses are omitted if source-line numbers are available for this location. The option *-address* includes code addresses in all output whether or not source-line numbers are found.

Code locations

The code location field (field 5) consists of a source-line number or an instruction address or both. Either part may also be a range with a lower-bound and an upper-bound. For example, the code location "66-71[3B5F-3B6D]" means the source-lines number 66 through 71 which correspond to the instructions at the hexadecimal addresses 3B5F through 3B6D.

Normally, the bracketed code addresses are omitted if source-line numbers are available for this location. The option *-address* includes code addresses in all output whether or not source-line numbers are found.

Source-code lines around a code address

The connections between source lines and code addresses must be provided by the target compiler and linker and may not be precise or complete. For example, the compiler and linker perhaps connect a source line only with the address of the *first* instruction generated for the source line. If Bound-T then writes an output line that refers to a later instruction generated for this source line, there is no source-line number connected to exactly the address of *this* instruction.

The option *-lines around* (which is the default) lets Bound-T display the closest matching source-line number for a given code address. If no source-line number is connected exactly to this code address, Bound-T first looks for the closest match *before* this code address. If a source-line connection is found, it is displayed in the form "*number*–" to indicate that the code address comes after source-line *number*. If Bound-T finds no source-line connection before this code address, it looks for the closest match after this code address and displays it in the form "–*number*" if found.

For example, under *-lines around* (and *-address*) the call-path string

```
main@71-[3C40]=>A@-15[103F]=>B
```

shows that no line-number is connected exactly with the call from *main* to *A* at address 3C40, but the line number 71 is the closest number known before the call, while for the call from *A* to *B* no source-line number is known at or before the address 103F but the closest line-number after that address is 15.

The source-line number displayed under *-lines around* is usually the "right" one, but sometimes it can refer to another object-code module and thus to the wrong source file. This typically happens when the module that contains the code address has been compiled without debugging options, and so lacks source-line connections, but other modules have such connections.

The alternative option *-lines exact* makes Bound-T display only exactly matching source-line numbers, which means that it often displays only the code address and no source-line number.

Instruction addresses

The form of an instruction address is in fact target-specific so although the examples above showed addresses as single hexadecimal numbers, some target processors may use other formats. This is explained in the Application Note for each target processor.

All the output

The following table shows all the forms of basic output line that can occur and explains their meaning. Remember that fields 2 through 5 always contain the executable file name, source file name, subprogram name or call-path, and source line numbers or instruction addresses. However, for messages that report a problem in the assertion file, or in an HRT TPO file, the name of the relevant file is substituted for the source-file name, and the line number (if present) also refers to that file.

The explanation of the remaining fields (from field 6 on) first gives the format, using italic symbols for field values and separating fields with colons, and then explains the meaning of the symbols. The table is in alphabetic order by the keyword (field 1).

Table 14: Basic output formats

Keyword (field 1)	Explanation of fields 6-
Also	Gives additional source-code references for the preceding output line. An *Also* line arises when an output line refers to a program element with connections to more than one source file. The first output line (with a key that is not *Also*) shows the connections to one source file. Each appended *Also* line shows the connections to a further source file. This happens, for example, in Ada target programs where a program element can be connected to an Ada package *declaration* file as well as the corresponding package *body* file. An *Also* line has only five fields.
Analysis_Time	*time* An informative output message given once at the end of the analysis and showing the total elapsed (wall-clock) analysis *time* in seconds with three decimals (resolution 0.001 seconds). Note, this is time on the host machine on which Bound-T is executed, not time consumed by the target program on the target machine. This output is optional per the option *-anatime*.
Call	*callee source file* : *callee subprogram* : *callee line-numbers* An informative message that reports that a subprogram call has been detected in the subprogram being analyzed. The caller subprogram and the location of the call are identified in fields 3 through 5; the callee subprogram is similarly identified in fields 6 through 8.

Keyword (field 1)	Explanation of fields 6-
	This output is optional per the option *-trace calls*.
Error	*message* Reports an error that may prevent further analysis and means that the later analysis results, if any, are probably wrong in some way. For example, the command line may have named a subprogram that does not exist in the target program. Section 9.2 lists and explains all generic error messages that can arise with any target processor. The Application Notes explain any additional error messages for specific targets.
Exception	*message* As for the *Fault* case below, but shows that the fault led to an exception being raised. Please report to Tidorum Ltd. as for a *Fault*.
Fault	*message* Reports an unexpected error that may prevent further analysis and is probably due to a fault in Bound-T itself, not necessarily in the input data or the way Bound-T was invoked. Please report any occurrence of this message to Tidorum Ltd., preferably together with the target executable, the command line and any other input files (assertion file, TPOF).
Integrated_Call	*callee source file* : *callee subprogram* : *callee line-numbers* Basically the same as the “Call” output line (which see): an informative message that reports that a subprogram call has been detected in the subprogram being analyzed. The caller subprogram and the location of the call are identified in fields 3 through 5; the callee subprogram is similarly identified in fields 6 through 8. However, for an “Integrated_Call” the flow-graph of the callee becomes a part of the flow-graph of the caller and is analyzed as such; the callee is not considered a distinct “subprogram” to be analyzed on its own. Whether a call is analyzed in this “integrated” way can be controlled by an assertion (see section 8.5). Integrated analysis can be the default for certain subprograms for some target processors and target compilers; they are usually library routines that implement prelude/postlude code for application subprograms. This output is optional per the option *-trace calls*.
Loop_Bound	*number* An informative message that reports the computed upper bound on the *number* of iterations of a loop. This is an upper bound on the number of times the loop-head is re-entered from the body of the loop (via a "repeat edge"), for each time that the loop is started. For loops in which the termination test is at the end of the loop body, the bound is usually one less than the number of times the loop body is executed. See section 8.10 for the terms loop-head and repeat edge. The bound may depend on actual call parameters, in which case the sub-or-call field shows the call path to which this bound applies.
Note	*message* An informative message, which can be good or bad, but in any case is not severe enough to be considered worthy of a warning or error message. These messages are written only if the *-verbose* (or *-v*) option is chosen.
Param_Bounds	*parameter* : *bounds*

Keyword (field 1)	Explanation of fields 6-
	Shows the derived *bounds* on the *parameter* in the call identified by fields 3 through 5. The bounds will be used for the context-dependent analysis of the callee. This output is optional per the option *-trace params*.

Recursion_Cycle	Calls *callee*
	
	Shows one call in a recursive cycle of calls between subprograms. When Bound-T detects a recursive cycle, it first emits an *Error* line reporting that recursion exists and then follows this with one or more *Recursion_Cycle* lines that together describe a recursive cycle of calls.
	
	In each *Recursion_Cycle* line, the sub-or-call field names the caller and field 6 names the *callee*. Here is an example of a cycle with three members, the subprograms *glop*, *fnoo* and *emak*:
	
	`Recursion_Cycle:prg.exe:prg.c:glop:34-42:Calls fnoo`
	`Recursion_Cycle:prg.exe:prg.c:fnoo:11-32:Calls emak`
	`Recursion_Cycle:prg.exe:prg.c:emak:43-66:Calls glop`
	
	Note that Bound-T shows only one recursion cycle, but there may be others.

Stack	*stack* : *usage*
	
	Reports the computed upper bound on the total *usage* of a certain *stack* for a subprogram, as requested by the *-stack* option. The stack-usage unit depends on the target processor and is usually the natural unit for memory size on this processor, such as octets on an 8-bit processor. For example:
	
	`Stack:prg.exe:prg.c:main:34-42:HW_stack:15`
	`Stack:prg.exe:prg.c:main:34-42:C_stack:22`
	
	These output lines show that the root subprogram *main* (together with its callees) needs at most 15 units of space on the stack called *HW_stack*, plus 22 units on the stack called *C_stack*. See section 3.11.

Stack_Path	*stack : local-usage* : *total-usage*
	
	Reports on the call-path that causes the worst-case usage of certain *stack*, for a root subprogram, as requested by the *-stack_path* option.
	
	The *local-usage* is an upper bound on the local stack height in the current subprogram. This is the amount of stack required for the local variables of the current subprogram, without considering the stack usage of lower-level callees.
	
	The *total-usage* is an upper bound on the total stack space required by the current subprogram together with its lower-level callees.
	
	The stack-usage unit depends on the target processor and is usually the natural unit for memory size on this processor, such as octets on an 8-bit processor.
	
	There will be one *Stack_Path* line for each subprogram in the call-path that requires the most stack, and these lines traverse the path in top-down order, with the current subprogram indicated in the sub-or-call field. For example, the path from the root subprogram *main* via *fnoo* to *emak* would be shown as:
	
	`Stack_Path:prg.exe:prg.c:main:34-42:SP:5:15`
	`Stack_Path:prg.exe:prg.c:fnoo:11-32:SP:6:10`
	`Stack_Path:prg.exe:prg.c:emak:43-66:SP:4:4`
	
	These output lines show that *main* needs 15 units of space on the stack called *SP*. Of this space, *main* itself uses 5 units and the call to *fnoo* uses the remaining 10 units. The subprogram *fnoo* itself uses 6 units and *emak* uses the remaining 4 units. See section 3.11.

Synonym	*name*

Keyword (field 1)	Explanation of fields 6-
	Reports that the subprogram identified in fields 3 through 5 has another *name*. The symbol-table in the program connects this *name* to the same entry address. This output is optional per the option *-synonym*.
Time_Table	*total* : *self* : *calls* : *min* : *max* : *subprogram* : *source-file* : *code-location* One row in the tabular break-down of the WCET bound for the root subprogram identified in fields 3 through 5. This row reports the part of the WCET bound that is due to the given *subprogram* which is located in the given *source-file* and *code-location*. The worst-case execution path of the root subprogram executes the given number of *calls* of this *subprogram*. Together, these calls contribute the given *total* time to the WCET bound, of which the *self* amount is spent in the *subprogram* itself and the rest (*total* – *self*) in lower-level subprograms. The lower-level subprograms will be represented by their own *Time_Table* lines. The fields *min* and *max* show the smallest and largest WCET bound found for this *subprogram* (including its callees) over all its calls on the worst-case execution path of the root. The two fields can be different only if the WCET bound for *subprogram* is context dependent. This output is issued only under the option *-table*. See section 7.4 for further explanation.
Unused	*Unused subprogram* This subprogram will not be analysed because it is asserted to be unused. This message is issued only under the option *-trace unused*.
Unused_Call	*callee source file* : *callee subprogram* : *callee line-numbers* This call is considered infeasible because the callee subprogram is asserted to be unused. See the output line *Call* for an explanation of the output fields. This message is issued only under the option *-trace unused*.
Warning	*message* A warning message that means that the analysis results may not be correct . You should check if the reason for the warning really makes the results wrong; the warning may be a false alarm of something that does not affect the results. Section 9.1 lists and explains all generic warning messages that can arise with any target processor. The Application Notes explain any additional warning messages for specific targets.
Wcet	*time* The field *time* is the computed upper bound on the execution time of the subprogram named in the sub-or-call field, which has been determined independently of any actual parameter values (in a "null" context). If the *-split* option is used, there are two additional output fields as follows: *time* : *self* : *callees* The meaning of *time* is not changed; *self* is the part of *time* that is spent in the subprogram itself, while *callees* is the part that is spent in lower-level callees. The time is given in target-specific units, usually clock cycles.
Wcet_Call	*time*

Keyword (field 1)	Explanation of fields 6-
	The field *time* is the computed upper bound on the execution time of a subprogram call, when the bound depends on the actual call parameters (context). The sub-or-call field shows the call-path in top-down order, starting from the topmost subprogram that provides context. If the *-split* option is used, there are two additional output fields as follows: *time* : *self* : *callees* The meaning of *time* is not changed; *self* is the part of *time* that is spent in the subprogram itself, while *callees* is the part that is spent in lower-level callees. This kind of output can occur only when the option *-max_par_depth* is positive (as it is by default). The time is given in target-specific units, usually clock cycles.

7.3 List of Unbounded Program Parts

When Bound-T fails to find the requested time or space bounds on some parts of the target program, it first issues one or more error messages (usually the message "Could not be fully bounded") and then lists the unbounded parts in a specific but simple format. This section explains the format.

Call graph framework

The unbounded program parts are listed within a hierarchically indented display of the relevant portion of the call-graph. This is similar to the structure of the detailed output described in section 7.5.

Assume, for example, that the unbounded parts lie in the subprograms *Foo*, *Bar*, *Upsilon* and *Chi*, which are related by the calls *Foo* → *Bar*, *Bar* → *Upsilon* and *Foo* → *Upsilon*. Assume further that the call *Bar* → *Upsilon* gives enough context (parameter values) to bound some, but not all parts of *Upsilon*, while the context in *Foo* → *Upsilon* leaves some other *Upsilon* parts unbounded. Thus, the execution bounds for the two calls are different.

If Bound-T is asked to find the WCET bound for the root subprogram *Main*, which calls *Foo* and *Chi*, the unbounded parts are displayed as follows:

```
Main@23=>Foo
   list of unbounded parts in Foo
   Main@23=>Foo@104=>Bar
      list of unbounded parts in Bar
      Main@23=>Foo@104=>Bar@212=>Upsilon
         list of unbounded parts in this call of Upsilon
      Main@23=>Foo@123=>Upsilon
         list of unbounded parts in this call of Upsilon
Main@37=>Chi
   list of unbounded parts in Chi
```

Thus, for each subprogram that contains unbounded parts there is first a line that gives the full call path from the root subprogram and then the list of unbounded parts. The call path includes the code locations of the calls (after the '@' characters) giving the source-line number and/or the machine address of the call.

If the set of unbounded parts is context-dependent (as for *Upsilon* in the example) each different context is shown separately with the list of unbounded parts in that context.

When there are several call paths to the same subprogram, but the analysis results are the same for these paths, only the longest path is shown (depth-first traversal of the call graph). Use the option -show callers to list all call paths (see below).

This output includes only subprograms that have some unbounded parts. Fully bounded subprograms may appear in the call-path strings on the way to a subprogram that has unbounded parts.

The list of unbounded parts can list unbounded loops, unbounded local stack height and irreducible flow-graphs.

Unbounded loop

In the list of unbounded parts, an unbounded loop is shown as follows:

```
call path to the subprogram
  Loop unbounded at srcfile:location, offset offset
```

If the loop is eternal (as defined in section 5.17), the form is

```
call path to the subprogram
  Loop unbounded (eternal) at srcfile:location, offset offset
```

The *srcfile* part is the name of the source-code file (full name or base name, according to the *-source* option). The code *location* usually shows the range of source-line numbers for the loop. It shows the machine-code address range if the option *-address* is used or if the source line numbers are unknown. The *offset* is the code-address offset from the start (entry point) of the subprogram that contains the loop to the start of the loop (the instruction at the loop head). The offset can be used to identify the loop in an assertion (see section 8.6).

Unbounded stack

For stack usage analysis (option *-stack* or *-stack_path*), an unbounded local stack height appears as follows in the list of unbounded parts:

```
call path to the subprogram
  Local stack-height unbounded for stack name : stack height
```

The concepts of “stack name” and “local stack height” are explained in section 3.11. The stack height shown in this output is an “unsafe” or “unknown” value.

Irreducible flow-graph

An irreducible control-flow graph is considered an “unbounded part” if it prevents the analysis. This is always the case for time analysis but stack usage can sometimes be analysed for irreducible graphs (by constant propagation).

In the list of unbounded parts, the irreducibility is shown as follows:

```
call path to the subprogram
  Irreducible flow-graph at source file name : code location
```

The code location shows the source line numbers and possibly the machine-code address range for the irreducible subprogram.

All call paths (-show callers)

For solving problems with unbounded loops or other program parts it is often helpful to know where and how the subprogram in question is called. The option *-show callers* adds a list of all the call-paths to the subprogram after the list of unbounded parts in the subprogram. In the example above, the list for the *Upsilon* subprogram with *-show callers* would show the two possible paths in this way:

```
Main@23=>Foo@104=>Bar@212=>Upsilon
  Loop unbounded at :[31-32]
  All paths to Upsilon:
    Main@23=>Foo@123=>Upsilon
    Main@23=>Foo@104=>Bar@212=>Upsilon
    ---
```

The line with the string "---" marks the end of the list of call-paths. When a subprogram has a context-dependent set of unbounded parts and thus appears more than once in this output, the call-paths are listed only in the first appearance.

7.4 Tabular Output

WCET break-down

The *-table* option makes Bound-T emit a tabular break-down of the WCET bound for each root subprogram. The purpose of the table is to identify the "hot spot" subprograms that consume major parts of the execution time. The table can also be useful for checking the scenario that Bound-T has identified as the worst case as the table gives an overview of which subprograms are called and how many times they are executed in total.

The table has one row for each subprogram in the call-tree, in top-down order starting from the root subprogram. Each row in the table is emitted as one basic output line with the key *Time_Table* as described in section 7.2.

To understand the structure of a *Time_Table* line, consider the worst-case execution path of the root subprogram and how a given lower-level subprogram *S* occurs in this path.

The path traverses the root subprogram and, via calls and returns, the lower-level subprograms. Some calls occur in loops and may therefore be repeated many times. The same subprogram *S* may be called from several places, from the same or different subprograms. Whenever *S* is called, its execution takes some time; part of this time is spent in *S* itself and the rest in subprograms that *S* calls. The execution time of *S* may be different for different calls, for example if *S* has a loop that depends on a parameter.

The *Time_Table* line for the subprogram *S* shows:

- the total number of times *S* is executed (called) in the root's worst-case execution path,
- the total (sum) execution time of *S* including its callees, for all these calls,
- how much of the total time is spent in *S* itself, in all these calls, and
- the range (min .. max) of the execution time of *S*, over all these calls.

To be precise, here the term “execution time” really means the upper bound on execution time (WCET) that Bound-T computes. Also, the “worst-case execution path” of the root subprogram is really the potential execution path that Bound-T considers as the worst-case path, although it may be infeasible in parts.

Example

Consider the following C program, with line-numbers in the left margin:

```
 1   void A (void);
 2   void B (void);
 3   void C (unsigned char n);
 4
 5   int A_count = 0;  /* Counts executions of A(). */
 6   int B_count = 0;  /* Counts executions of B(). */
 7   int C_count = 0;  /* Counts executions of C(). */
 8
 9   void A (void)
10   {
11      A_count ++;
12   }
13
14   void B (void)
15   {
16      B_count ++;
17      A ();
18      C (20);
19   }
20
21   void C (unsigned char n)
22   {
23      unsigned char k;
24      C_count ++;
25      for (k = 0; k < n; k++)
26      {
27         A ();
28      }
29   }
30
31   void main (void)
32   {
33      unsigned char x;
34      A ();
35      for (x = 0; x < 10; x++)
36      {
37         B ();
38      }
39      C (5);
40   }
```

You see here a trivial function *A*, which simply counts how many times it is executed, and a slightly more complex function *B*, which also counts its executions and also calls *A* and *C*(20). The function *C*(*n*) counts its executions and calls *A* in a loop that executes *n* times. At the top, the *main* function calls *A* once, then calls *B* ten times, and finally calls *C*(5). The following figure illustrates the call graph. The numbers on the call-arcs show how many times the call is executed for one call of *main*.

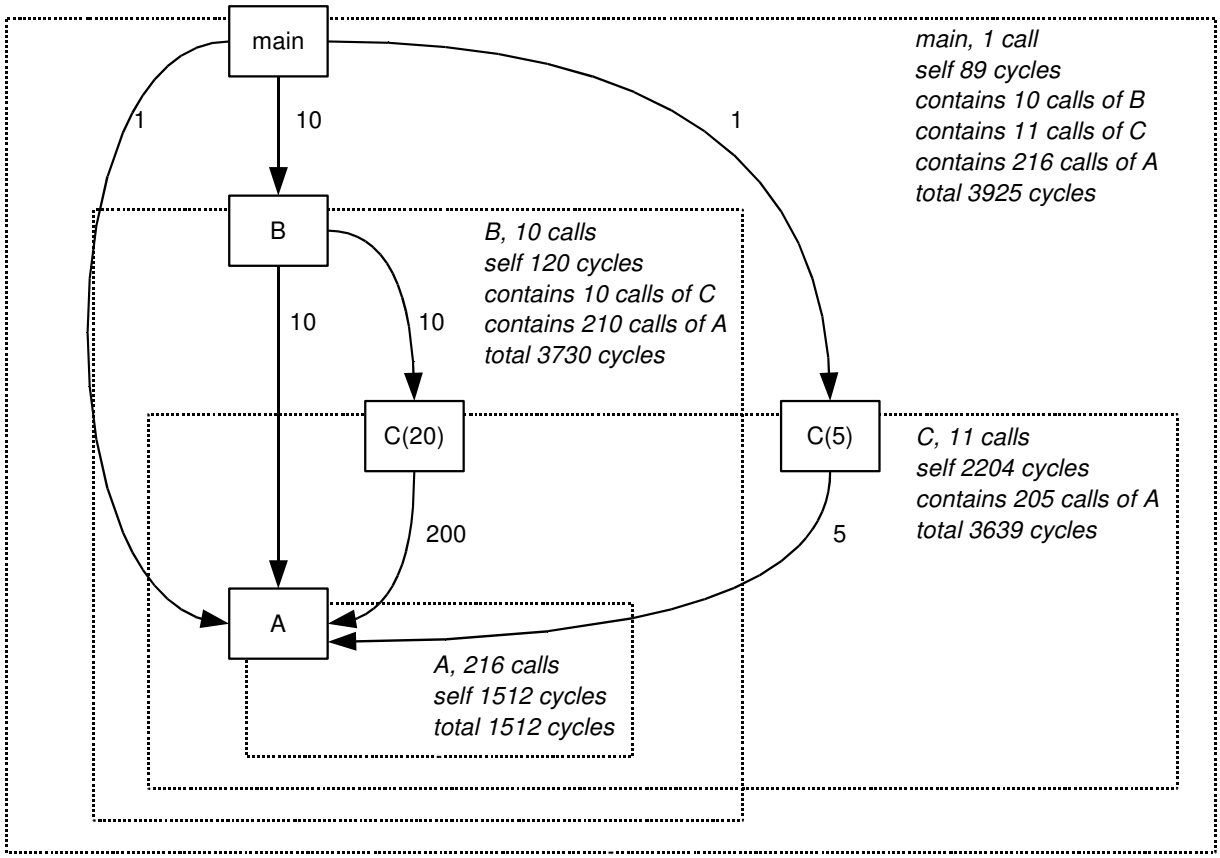

Figure 5: Call-graph for example of tabular output

Since *A* is called 216 times, the final value of the variable *A_count* in the program is 216. Compare this with the *-table* output for *A* from a WCET analysis of *main* (this was for an Intel-8051 target processor):

```
Time_Table:ex:ex.c:main:31-40:1512:1512:216:7:7:A:ex.c:9-12
```

The fields 3 through 5 (`ex.c:main:31-40`) show that this *Time_Table* line represents one row in the break-down of the WCET bound for the root subprogram *main* which lies on lines 31-40 of the source file *ex.c*.

The fields 11 through 13 (`A:ex.c:9-12`) show that this *Time_Table* line represents the row for subprogram *A*, which lies on lines 9-12 of the source file *ex.c*.

The fields 6 through 10 (`1512:1512:216:7:7`) show the role of *A* in the WCET bound of *main*:

- field 6 shows the total execution time (bound) for all the calls of *A* executed in one call of *main*: 1512 cycles.
- field 7 shows how much of the total time is spent in *A* itself; this is also 1512 cycles because *A* calls no other subprograms.
- field 8 shows how many times *A* was executed: 216 times.
- field 9 shows the smallest execution time (bound) on *A*, within these 216 executions: 7 cycles.
- field 10 shows the largest execution time (bound) on *A*, within these 216 executions: it is also 7 cycles because *A* is not context-dependent.

Since each of the 216 calls of *A* takes 7 cycles, the total time should be $7 \times 216 = 1512$, which is consistent with field 6.

Here are all the *Time_Table* output lines for *main*:

```
Time_Table:ex:ex.c:main:31-40:3925:89:1:3925:3925:main:ex.c:31-40
Time_Table:ex:ex.c:main:31-40:1512:1512:216:7:7:A:ex.c:9-12
Time_Table:ex:ex.c:main:31-40:3730:120:10:373:373:B:ex.c:14-19
Time_Table:ex:ex.c:main:31-40:3639:2204:11:99:354:C:ex.c:21-29
```

and here is a table that shows the essential fields 6 through 11 in a more readable way.

Table 15: Tabular output example

Subprogram	Total time	Self time	Calls	Min	Max	Remarks
main	3925	89	1	3925	3925	This is the root subprogram, so of course it is executed exactly once.
A	1512	1512	216	7	7	This was explained in detail above.
B	3730	120	10	373	373	A context-independent time bound.
C	3639	2204	11	99	354	Explained below.

The last row for *C* is the most interesting one. The eleven calls are made up of one call from *main* plus one call from each of ten executions of *B*. The calls $B \rightarrow C$ take longer (354 cycles) because the parameter *n* is 20. The call $main \rightarrow C$ is faster (99 cycles) because *n* is only 5.

These context-dependent bounds are also reflected in other basic output lines. The context-dependent loop-bounds in *C* are shown as:

```
Loop_Bound:ex:ex.c:B@18-=>_C:26-28:20
Loop_Bound:ex:ex.c:main@39-=>_C:26-28:5
```

The context-dependent WCET bounds appear as:

```
Wcet_Call:ex:ex.c:B@18-=>_C:21-29:354
Wcet_Call:ex:ex.c:main@39-=>_C:21-29:99
```

The total execution time for the eleven calls of *C* should thus be $1 \times 99 + 10 \times 354 = 3639$ cycles, which agrees with the total time shown on the *Time_Table* line.

Adding up the times

We have called the time-table output a "break-down" of the WCET bound for the root subprogram. However, you can easily see in the example above that the sum of the "total" execution time (bounds) of the lower-level subprograms *A*, *B*, *C* is 1512 + 3730 + 3639 = 8881 cycles, which considerably exceeds the execution time (bound) of 3925 cycles for the root subprogram *main*. How is this possible?

This happens because the time table is a *flat* representation of a *hierarchical* breakdown. The "total" time for *B* includes the execution of *A* and *C* when called from *B*, and the "total time" for *C* includes the execution of *A* when called from *C*. The sum of the "total" times thus includes some subprogram executions many times over, and so is meaningless.

The sum of the "self" times, on the other hand, is meaningful. In the example, the "self" times sum up as 89 + 1512 + 120 + 2204 = 3925 cycles which is exactly the WCET bound for the root subprogram. So the "self" times are the real break-down, while the "total" times are hierarchical sub-totals. The dotted rectangles in the call-graph figure, , show what each such sub-total includes.

Formatting script

The *Time_Table* output lines are dense and hard to read from the raw Bound-T output. The following shell script for Unix-like systems selects the *Time_Table* lines, reformats them in a more readable way and sorts them in order of descending "total" time. The script can be run as a filter on the output from Bound-T. You can find the script under the name *table.sh* in the Bound-T installation package or at *http://www.bound-t.com/scripts/table.txt*.

```
(
  echo -e "Total\tSelf\tNum\tTime Per Call"
  echo -e "Time\tTime\tCalls\tMin\tMax\tSubprogram"
  echo -e "-----\t----\t-----\t-------------\t----------"

  egrep '^Time_Table' |
  cut -d: -f6-11 |
  sort -t: -nr |
  tr ':' '\011'
) |
expand –tabs=10,20,30,40,50
```

The script assumes that the Bound-T run uses the default field-separator character, the colon. If some other character is used, put it as the first argument of the *tr* command in the script.

For the example in this section, the result of passing the raw Bound-T output through this filter is the following:

```
Total     Self      Num       Time Per Call
Time      Time      Calls     Min       Max       Subprogram
-----     ----      -----     -------------       ----------
3925      89        1         3925      3925      main
3730      120       10        373       373       B
3639      2204      11        99        354       C
1512      1512      216       7         7         A
```

Non-appearance of “integrated” subprograms

If the analysis includes subprograms that are analysed as integrated parts of their callers (as explained further in section 5.11) these subprograms do not appear as rows in the tabular output. The execution time of an integrated subprogram is included in the execution time of its callers. The number of calls to an integrated subprogram is not reported at all.

7.5 Detailed Output

The option *-show* (which needs an argument) makes Bound-T show the results of the time and/or stack-usage analysis in a "detailed" format. This is in addition to the basic output format that was described in section 7.2. The optional detailed format is mainly intended for testing and troubleshooting at Tidorum Ltd, but it can perhaps give you some insight into the nature and structure of the analysis results.

Several internal or intermediate analysis results are shown only in this detailed output and do not appear in the basic output. One example is the reachability or unreachability of flow-graph parts (*-show model*).

Output options

The detailed output can show various things as selected by the *item* arguments to the *-show* option. Chapter 6 explains the option syntax and lists the set of *items* with brief explanation. The subsections below give examples of the output for each possible *item*.

Call graph framework

The detailed output for one root subprogram is structured as a hierarchically indented display of the call-graph rooted at this subprogram, similar to the structure of the list of unbounded program parts described in section 7.3.

If the *deeply* item is selected (that is, the option *-show deeply* is used), the detailed output is structured hierarchically following the call graph. The details for one subprogram or call are headed by a line that gives the call path. These call-path lines are sequentially numbered and the sequence number is used to avoid repeating the detailed output when the same analysis results are used in several contexts.

For example, assume that the root subprogram *Main* calls the subprogram *Bar* directly as *Main* $\rightarrow$ *Bar* and indirectly along the path *Main* $\rightarrow$ *Foo* $\rightarrow$ *Bar*. The detailed output from the analysis of *Main* would be structured like this:

```
1    Main

     detailed output for the analysis of Main

2    . Main@12=>Bar

       detailed output for the analysis of Bar in this context

3    . Main@23=>Foo

       detailed output for the analysis of Foo in this context

4    . . Main@23=>Foo@102=>Bar

         detailed output for the analysis of Bar in this context
```

The periods '.' in front of the call-paths are just indentation markers.

If the analysis of *Bar* is context-dependent, so that Bound-T analyses *Bar* separately in the contexts *Main* $\rightarrow$ *Bar* and *Main* $\rightarrow$ *Foo* $\rightarrow$ *Bar*, the detailed results of these analyses are output separately after the call-path lines number 2 and 4, respectively. Otherwise, that is if Bound-T

uses the same analysis results for *Bar* in both contexts, the entire detailed output for the latter context, the call *Main* → *Foo* → *Bar*, consists of the call-path line and a reference to the first appearance of these results in line 2:

```
4    . . Main@23=>Foo@102=>Bar

         See above line 2.
```

General information (-show general)

The general information gives the fully scope-qualified name of the subprogram in question, the source-file name and source-line range, the code address range (at least under the -address option), the context (call path) on which the results depend (or "none" for context-independent analysis), and a summary line. The summary line reports if the control-flow graph is reducible or irreducible, if the control-flow paths are bounded or not, if the execution time of each flow-graph node has been computed, if the stack space is bounded or not, and if the subprogram calls some subprograms that were not analysed because assertions replaced them by "stubs".

If there are calls to stubs the "stub level" is reported; this is a number that shows the length of the shortest call-path to a stub, with 0 = the subprogram is itself a stub; 1 = the subprogram directly calls a stub; 2 = the subprogram calls a subprogram that calls a stub; and so on.

There may be zero, one or more source-file names and source-line ranges, depending on the source-to-code map that the compiler generates.

Here is an example, including the call-path line at the start and assuming context-dependent results:

```
3    . Main@23=>Foo

       Full name      : libs|Foo
       Source file    : libs.c
       Source lines   : 12-47
       Code addresses : [0A4F-0D17]
       Call context   : Main@23=>Foo

       Execution bounds # 10
       Reducible, paths bounded, time computed, space not bounded,
       calls no stubs.
```

The "execution bounds" number (10 in the above example) is an internal index that you can ignore.

Time and space bounds (-show bounds)

The option *-show bounds* includes the execution time bound and the stack usage bound in the detailed output. However, the bound is shown only if the corresponding analysis is done. Assuming that both *-time* and *-stack* analysis is selected, the output appears as follows:

```
3    . Main@23=>Foo

       WCET: 124
```

```
Local stack height for P-stack: 12
Total stack usage  for P-stack: 44
```

If no execution time bound is known, the WCET line appears as “WCET is unknown”. If the WCET for this subprogram or call is asserted, the WCET line appears as “WCET: 124 (asserted)”.

The lines for the local stack height and total stack usage are repeated for each stack in the target processor.

Loop bounds (-show loops)

The option *-show loops* includes in the detailed output the computed or asserted repetition bounds for each loop in the subprogram. For example:

```
3    . Main@23=>Foo

       Loop 1 : libs.c:22-31
         Repeat <= 8
       Loop 2 : libs.c:35-36
         Neck <= 16
```

The loops are numbered in an arbitrary way except that the number of an inner loop is smaller than the number of an outer loop. The source-file name and source-line numbers of the loop are shown, if known.

A computed (automatically determined) repetition bound is shown as “Repeat <= n”. An asserted bound is shown as “Neck <= n”. Additional output lines show if the loop is unbounded or eternal.

Callers (-show callers)

The option *-show callers* adds to the detailed output a list of all call-paths from some root subprogram to the current subprogram. This is sometimes called the inverse call tree. For example:

```
2    . Main@12=>Bar

       All paths from a root to Bar:
         Main@12=>Bar
         Main@23=>Foo@102=>Bar
         ---
```

The line with the string “---” marks the end of the list of call-paths. The call-paths to a given subprogram are listed only in the first appearance of the detailed output for this subprogram.

Execution counts (-show counts)

The option *-show counts* adds to the detailed output a table that shows how many times the worst-case execution path executes each part of the control-flow graph. These results are available only after a successful analysis for WCET. The table also shows which parts of the flow-graph are reachable (feasible) or unreachable (see the discussion of flow-graph pruning in section 6.5).

As elsewhere in Bound-T, the term “node” in this output means a basic block of the flow-graph and the term “step” means a flow-graph element that models a single instruction or sometimes a part of an instruction. Thus, a node consists of a sequence of steps, as this table also shows.

Here is an example showing the execution counts in the subprogram *Simple* when called from the root msubprogram *Main*:

```
2    . Main@12=>Simple

       Execution counts of nodes and edges:

         A '+' means feasible, a '-' means infeasible.

         Node      Count   Steps in node
         + 1        1      1 2
         - 2        0      3
         + 3        32     4 5
         + 4        32     6
         + 5        1      7

         Edge      Count   S -> T
         - 1        0      1 -> 2
         + 2        31     4 -> 3
         + 3        1      4 -> 5
         + 4        32     3 -> 4
         + 5        0      3 -> 5
         + 6        1      1 -> 3
```

The above table shows a control-flow graph with seven steps, numbered 1 to 7, collected into five basic-block nodes numbered 1 to 5. Node 1 contains the steps 1 and 2 (in that order), node 2 contains just the step 3, and so on until node 5 that contains just the step 7. In the execution path that Bound-T considers to be the worst case (take the longest time), nodes 1 and 5 are executed once while nodes 3 and 4 are executed 32 times; node 2 is not executed at all. In fact, the '-' sign at the start of the line for node 2 means that this node was found to be unreachable, so of course it is not executed.

There are six edges between the nodes, numbered 1 to 6. Edge 1 goes from node 1 to node 2, edge 2 goes from node 4 to node 3, and so on until edge 6 which goes from node 1 to node 3. In the worst-case path, edges 1 and 5 are not executed at all (in fact edge 1 is unreachable). Edges 3 and 6 are executed once. Edge 2 is executed 31 times and edge 4, 32 times.

Note that edge 5 is considered reachable, but since it is not executed in the worst-case path it probably represents a quicker execution path, for example an early exit from the loop that contains nodes 3 and 4.

To find out which machine instructions correspond to each step and node, use the option *-trace decode* to generate a disassembly including step numbers, and use the option *-trace nodes* to generate a list of nodes with code locations.

The execution counts are also shown in the flow-graph drawing generated with the options *-dot* and *-draw*. See section 7.6.

Computation model (-show model)

The option *-show model* adds to the detailed output a presentation of the “computation model” for the current subprogram in the current context. The computation model shows the arithmetic effect (computations and assignments to variables) of each step in the control-flow graph, the logical precondition of each edge in the graph, and whether the step or edge is

reachable or unreachable (see the discussion of flow-graph pruning in section 6.5). The model also associates each call in the current subprogram with a “calling protocol” that can have context-dependent attributes.

The model is displayed as three tables: a list of all steps with their arithmetic effects; a list of all edges with their source and target steps and logical preconditions, and a list of all calls that shows the step in which the call occurs, the caller and callee, and the calling protocol with its attributes.

As elsewhere in Bound-T, the term “step” means a flow-graph element that models a single instruction or sometimes a part of an instruction.

Here is an example of the detailed output of the computation model for a root subprogram called *TempCon*. The output is rather long so we insert an explanation of each of the three tables just after the table in question.

```
1    TempCon

     Computation model:

       References to this model: 1
       A '+' means feasible, a '-' means infeasible.

       Step  Effect
       + 1   p = (p+ 1), Loc1 = p, SH =  2
       + 2
       - 3   n = (n- 1)
       + 4
       + 5
       + 6   r = ?
       + 7   r = Loc1, p = (p+-1), SH =  1
       + 8
       + 9   b = ?, c = ?
       + 10
```

The table above lists the ten steps in the control-flow graph of *TempCon* and shows the arithmetic effect of each step. The effect is a list of assignments of the form *cell* = *expression*. The cells are registers, flags or memory locations; their names depend on the target processor and usually have no relation to the source-code variable names.

In this example, the arithmetic effect of step 1 increments cell *p*, assigns the value of cell *p* to the cell *Loc1*, and assigns the value 2 to the cell *SH*. All the assignments in one step are done in parallel so that one first evaluates all the expressions using the original values of the cells and then assigns the new values to the target cells. This means that step 1 assigns the original (non-incremented) value of *p* to *Loc1*, not the new, incremented value.

Some steps, here for example step 2, have no arithmetic effect. Often such steps model jump instructions.

When some effect of a machine instruction is not modelled for some reason (too complex or not interesting for the analysis) it is represented by a question mark '?' and considered to have an unknown value. In this example, step 6 assigns an unknown value to cell *r* and step 9 assigns unknown values to the cells *b* and *c*.

Reachable steps are shown by '+' signs and unreachable (infeasible) steps by '-' signs. In this example, only step 3 is unreachable.

The detailed output for *-show model* continues with a table of the flow-graph edges:

```
Edge   S -> T     Precondition
+ 1    1 -> 2     true
- 2    2 -> 3     false
- 3    3 -> 4     false
+ 4    2 -> 4     true
+ 5    5 -> 6     true
+ 6    4 -> 5      not ((a>b))
+ 7    6 -> 7     true
+ 8    4 -> 7     (a>b)
+ 9    8 -> 9     true
+ 10   7 -> 8     true
+ 11   9 -> 10    true
```

The above table lists the 11 edges between steps in the flow-graph for *TempCon*. Reachable edges are marked '+' and unreachable (infeasible) edges are marked '-'.

For each edge, the column headed “S -> T” shows the numbers of the source step and the target step. For example, edge number 8 goes from step number 4 to step number 7. A comparison to the table of steps shows that the unreachable edges, edges 2 and 3, are connected to the unreachable step 3, which is consistent. However, in general there may also be unreachable edges between reachable steps.

For each edge, the column headed “Precondition” shows the logical condition that must hold if the edge is taken (executed). The value “true” indicates an unconditional edge or an edge with an unknown precondition and “false” indicates a false precondition which is the same as an unreachable edge. Otherwise, the precondition is a relation between arithmetic expressions composed of cell values. For example, after executing step 4, the target program can take edge 8 only if the value of cell *a* is greater than the value of cell *b* at this time.

Note that the precondition is only a *necessary* condition for taking the edge, but it may not be a *sufficient* one. Thus, a “true” precondition does not mean that the edge *is* always taken; it means that the edge *can* always be taken, as far as the analysis knows.

The last part of the detailed output for *-show model* is a table of all calls from *TempCon* to other subprograms:

```
Call  Step  Caller=>Callee           Protocol
+ 1    6    TempCon@[95]=>ReadTemp   Stack, SH=2
+ 2    9    TempCon@[97]=>Heat       Stack, SH=1
```

This table shows the two calls from *TempCon* to *ReadTemp* and *Heat*, respectively. The “Step” column shows the number of the step that models the call in the control-flow graph of *TempCon*. These “call steps” are special steps that model the entire execution of the callee but do not correspond to any machine instruction in the caller. Bound-T inserts such a call step in the caller's flow-graph immediately after the step that models the real call instruction.

The column headed “Caller=>Callee” shows the caller (*TempCon*), the code location of the call (in this example as code addresses only) and the callee (*ReadTemp* and *Heat*, respectively).

The “Protocol” column shows the calling protocol associated with the call. In this example, both calls use the “Stack” protocol, but with different attributes: the *SH* attribute has the value 2 for the first call and the value 1 for the second call. The names and attributes of the calling protocols depend on the target processor and possibly also on the target programming tools (cross-compiler and linker).

This completes the example and description of *-show model*.

Time per node (-show times)

The option *-show times* adds to the detailed output a table showing the execution time of each node (that is, a basic block) in the control-flow graph. The total time per node is broken down into the "local" time, consumed by instructions in the current subprogram, and the time consumed by other subprograms called from this node (callees).

Here is an example of the detailed *-show times* output for the root subprogram *TempCon*:

```
1    TempCon

     Execution time of each node, in cycles:

       Node    Total  =  Local  +  Callees
       1       2          2          0
       2       1          1          0
       3       1          1          0
       4       1          1          0
       5       12         0          12
       6       2          2          0
       7       25         0          25
       8       1          1          0
```

The table shows 8 nodes and their execution time in cycles. For example, node 1 uses 2 cycles itself (that is, the instructions in *TempCon* that belong to node 1 take 2 cycles to execute) and does not call other subprograms. Node 7 contains no instructions from *TempCon* (or these instructions take no time to execute) but calls some other subprogram that take 25 cycles to execute.

In fact, since Bound-T currently makes each call into its own node that contains no instructions from the caller, the times in the columns "Local" and "Callees" are never both positive. The "Callees" time is zero for ordinary nodes and the "Local" time is zero for nodes that contain a call.

The time per node is known only from execution-time analysis. If you combine the options *-no_time* and *-show_times*, the output will be "Execution times of nodes not known".

Stack usage per call (-show spaces)

The option *-show spaces* adds to the detailed output information about the stack usage at various points within the control-flow graph of each subprogram.

At present, this information is limited to the take-off height for each call. As defined in section 3.11, the take-off height for a call is the local stack height in the caller, immediately before control is transferred to the callee. However, if the call instruction pushes a return address on the stack, this is usually considered a part of the callee's stack usage so it does not contribute to the take-off height.

The take-off height is reported in target-specific units, explained in the relevant Application Note. If the target program uses several stacks, the output contains a separate table of take-off heights is for each stack.

Here is an example of the detailed *-show spaces* output for the root subprogram *TempCon*:

```
1 TempCon

  Take-off stack-heights at calls for P-stack:

    Call  Caller=>Callee          Height
    1     TempCon@[95]=>ReadTemp  2
    2     TempCon@[97]=>Heat      1, leads to maximum stack usage.
```

The table list the two calls from *TempCon* to *ReadTemp* and *Heat*, respectively. The "Height" column shows the take-off height of the "P-stack" for the call. The comment after the take-off height for call 2 indicates that this call is on the worst-case stack-usage path for *TempCon*. Although the take-of height of this call is less than for call 1, the callee (*Heat*) uses so much stack that call 2 was chosen for the worst-case stack path.

Some or all of this information is known only through specific stack usage analysis, that is, if you include the option *-stack* or *-stack_path*.

At the moment there is no way to make Bound-T display the *total* stack usage for each call. The *-stack_path* option displays this only for the calls on the worst-case stack-usage path.

Final stack height (-show stacks)

The option *-show stacks* adds to the detailed output a table that shows the final local stack height on return from each subprogram, for each stack in the target program. Since the local stack height is by definition zero on entry to a subprogram the final height also shows the net effect that the subprogram has on the stack height, that is, if there are more pushes than pops (positive final height) or more pops than pushes (negative final height).

Bound-T analyses the subprograms for final local stack height even if stack usage analysis is not requested (with the options *-stack* or *-stack_path*) because the local stack height must be known for tracking references to data in the stack.

Here is an example of the detailed *-show stacks* output for the root subprogram *TempCon*:

```
1 TempCon

  Final stack height on return from subprogram:

    Stack          Final height
    P-stack        -1
```

The table shows that the final local height of the stack "P-stack", on return from *TempCon*, is -1, which means that the *TempCon* subprogram pops one element more than it pushes. Perhaps this popped element is the return address.

Input and output cells (-show cells)

The option *-show cells* adds to the detailed output information about how each subprogram uses "storage cells" in its computation. Here a storage cell means any memory location or register in the target processor that can hold an arithmetic value and that Bound-T models in its analysis. The information is output as three lists of cells:

- Input cells. These are the cells that the subrogram reads (uses) before it writes a new value in the cell. Such cells may be input parameters or global variables on which the subprogram depends.

- Basis cells. These are all the cells included in the Presburger-arithmetic model of the subprogram's computation.
- Output cells. These are the cells that the subprogram can write new values to. However, only statically identified cells are listed; if the subprogram writes via dynamic pointers, it can modify any storage cell that the pointer can reference.

There is a fourth list that shows the initial bounds (value ranges) that are known for some cells at the start of the subprogram. Such a bound can be derived from the calling context, from an assertion or from the calling protocol.

All cells in the output are named from the point of view of the current subprogram. This means that the output may include cells that are private (local) to the subprogram, such as local variables in the subprogram's call-frame in the stack.

Here is an example of the detailed *-show cells* output for the subprogram *ReadTemp* that is called from the root subprogram *Calibrate*:

```
2    . Calibrate@[105]=>ReadTemp

         Input cells:
           k

         Basis cells for arithmetic analysis:
           k
           r

         Initial cell bounds on entry:
           k=10
           SH=1
           ZSH=0

         Output cells:
           r
```

The list of *input* cells shows that *ReadTemp* uses the initial value of the cell *k* in some way that is important to the analysis (for example, as a loop bound). The list of *basis* cells shows that also the cell *r* is used in such a way, but the fact that *r* is not an input cell means that *ReadTemp* always assigns a value to *r* before using *r*, so the initial value of *r* is not relevant. The Presburger arithmetic model tracks the basis cells *k* and *r*.

The list of initial cell bounds shows that the call-context in *Calibrate* (or some applicable assertion) constrains the input cell *k* to the value 10, which means that the analysis for *Calibrate* → *ReadTemp* has all the values that can be useful for the automatic loop-bound analysis.

The initial bounds on the cells *SH* and *ZSH* are not useful in this case, as they are not input cells for *ReadTemp*. (In this example, these bounds are derived from the calling protocol; the cells in question show the local stack height of the two stacks.)

The list of *output* cells shows that *r* is the only (statically identified) cell to which *ReadTemp* assigns a value. However, this does not always mean that the caller can see a change in the cell's value; it may be that the cell is private to *ReadTemp* and not visible to the caller, or *ReadTemp* may save the original value of the cell and then restore it before returning to the caller.

7.6 DOT Drawings

This section explains the function of the *-dot* and *-dot_dir* options and the form of the resulting drawings.

The -dot option and the dot tool

The options *-dot*, *-dot_dir* and *-draw* make Bound-T draw call-graphs and control-flow graphs of the subprograms it analyses. The drawings are created as text files in a syntax suitable for the *dot* tool, part of the *GraphViz* package available from *http://www.graphviz.org/*. The *dot* tool can lay out the drawings in Postscript or other graphic formats for display by a suitable viewer tool.

For example, a Bound-T command of the form

```
boundt -dot graph.dot ...
```

creates the file *graph.dot*, which contains text in the *dot* syntax to define the logical structure and labelling of the drawings created by Bound-T. To lay out the drawings as a PostScript file, for example *graph.ps*, you may then use the following command:

```
dot -Tps <graph.dot >graph.ps
```

The -dot_dir option and the names of drawing files

The *-dot* option creates a single file that contains all drawings from one Bound-T run. If you then use the *dot* tool to create a PostScript file, each drawing will go on its own page in the PostScript file. However, *dot* can also generate graphical formats that do not have a concept of "page" and then it may happen that only the first drawing is visible. If you want to use such non-paged graphical formats it is better to create a directory (folder) to hold the drawing files and use the Bound-T option *-dot_dir* instead of the option *-dot*. The *-dot_dir* option creates a separate file for each drawing, named as follows:

- The call-graph of a root subprogram is put in a file called *cg_R_nnn.dot*, where *R* is the link-name of the root subprogram, edited to replace most non-alphanumeric characters with underscores '_', and *nnn* is a sequential number to distinguish root subprograms that have the same name after this editing.
- If the call-graph of some root subrogram is recursive, Bound-T draws the joint call-graph of all roots and puts it in a file called *jcg_all_roots_001.dot*.
- The flow-graph of a subprogram is put in a file called *fg_S_nnn.dot*, where *S* is the link-name of the subprogram, edited as above, and *nnn* is a sequential number to distinguish subprograms that have the same name after this editing and also to distinguish drawings that show different flow-graphs (execution bounds) for the same subprogram.

The sequential numbers *nnn* start from 1 and increment by 1 for each drawing file; the same number sequence is shared by all types of drawings and all subprograms. For example, if we analyse the root subprogram *main?func* that calls the two subprograms *start$sense* and *start$actuate*, with the *-dot_dir* option and *-draw* options that ask for one flow-graph drawing of each subprogram, the following drawing files are created:

- *cg_main_func_001.dot* for the call-graph of *main?func*
- *fg_main_func_002.dot* for the flow-graph of *main?func*
- *fg_start_sense_003.dot* for the flow-graph of *start$sense*

- *fg_start_actuate_004.dot* for the flow-graph of *start$actuate*.

Call graphs

Figure 6 below is an example of a non-recursive call-graph drawing. The rectangles represent subprograms and the arrows represent feasible calls from one subprogram to another. This call-graph shows the root subprogram *main* calling subprograms *Count25*, *Foo7*, *Foo* and *Extract*, some of which in turn call *Count* and *Ones*.

Each rectangle is labelled with the subprogram name (in this example the compiler adds an underscore '_' before the names), the number of times the subprogram is called, the number of call paths, the execution time in the subprogram itself and the execution time of its callees. The number of calls and the execution times refer to the execution that defines the worst-case execution-time bound for the root subprogram.

In this example call-graph most subprograms have context-independent execution bounds (same WCET bound for all calls). The exception is the subprogram *Count* which has some context-dependent bounds. This can be seen from the annotation for the execution time of *Count*: “time 1469 = 10 * 94 .. 158” which means that the execution time bounds for one call of *Count* range from 94 cycles to 158 cycles, depending on the call path, such that the total bound for the 10 executions of *Count* is 1469 cycles.

Call-graph drawings can become quite cluttered and hard to read when some subprograms are called from very many places. For example, programs that do lots of trigonometry can have numerous calls to the *sin* and *cos* functions. You can use “hide” assertions to omit chosen subprograms from the call-graph drawing; see section 8.5.

When a subprogram calls several other subprograms the left-to-right order of the arcs that represent these calls in the call-graph drawing is arbitrary. The order in the drawing says nothing about the order of the calls in the source code nor about their order of execution.

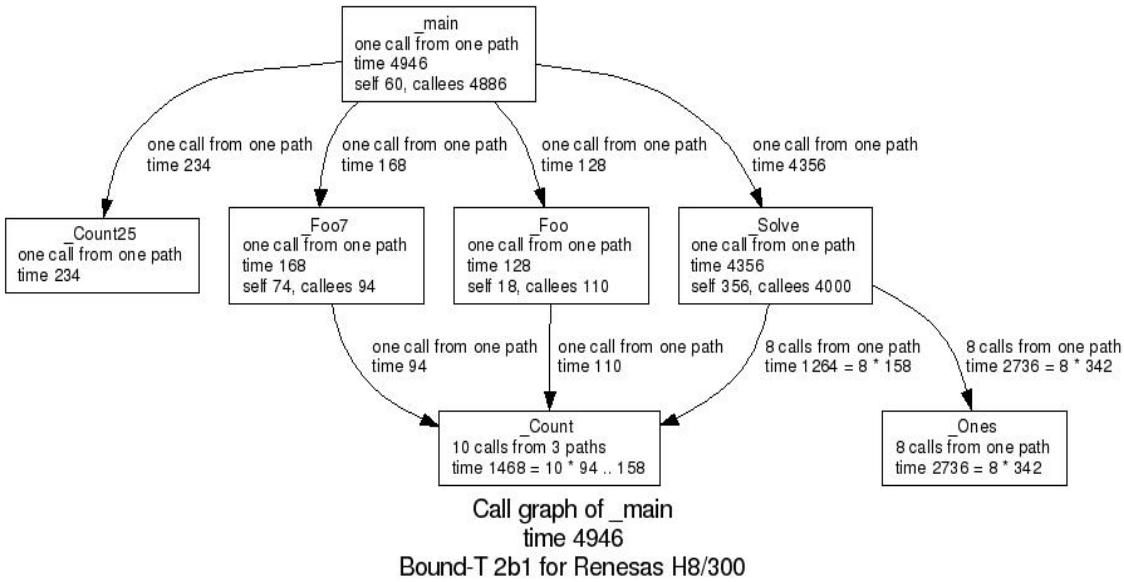

Figure 6: Example non-recursive call graph

Recursive call graphs

If the call-graph of some root subprogram is recursive, Bound-T draws the joint call-graph of all root subprograms in a special form. Figure 7 below is an example of a recursive call-graph drawing. The rectangles represent subprograms and the arrows represent calls from one subprogram to another. This call-graph shows the root subprogram *fnoo* calling subprograms *emak* and *glop*. There are two recursive cycles, one between *fnoo* and *glop* and another that contains all three subprograms.

Each rectangle is labelled with the subprogram name. Each arrow is labelled with the number of call sites (note, not the number of dynamically executed calls). In the example, *fnoo* contains one call of *glop* but two calls of *emak*. There is no information on execution time bounds, number of executions and so on because Bound-T cannot analyse recursive programs.

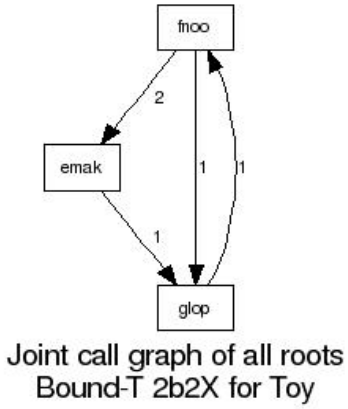

Figure 7: Example recursive call graph

Flow graphs

Figure 8 below is an example of a control-flow graph drawing that shows a subprogram called *Count*. This is the same subprogram *Count* that appeared in the call-graph example above. The rectangles are the basic blocks (nodes) of the code in *Count*, that is, sequences of instructions that do not branch and are not entered in the middle. The arrows, or graph edges, represent the flow of control between basic blocks.

The *entry* node is the rectangle at the top, the node that is entered by an arrow that does not start from another node but from a text label. The label shows the name of the subprogram and an abbreviation of the call-paths from the root subprogram to this subprogram.

A node that is not the start of any edge is a *return* node; the subprogram returns to its caller after executing a return node. In this example there is one return node (the bottom one), but in general there can be zero, one or several.

Flow-graph drawings include both feasible and infeasible nodes and edges. However, they stop at calls of subprograms that are known or asserted not to return to the caller or are asserted as “not used”.

The textual labels in the nodes and on the edges show the execution bounds for the subprogram. There are two forms depending on the *-draw* option that was used:

– show the total or summary of all execution bounds for this subprogram: *-draw total*

– show a single set of execution bounds for this subprogram: all other *-draw* options.

The example shows a summary flow-graph with *-draw total*. Specific items under the *-draw* option can add or remove information; the example shows the default information.

The option *-draw line* (which is included in the defaults) labels each node with the corresponding source-line numbers (when known). In this example some source-line numbers are known for the entry node (lines 41 and 42 in the file `routines.c`), the body of the loop (lines 42 and 44 in the same file) and the return node (line 46 in the same file).

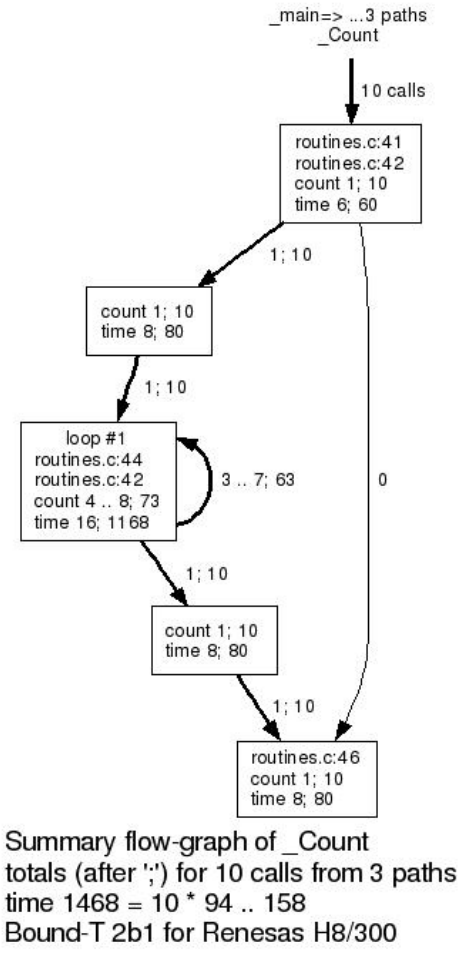

Figure 8: Example control-flow graph

The option *-draw count* (which is included in the defaults) labels each node and edge with its execution count which is the number of times this node or edge is executed in the worst-case execution path as determined by Bound-T. Moreover, edges on the worst-case path are drawn with a thick line and the other edges with a thin line (and labelled with zero executions).

For a *-draw total* drawing the execution counts are given as two numbers separated by a semicolon. The first number is the execution count per execution of this subprogram. The second number is the total execution count over all executions of this subprogram included in

the worst-case execution path of the root subprogram. In this example, the entry node is labelled with “count 1; 10” which means that the entry node is executed once for every call of this *Count* subprogram and is executed a total of 10 times within the worst-case execution of the root subprogram *main* (evidently because *Count* is called 10 times).

On the other hand, the node that forms the body of the loop is labelled with “count 4 .. 8; 73” which means that it is executed between 4 and 8 times for every call of *Count* (depending on the call path; this subprogram has context-dependent bounds) and a total of 73 times in the worst-case execution of *main*.

The option *-draw time* (which is included in the defaults) labels each node with its execution time, in target-specific units, usually processor cycles or clock cycles. For a *-draw total* drawing the execution times are given as two numbers separated by a semicolon. The first number is the worst-case time for one execution of this node. The second number is the total execution time consumed by this node over all executions of this subprogram included in the worst-case execution path of the root subprogram. In this example, the loop node is labelled with “time 16; 1168” which means that it takes 16 cycles to execute once, while 1168 cycles is the total execution time spent in this node, within the worst-case execution of *main*. This agrees with the total execution count of 73 because $1168 = 73 \times 16$.

Flow graphs with calls

Figure 9 below shows another flow-graph drawing, this time of the (very simple) subprogram *Foo*.This subprogram calls subprogram *Count*. The middle node in the diagram represents this call, as shown in the first line of the node label. The last line in the call-node label shows the execution time of the callee, in this case 110 cycles for this call of *Count*.

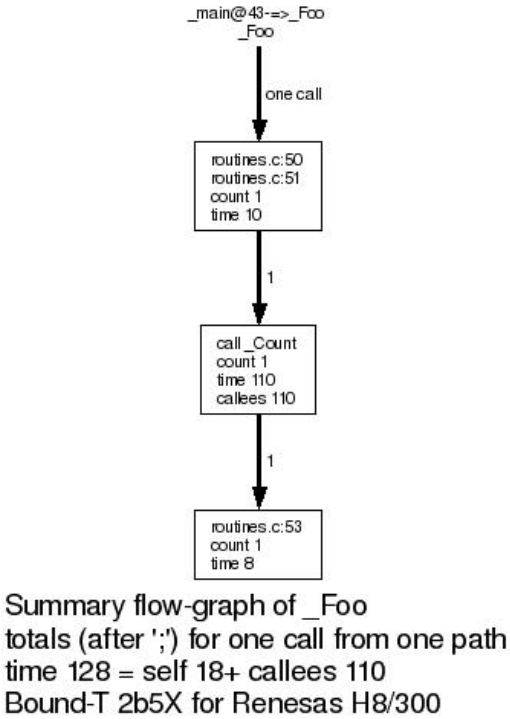

Figure 9: Example control-flow graph with call

8 ASSERTION LANGUAGE SYNTAX AND MEANING

8.1 Introduction

The command-line option *-assert filename* makes Bound-T read the assertions from the text file by the name *filename*. Chapter 5 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.

8.2 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 5.13.

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*
call	**calls**
contain	**contains**
cycle	**cycles**
define	**defines**
is	**are**
loop	**loops**
repeat	**repeats**
time	**times**
use	**uses**

Keyword pair	*Obsolete but equivalent keyword*
call to	**call_to**
end call	**end_call**
end loop	**end_loop**
is in	**is_in**
loop that	**loop_that**
no arithmetic	**no_arithmetic**

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

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

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

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

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 5. 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.

8.3 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 5.12 and 8.2.

Scope delimiter definitions

Scope_Delimiter → **delimiter** *character*

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

Scope definitions

Scope → **within** *string*

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

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

```
within "API"
```

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

8.4 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 8.6) 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.

8.5 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
            |  return
            |  integrate
            |  unused
            |  used
            |  hide
            |  not Sub_Option
```

The **arithmetic** option can locally override the command-line option for arithmetic analysis (*-arithmetic* or *-no_arithmetic*, see section 6) and the automatic decision that checks if arithmetic analysis is needed for a particular subprogram.

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.

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

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

The **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 any callees of a hidden subprogram are not automatically hidden, too; they may need their own **hide** options.

The **no** keyword negates the option setting. It 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 **integrate** property cannot be negated (disabled); it can only be asserted (enabled).

8.6 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 8.7 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 16 below defines exactly the meaning of each type of loop property.

Table 16: Meaning of loop properties

Property	Loop L has this property if and only if:
contains Count Other_Loop	Within the set of all inner loops directly contained in *L*, the given Count of loops matching the *Loop_Description*.
is in Other_Loop	*L* is directly contained in an outer loop that matches the *Other_Loop* description.
is not in Other_Loop	Either *L* is not contained in any outer loop, or the loop that directly contains *L* does not match the *Other_Loop* description.
contains Count Other_Call	*L* (or some inner loop) contains the given Count of calls matching *Other_Call*.
calls Sub_Name	Same as "contains >= 1 calls to *Sub_Name*".
uses Variable_Name	*L* (or some inner loop) contains an instruction that reads (uses) the value of the variable identified by *Variable_Name*.
defines Variable_Name	*L* (or some inner loop) contains an instruction that writes (assigns a value) to the variable identified by *Variable_Name*.
is labelled Label_Name	*L* (or some inner loop) contains the instruction that has the code address assigned to *Label_Name*.
is not labelled Label_Name	Neither *L* nor any of its inner loops contains the instruction that has the code address assigned to *Label_Name*.
executes code-address	*L* (or some inner loop) contains the instruction that has the given code address.
executes offset code-offset	*L* (or some inner loop) contains the instruction at the given offset from the start (entry point) of the subprogram that contains the loop. This form can be used only when the containing subprogram is given, that is, within a *Sub_Block*.
not Loop_Property	The negation (logical complement) of the *Loop_Property*.

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.

8.7 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 17 below defines the meaning of each type of call property.

Table 17: Meaning of call properties

Property	Call C to Sub_Name has this property if and only if:
is in Other_Loop	*C* is contained in a loop that matches the *Other_Loop* description. Note that this loop is not necessarily the innermost loop that contains *C*.
is not in Other_Loop	*C* is not contained in any loop, or no loop that contains *C* matches the *Other_Loop* Description. Note that the test applies also to outer loops, not just to the innermost loop that contains *C*.
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 *Call_Property.*

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** ...".

8.8 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 ;
        |  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 5. 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 18: 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				+	+	+

Asserted fact	Global	Subprogram entry	Subprogram body	Loop	Static call	Dynamic call
Execution time 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 te 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.

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

8.9 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 19: 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 analysis, it is not enough to assert a WCET for every call of the subprogram; you must assert a WCET for the whole subprogram, and then you can assert other WCET values for specific calls if you wish.

8.10 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 20: 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 10 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.

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.

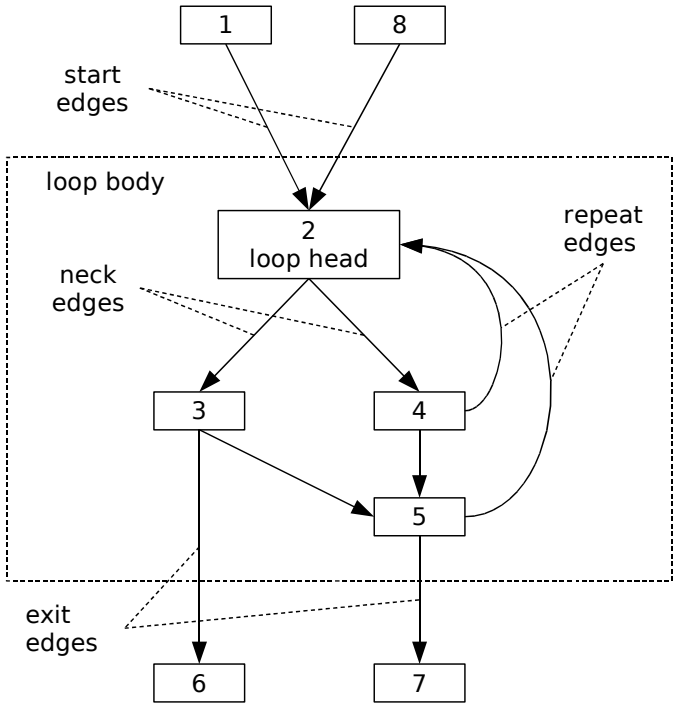

Figure 10: A loop in a flow-graph

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.

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.

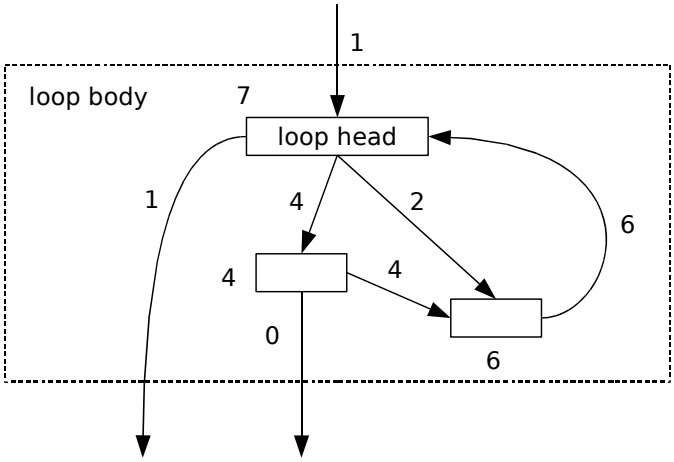

Figure 11: A general kind of loop asserted to repeat 6 times

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 12 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.

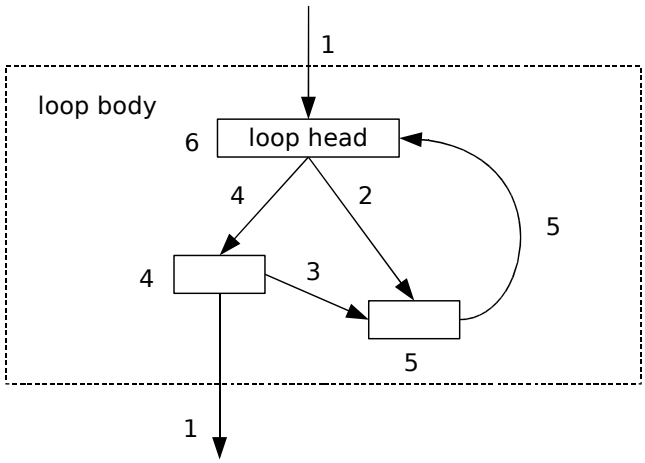

Figure 12: A middle-exit loop asserted to repeat 6 times

Figure 13 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.

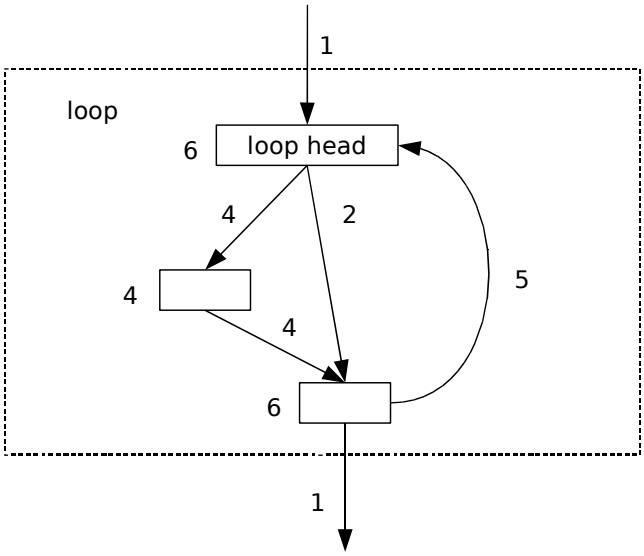

Figure 13: An exit-at-end loop asserted to repeat 6 times

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.

8.11 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 21: 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 5.8 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.

8.12 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 5.9 explained.

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

Table 22: 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.

8.13 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 23: Meaning of property value assertion

Context	Assertion holds:
Globally	During the entire analysed execution at every reached point, unless overridden by an assertion on this property in another context.
Subprogram body	For any execution of this subprogram and at all points in the subprogram, unless overriden by an assertion on this property in a loop or call within this subprogram.
Loop	For any execution of this loop and at all points in the loop, unless overridden by an assertion on this property in an inner loop or a call within this loop.
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).

8.14 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.

8.15 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 24: 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 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.
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 counds

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 occcurs Bound-T is usually unable to decide if the reason is in the assertions or in the logic of the target program itself. Bound-T classifies the relevant program part as unreachable (and warns about it, if the option *-warn reach* is in effect).

Note that Bound-T does not search for such conflicts systematically; it discovers them only if the relevant program part needs some analysis, for example loop analysis. Thus, an analysis with contradictory assertions can succeed without discovery of the conflicts, but the conflicts may be revealed in a later re-analysis with a changed target program if there is now a loop, for example, that is covered by the conflicting assertions and relationships.

8.16 Error Messages from the Assertion Parser

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 section 7.2. 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 section 9.2. 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.

The assertion parser can also issue warning messages. These are few enough to be listed together with the other Bound-T warnings in section 9.1.

Table 25: Assertion error messages

Error Message		Meaning and Remedy
Ambiguous label name: *N* *or* Ambiguous subprogram name: *N* *or* Ambiguous variable name: *N*	*Problem*	The assertion tries to identify a label, subprogram, or variable by the name *N* but the name matches more than one label, subprogram, or variable (respectively) in the program and so is ambiguous.
	Reasons	The program contains more than one label, subprogram, or variable (respectively) with the name *N* but in different scopes. The name in the assertion does not specify the scope (well enough).
	Solution	Add scope levels to the name to make it unambiguous.
"*S*" expected, at "*T*"	*Problem*	At the end of a subprogram block, for a subprogram identified by *S*, the subprogram identifier is repeated in the form *T* which does not match *S*. See also the error message that begins “Mismatch...”.
"*A*" is not a valid cell address	*Problem*	An assertion contains the string *A* that is meant to denote a variable (storage cell) address, but is rejected by the assertion parser.
	Reasons	The string *A* is not written according to the rules for variable addresses that Bound-T uses for this target processor.
	Solution	Refer to the Application Note for this target and correct the string.
"*A*" is not a valid code address, at "*T*".	*Problem*	An assertion contains the string *A* that is meant to denote a code address, but is rejected by the assertion parser. The next token is *T*.
	Reasons	The string *A* is not written according to the rules for code addresses that Bound-T uses for this target processor.
	Solution	Refer to the Application Note for this target and correct the string.

Error Message		Meaning and Remedy
Assertion expected, at "*T*".	*Problem*	The current token *T* cannot be the start of an assertion, as would be expected here. See the nonterminal *Assertions* in section 8.2. The assertion parser silently skips the following text until it finds the start of the next assertion.
Assertion file contained errors.	*Problem*	Some errors were noted in the assertion file (and reported by the corresponding other error messages in this table). The analysis stops (after reading the rest of the assertion files, if any).
Assertion file could not be read.	*Problem*	The assertion file could not be opened because the user does not have read access to the file. The analysis stops (after reading the rest of the assertion files, if any).
Assertion file was not found.	*Problem*	The assertion file could not be opened because it seems not to exist. The analysis stops (after reading the rest of the assertion files, if any).
A “that” part is not allowed for this call.	*Problem*	This *Other_Call* structure is not enclosed in parentheses and therefore it cannot include a list of call properties introduced with the **that** keyword. See section 8.7.
	Solution	Add parentheses around the *Other_Call*, thus allowing a full *Call_Description*.
A “that” part is not allowed for this loop.	*Problem*	This *Other_Loop* structure is not enclosed in parentheses and therefore it cannot include a list of loop properties introduced with the **that** keyword. See section 8.6.
	Solution	Add parentheses around the *Other_Loop*, thus allowing a full *Loop_Description*.
Bound expected, at "*T*"	*Problem*	The assertion file should have a bound here, instead of the token *T*. See the nonterminal *Bound* in section 8.2.
"call" after "end" expected, at "*T*"	*Problem*	The keyword **call** should here follow the keyword **end**. See the nonterminal *Call_Block* in section 8.7.
Call properties expected, at "*T*"	*Problem*	The assertion file should have a call property here, instead of the token *T*. See the nonterminal *Call_Property* in section 8.7.
Calls do not have the "defines" property.	*Problem*	The assertion file tries to use the **defines** keyword to identify a call. This property is not yet supported for calls.
Calls do not have the "uses" property.	*Problem*	The assertion file tries to use the **uses** keyword to name a identify. This property is not yet supported for calls.
Cannot assert callees for a static call.	*Problem*	The assertion file tries to assert the possible callees for a static call, which is nonsense. In other words, there is a *Callee_Bound* in a *Call_Block* for a static call.
Clause expected, at "*T*"	*Problem*	The assertion file should have an assertion clause here, instead of the token *T*. See the nonterminal *Clause* in section 8.8.
Clause or "end call" expected, at "*T*"	*Problem*	The assertion file should have an assertion clause here, or the **end call** keywords, instead of the token *T*. See the nonterminals *Clause* in section 8.8 and *Call_Block* in section 8.7.

Error Message		Meaning and Remedy
Clause or "end loop" expected, at "*T*"	*Problem*	The assertion file should have an assertion clause here, or the **end loop** keywords, instead of the token *T*. See the nonterminals *Clause* in section 8.8 and *Loop_Block* in section 8.6.
Closing parenthesis after call expected, at "*T*"	*Problem*	A call description that is enclosed in parentheses should be followed by a ‘)’, instead of the token *T*. See the nonterminal *Other_Call* in section 8.6.
Closing parenthesis after loop expected, at "*T*"	*Problem*	A loop description that is enclosed in parentheses should be followed by a ‘)’, instead of the token *T*. See the nonterminal *Other_Loop* in section 8.6.
Closing parenthesis after parameters expected, at "*T*"	*Problem*	The assertions on subprogram parameters should be followed by a ‘)’, instead of the token *T*. See the nonterminal *Sub_Block* in section 8.5.
Context provides no base for code offset	*Problem*	A loop description uses the property “**executes offset** *code-offset*” but the subprogram that contains the loop is not specified so there is no base address for the offset. In other words, the description is in a *Loop_Block* that is a *Global_Bound*.
"cycles" expected, at "*T*"	*Problem*	This execution-time assertion should have the keyword **cycles**, instead of the token *T*, after the asserted execution time. It should be "**time** *N* **cycles**".
Dynamic call cannot have “call to” *or* Dynamic call cannot have “call_to”.	*Problem*	The assertion describes a call that is **dynamic** but is also a **call to** a named callee, which is an impossible combination.
“end call” expected, at "*T*"	*Problem*	The present *Call_Block* should end with **end call**, instead of the token *T*.
“end loop” expected, at "*T*"	*Problem*	The present *Loop_Block* should end with **end loop**, instead of the token *T*.
Execution time bounds for a loop are not allowed.	*Problem*	The assertion file tries to assert the **time** for a loop. This assertion is not supported for loops, only for sub-programs and calls.
Execution time bounds for the program are not allowed.	*Problem*	The assertion file tries to assert the **time** as a global fact. This assertion is not supported in the global context, only for subprograms and calls.
Integer value expected, at "*T*"	*Problem*	The assertion file should have an integer literal here, instead of the token *T*.
Integrated analysis cannot be negated.	*Problem*	The assertion tries to negate (disable) integrated analysis of a subprogram; this is not possible. See section 8.5.
Label not found: *L*	*Problem*	The assertion file names a label *L* but the target program's symbol table does not have a statement label named *L* in this scope.
	Reasons	The name *L* may be mistyped; if the default scope is used perhaps another scope should be named explicitly; or the target compiler may have mangled the names.
	Solution	Check for typos. Check the target program's symbol table using eg. *-trace symbols* or by dumping the file.

Error Message		Meaning and Remedy
"loop" after "end" expected, at "*T*"	*Problem*	The keyword **loop** should here follow the keyword **end**. See the nonterminal *Loop_Block* in section 8.6.
Loop description expected, at "*T*"	*Problem*	A loop description is expected here, instead of the token *T*. See the nonterminal *Loop_Description* in section 8.6.
Loop or call expected, at "*T*"	*Problem*	A loop-block or call-block is expected here, instead of the token *T*. See the nonterminals *Loop_Block* in section 8.6 and *Call_Block* in section 8.7.
Loop property expected, at "*T*"	*Problem*	The assertion file should here have a loop property, instead of the token *T*. See the nonterminals *Loop_Property* and *Loop_Properties* in section 8.6.
Mismatched subprogram identifier "*S*" after "end"	*Problem*	The subprogram identifier *Sub_Name* after the final **end** of a *Sub_Block* does not match the identifier given at the start of the block. This message is followed by an error message that shows the expected identifier.
"not", "in" or "labelled" expected, at "*T*"	*Problem*	The assertion file should here have one of the keywords **not**, **in**, or **labelled**, instead of the token *T*. See the nonterminal *Loop_Property* in section 8.7 and the nonterminal *Call_Property* in section 8.7.
Quoted character expected , at "*T*"	*Problem*	The assertion file should have a character in single quotes here ('c'), instead of the token *T*.
Quoted string expected , at "*T*"	*Problem*	The assertion file should have a string in double quotes here ("string"), instead of the token *T*.
Repetition bounds on subprograms not implemented	*Problem*	The assertion file contains an assertion on execution count (**repeats** *N* **times**) for a subprogram. This assertion is not allowed for subprograms, only for loops and calls.
Semicolon after callees expected, at "*T*"	*Problem*	The list of callee subprograms in a Callee_Bound (section 8.14) should be followed by a semicolon, instead of the token *T*.
Semicolon after "end call" expected, at "*T*"	*Problem*	The keywords **end call** should be followed by a semicolon, instead of the token *T*.
Semicolon after "end loop" expected, at "*T*"	*Problem*	The keywords **end loop** should be followed by a semicolon, instead of the token *T*.
Semicolon after "end subprogram" expected, at "*T*".	*Problem*	A subprogram block lacks the terminating semicolon. Instead, the next token is *T*.
Semicolon after subprogram option expected, at "*T*"	*Problem*	A subprogram option lacks the terminating semicolon. Instead, the next token is *T*.
Semicolon expected after clause, at "*T*"	*Problem*	An assertion clause lacks the terminating semicolon. Instead, the next token is *T*.
Semicolon expected after global bound, at "*T*"	*Problem*	An assertion (on the values of a variable or a property) in the global context lacks the terminating semicolon. Instead, the next token is *T*.
Subprogram address is invalid	*Problem*	An assertion tries to identy a subprogram by giving its (entry) address, but the address string is not in the proper form for this target processor. The invalid address string is shown in output field 4.

Error Message		Meaning and Remedy
	Solution	Refer to the Application Note for your target processor and write the entry address in the proper form.
Subprogram name "*N*" expected, at "*T*"	*Problem*	The subprogram name given at the end of a subprogram block does not match the name *N* given at the start of the block. Instead, the token at the end is *T*.
Subprogram name expected, at "*T*"	*Problem*	The assertion file should have a subprogram name (or address) here, in double quotes, instead of the token *T*.
Subprogram not found: *N*	*Problem*	The subprogram named *N* for this subprogram block (after the **subprogram** keyword) was not found in the target program.
	Reasons	Typing mistake in the name, or some name mangling by the compiler and linker.
	Solution	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 *-trace symbols* or dump the target program file.
Text at or after column *C* not understood: "*T*"	*Problem*	The text at or after column *C* on the current line in the assertion file is not a valid lexical "token" of the assertion language. The string *T* contains (part of) this text.
The integer literal "*n*" is not a valid number	*Problem*	The digit string *n* is not a valid number for some reason. Perhaps it has too many digits for the integer type that Bound-T uses for asserted numbers.
The “unused” property cannot be negated	*Problem*	A *Sub_Option* structure (section 8.5) uses the keywords **used** or **unused** and perhaps also the keyword **not** in a combination that says that the subprogram in question is “not unused”. This is considered an error because it is the default condition for any subprogram.
"times" expected, at "*T*"	*Problem*	This execution-count assertion should have the keyword **times**, instead of the token *T*, after the asserted execution time. It should be **repeats** *N* **times**.
Unrecognized property name: *P*	*Problem*	The assertion file names a target-specific "property" *P* but there is no such property for this target processor.
	Solution	Check the target-specific Application Note for the names of the properties for this processor.
Variable not found: *V*	*Problem*	The assertion file names a variable *V*, but the target program’s symbol table does not have a variable named *V* (in the implicit or explicit scope).
	Reasons	The name *V* may be mistyped; if the default scope is used perhaps another should be named explicitly; or the target compiler may have mangled the names.
	Solution	Check for typos. Check the target program’s symbol table using eg. *-trace symbols* or by dumping the file.
Variable name expected, at "*T*"	*Problem*	The assertion file should here have the name (or address) of a variable, instead of the token *T*.

9 TROUBLESHOOTING

This section explains how to understand and correct problems that may arise in using Bound-T, by listing all the warning and error messages that can be issued, what they mean, and what to do in each case.

If you cannot find a particular message here, please refer to the Application Notes for your target system and host platform; additional, specific messages may be listed there. Also, this section omits the messages related to the HRT analysis mode; see section 1.4. The error messages from the assertion parser, reporting errors in the assertion file, are listed in section 8.16.

9.1 Bound-T Warning Messages

Warning messages use the basic output format described in section 7.2, with the key field *Warning*. Fields 2 - 5 identify the context and location of the problem, and field 6 is the warning message, which may be followed by further fields for variable data.

The following table lists all Bound-T warning messages in alphabetical order. The target-specific Application Notes may list and explain additional target-specific warning messages. See section 1.4 regarding additional warning messages for HRT analysis mode.

As Bound-T evolves, the set and form of these messages may change, so this list may be out of date to some extent. However, we have tried to make the messages clear enough to be understood even without explanation. Feel free to ask us for an explanation of any Bound-T output that seems obscure.

Table 26: Warning messages

Warning Message		Meaning and Remedy
Assertion file already specified: *filename*	*Reasons*	The same assertion *filename* is specified in two or more *-assert* command-line options. The assertion file is only read and parsed once; repeated *-assert* options for the same file are skipped.
	Action	Correct the command-line options.
Assertion violated in step *S*: *T* := *V* /= *A*.	*Reasons*	During the constant-propagation analysis, Bound-T has found an instruction (in step number *S*) that assigns the value *V* to the variable (register or memory cell) *T*, but this variable is asserted to have the different value *A* throughout the current subprogram. This is a contradiction. Bound-T continues the analysis with the value V, overriding the assertion at this point.
	Action	Correct the assertion.
Callee never returns.	*Reasons*	Bound-T has discovered a call to a subprogram (the callee) that never returns to the caller.
	Action	This warning is given only if the command-line option *-warn return* is used. Omit this option to suppress this warning.

Warning Message		Meaning and Remedy
Cannot interpret constant as unsigned with *N* bits: *V*	*Reasons*	During the constant-propagation analysis of a bit-wise logical operation of width *N* bits, one operand has received a constant negative value *V* that is too negative to be considered an *N*-bit two's complement number. The analysis continues with the *N*-bit value "all ones".
	Action	Check how Bound-T decodes the target program at this point (use the option *-trace effect*). The warning may indicate that an instruction operand is decoded incorrectly.
Closing scope *A* but current scope is *B*	*Reasons*	Minor internal problem in Bound-T that is unlikely to influence the analysis.
	Action	Please report to Tidorum Ltd.
Conflicting "arithmetic" assertions.	*Reasons*	For the current subprogram there are both assertions that enable arithmetic analysis (**arithmetic**) and assertions that disable it (**not arithmetc**), which creates an ambiguity. For this subprogram, Bound-T will apply the command-line options that control arithmetic analysis.
	Action	Correct the assertion file(s).
Conflicting assertions on entry: *interval*	*Reasons*	Combining all facts and assertions that apply on entry to the current subprogram, but omitting the parameter bounds derived from the calling context or asserted for the call, the variable named in the *interval* has no possible values (the interval is empty). This is a contradiction. The *interval* has the form *min <= variable <= max* where min and max are constants such that *min > max*. The contradiction may be beetween different assertions on the same variable, or between an assertion on a variable and some target-specific implicit bounds on the variable.
	Action	Correct the assertion file(s).
Conflicting assertions or context on entry: *interval*	*Reasons*	Same as above ("conflicting assertions on entry") except that the conflicting facts and assertions include the parameter bounds derived from calling context or asserted for the call. If this warning is given for a particular variable, but the warning "conflicting assertions on entry" is not given for the same variable, the conflict depends on the context-specific parameter bounds.
	Action	For further explanation and possible actions see the warning "Conflicting assertions on entry".
Constant interpreted mod 2^*N*: *V*	*Reasons*	During the constant-propagation analysis of a bit-wise logical operation of width *N* bits, one operand has received a constant value *V* that is larger than the maximum unsigned *N*-bit value. The analysis continues with the value *V mod* 2^N.

Warning Message		Meaning and Remedy
		In a real execution, the computation of this operand involves overflow of some form, which implicitly applies the *mod* operation. Another reason may be an assertion that gives an *N*-bit register a value out of the *N*-bit range.
	Action	Check the assertions. Change the program so that overflow does not occur (but the analysis is correct here, even for overflow).
Constant out of range for *N*-bit bit-wise operation: *V*	*Reasons*	During the constant-propagation analysis of a bit-wise logical operation of width *N* bits, one operand has received a value *V* that is negative or too large for any bit-wise operation. The analysis continues but considers that this operand has an unknown value. Another reason may be an assertion that gives an *N*-bit register a value out of the *N*-bit range.
	Action	Check the assertions. Check how Bound-T decodes the target program at this point (use the option *-trace effect*). The warning may indicate that an instruction operand is decoded incorrectly.
Constant out of range for unsigned *N* bits: *V*	*Reasons*	During the constant-propagation analysis of a bit-wise logical operation of width *N* bits, one operand has received a value *V* that is too large (negative or positive) to be used as an unsigned operand in any bit-wise operation. The analysis continues but considers that this operand has an unknown value.
	Action	Check how Bound-T decodes the target program at this point (use the option *-trace effect*). The warning may indicate that an instruction operand is decoded incorrectly.
Constant propagation stops at iteration limit	*Reasons*	The constant-propagation analysis may resolve the address used in a dynamic memory access and thus change it to an access to a statically known memory cell. The constant-propagation analysis is then repeated for the extended computation model. This warning reports that the maximum number of such iterations was reached. Bound-T continues with other analyses, even if another round of constant propagation could improve the computation model.
	Action	If more iterations are desired, use the option *-const_iter* to increase the limit.
Constant *V* exceeds calculator range, considered unknown.	*Reasons*	While building the arithmetic model of an instruction, Bound-T found a constant value *V* with an absolute value that is larger than the maximum set by the option *-calc_max*. The constant is considered to have an unknown value. The value is probably an address constant or a bit-mask and may or may not have an effect on the arithmetic analysis of loop bounds.

Warning Message		Meaning and Remedy
	Action	This warning is given only if the command-line option *-warn literal* is used. To suppress the warning but still exclude the value from the arithmetic analysis, omit this option. The value of the limit can be set by the command-line option *-calc_max*. To include the value in the arithmetic analysis, increase the limit using this option. However, this increases the risk that the Omega calculator runs into overflow, which makes the arithmetic analysis fail.
Duplicated symbol : *dup-conn* : *first-conn*	*Reasons*	The symbol-table in the target executable file contains two (or more) occurrences of the same symbol (identifier) which are not distinguished by scope or other context. Moreover, these two occurrences connect the same symbol to different machine-level values (*eg.* different addresses) so the symbol is ambiguous. The part *first-conn* describes the first occurrence of the symbol and the part *dup-conn* describes the current (second, third, ...) occurrence of the symbol. The *first-conn* and *dup-conn* parts each consist of three fields separated by colons: the kind of the symbol, the symbol (identifier) itself, fully qualified by scope; and the machine-level value connected to the symbol. The symbol-kind is one of “Subprogram”, “Label”, “Variable” or “Source line”. This warning often occurs with compiler-generated symbols for subprogram prelude and postlude code, return point addresses, and so on. The target compiler/linker created the file with this content, or Bound-T did not recognize the distinguishing features of these insignificant symbols. This warning is emitted only if the command-line option *-warn symbol* is enabled (it may be enabled by default). The option *-warn no_symbol* suppresses this warning.
	Action	The first occurrence of the symbol is accessible (to assertions, for example), the others are not (or must be referred to via their addresses). No action by the user can correct this problem in general. For some target processors command-line options may control which symbol-tables that Bound-T scans for symbols; limiting the selection may remove some of these warnings.
Duplicated symbol and value : *dup-conn*	*Reasons*	The symbol-table in the target executable file contains two (or more) occurrences of the same symbol (identifier) which are not distinguished by scope or other context. However, these two occurrences connect the symbol to the same machine-level values (*eg.* addresses) so this duplication does not make the symbol ambiguous. The part *dup-conn* describes the two occurrences which are equal. See the warning “Duplicated symbol”, above, for a description of the form of *dup-conn*.

Warning Message		Meaning and Remedy
		This warning is emitted only if the command-line option *-warn symbol* is enabled (it may be enabled by default). The option *-warn no_symbol* suppresses this warning.
	Action	No action is needed.
Dynamic call.	*Reasons*	The call instruction under analysis defines the address of the callee by a dynamic computation; for example, a register-indirect call. Bound-T will try various forms of analysis to find the possible callee addresses and include those subprograms in the call-graph of the program under analysis.
		This warning is emitted only if the command-line option *-warn flow* is enabled (it may be enabled by default). The option *-warn no_flow* suppresses this warning.
	Action	Check that Bound-T has constructed a correct call-graph for this program. The analysis of dynamic calls may be imprecise. You can use a **dynamic call** assertion to list the possible callees.
Dynamic control flow.	*Reasons*	The instruction under analysis defines the address of the next instruction by a dynamic computation; for example, a register-indirect jump. Bound-T will try various forms of analysis to find the possible addresses and include them in the flow-graph of the subprogram under analysis.
		This warning is emitted only if the command-line option *-warn flow* is enabled (it may be enabled by default). The option *-warn no_flow* suppresses this warning.
	Action	Check that Bound-T has constructed a correct flow-graph for this subprogram. The analysis of dynamic control flow may be imprecise.
Empty assertion file.	*Reasons*	The assertion file that was given as an option to Bound-T turned out to be empty (of assertions; it may contain comments and blank lines).
	Action	Correct the file or the option.
Eternal loop (no exit edges).	*Reasons*	The subprogram under analysis contains a loop that has no exit (not even a conditional exit), so it must be eternal.
	Action	Modify the program or assert how many iterations of this loop should be included in the WCET. The warning will appear even if the loop is asserted unless suppressed with the option *-warn no_eternal*.
Eternal loop not included in WCET.	*Reasons*	This unbounded eternal loop was excluded from the possible execution paths.
		This warning should never appear because Bound-T requires all eternal loops to be bounded (by assertions) before it finds the worst-case paths.
	Action	Take this into account when using the WCET value.

Warning Message		Meaning and Remedy
Ignored multi-location invariance assertion on *V*	*Reasons*	An invariance assertion applies to a variable *V* for which the code uses different locations (memory cells, registers) at different points of the current subprogram. Bound-T does not support invariance assertions on such variables.
	Action	There is no sure work-around, but different optimization options for the target compiler may help.
Infeasible edge *e* splits node *n*	*Reasons*	Bound-T has examined the logical condition of the control-flow edge (number *e*) between two consecutive instructions in the same basic block (node number *n*) and found the condition to be false in every execution for the current context and assertions. This means that the edge is infeasible (cannot be executed). This is strange since there is no *alternative* edge, as the edge is *within* a basic block. The whole node that contains the infeasible edge becomes infeasible, too, and is excluded from the WCET.
	Action	Check the conditions in this region of the code and verify that the node is indeed infeasible.
Loop body executes once (asserted to repeat zero times)	*Reasons*	This loop consists of a single basic block: the loop head block. The assertion that the loop repeats zero times normally means that the loop head can execute once and the rest of the loop zero times. Here the loop consists of the head alone, so all of the loop executes once.
	Action	Check that the zero-repeats assertion is correct for this loop. If the intent was to say that the loop does not execute at all, find some way to assert that.
Negative *safety* stack height ($stack = S$) changed to zero.	*Reasons*	The stack-usage analysis has found a negative value *S* for the maximum height of the named *stack* within this subprogram, but will instead use and report a zero stack height. The message also indicates the *safety* of the value *S*. See section 3.11.
	Action	This should never happen. Please inform Tidorum Ltd.
Negative *safety* stack usage ($stack = S$) changed to zero.	*Reasons*	The stack-usage analysis has found a negative value *S* for the overall usage of the named *stack* for this subprogram, but will instead use and report a zero stack usage. The message also indicates the *safety* of the value *S*. See section 3.11.
	Action	This should never happen. Please inform Tidorum Ltd.
No valid assertions found in this file.	*Reasons*	The assertion file that was given as an *-assert* option to Bound-T turned out to be empty (of valid assertions; it may contain comments and blank lines and malformed assertions).
	Action	Correct the file or the option.
Non-returning subprogram	*Reasons*	This subprogram seems to have no feasible (reachable) return points in this context; it cannot return to its caller. One possibility is that the subprogram ends in an eternal loop.

Warning Message		Meaning and Remedy
	Action	Check the assertions, if any, to verify that no return is expected.
No relevant arithmetic to be analysed.	*Reasons*	The current subprogram contains some dynamic and as yet unbounded parts, such as loops or calls to subprograms that seem to need context-specific loop bounds, but the computations in the current subprogram seem irrelevant to these unbounded parts as they do not assign values to any variables on which the unbounded parts depend. Therefore Bound-T omits the arithmetic analysis of the current subprogram (in this context) as useless.
	Action	None. If the current subprogram is not a root subprogram, and if the maximum parameter-depth (option *-max_par_depth*) is not exceeded, Bound-T will automatically try to find relevant computational context from higher levels in the call-tree.
Null bounds on callee *which is immediately followed by* Null bounds on callee within these bounds	*Reasons*	While analysing a subprogram that calls other subprograms Bound-T unexpectedly found no analysis results (execution bounds) on one of the callees. Fields 3 to 5 of the first warning line identify the call. Fields 3 to 5 of the second warning line identify the caller.
	Action	This warning should normally never appear because Bound-T normally analyses callees before their callers. Please inform Tidorum.
Other address connection not used.	*Reasons*	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. This is generally harmless since the analysis uses the address rather than the symbolic name. The symbolic name appears only in the outputs.
	Action	Change the assertion to use the desired symbolic name.
Other identifier connection not used.	*Reasons*	An assertion uses a symbolic (source-code) name for a subprogram or statement-label, but this name is connected to more than one machine-code location (multiply defined). This message shows a connection that Bound-T did not use.
	Action	Add scope context to the name in the assertion, to make the name unambiguous. See section 5.12.
stack : Stack usage bound is unsafe (too small)	*Reasons*	The stack usage analysis could not find a true upper bound on the usage of the named *stack* in the current subprogram and current context, but it has produced an unsafe "lower bound" on the upper bound.
	Action	Inspect the subprogram code to understand why Bound-T cannot bound the stack usage; then modify the code or add assertions to support the analysis. See section 3.11.

Warning Message		Meaning and Remedy
Resolved callee never returns.	*Reasons*	While analysing a dynamic call, Bound-T has discovered one possible callee that never returns to the caller.
	Action	This warning is given only if the command-line option *-warn return* is used. Omit this option to suppress this warning.
Return point is dynamically computed.	*Reasons*	The return point (return address) of this call is computed dynamically, not set statically. Bound-T will try to analyse the computation to find the return point.
	Action	If Bound-T is unable to find the return point, as shown by an error message about unresolved dynamic flow, change the program to use a static return address or a simpler computation of the return point. If Bound-T is able to find the return point, check that the return point is correct (for example, is not affected by unresolved dynamic data references). This warning is given only if the command-line option *-warn computed_return* is chosen. Omit this option to suppress the warning.
Shallow scope for line-number *N*: *scope*	*Reasons*	The debugging information in the executable file connects source-line number *N* to a certain machine address but the *scope* given for this line-number is incomplete. The scope is expected to contain two levels, the source-code file and the subprogram that contain this line, but fewer levels are given for line *N*.
	Action	This warning is currently disabled. If it occurs, please inform Tidorum.
Subprogram not found in program	*Reasons*	In the assertion file, the subprogram named for this subprogram block (after the **subprogram** keyword) was not found in the target program. The name may be mistyped, or the name may have been “mangled” by the compiler and linker.
	Action	Correct the assertion file to use the subprogram name as it exists in the target program executable (the link-name). To find the possibly mangled name, use *-trace symbols* or dump the target program file.
Tail call to callee that never returns.	*Reasons*	This call appears to be a tail call, that is, it creates a state in which the callee will return to the same place to which this subprogram would return. However, the callee seems not to return at all.
	Action	Probably no action is needed. When the compiler optimised the call to a tail call it was probably not aware that the callee does not return at all.
This address connection used.	*Reasons*	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.

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.
	Action	Change the assertion to use the desired symbolic name.
This identifier connection used.	*Reasons*	An assertion uses a symbolic (source-code) name for a subprogram or statement-label, but this name is connected to more than one machine-code location (multiply defined). This message shows the connection that Bound-T uses.
	Action	Add scope context to the name in the assertion, to make the name unambiguous. See section 5.12.
Time is not bounded for *A=>B*	*Reasons*	While annotating the control-flow graph of the subprogram *A* with execution times, Bound-T found that no WCET is available for the call from *A* to the subprogram *B*.
	Action	This should never happen. Please inform Tidorum Ltd.
Unbounded residual pool for value list	*Reasons*	While Bound-T was listing (enumerating) the possible values of an address expression (usually the addresses of the branches of a switch/case control structure), it was unable to find the remaining values in the list.
	Action	Check that Bound-T has located all the branches of the switch/case structure at this point. If not, try to assert the possible values of the switch/case index.
Undefined bounds accessed for call-bounds.	*Reasons*	While building the execution bounds for a subprogram from those of its callees, Bound-T noticed that no execution bounds exist for the subprogram itself.
	Action	This should never happen. Please inform Tidorum Ltd.
Unreachable call.	*Reasons*	This call seems infeasible (unreachable) because the arithmetic analysis of the parameter values for the calling protocol or the callee signals a contradiction (impossible constraints). Thus, Bound-T excludes the path(s) with this call from the WCET. The contradiction can be intrinsic in the target program (for example, an "**if false then**" statement), or it can be due to the calling context (for example, an "**if** *B* **then**" statement where the parameter *B* is false in the current context), or it can be due to an assertion (for example, an "**if** $N > 5$ **then**" statement together with the assertion $N < 2$). See the discussion of "contradictory value bounds" in section 8.15.
	Action	Check the conditions under which the call is executed. Check that the assertions are valid.
Unreachable eternal loop (asserted to repeat zero times).	*Reasons*	This eternal loop is asserted to be repeated zero times, which Bound-T takes to mean that execution never reaches this loop.

Warning Message		Meaning and Remedy
	Action	Check that the zero-repeats assertion is valid, that is, that all paths to this loop really should be excluded from the worst-case analysis. Otherwise, change the assertion to state that the loop repeats once, or as many times as you like.
Unreachable exit-at-end loop (asserted to repeat zero times).	*Reasons*	This loop is asserted to be repeated zero times, but it is only exited at the end of the loop body. Thus, if the loop is reached at all, the loop body must be exected once. Bound-T resolves this conflict by considering the whole loop unreachable.
	Action	Check that the zero-repeats assertion is valid, that is, that all paths to this loop really should be excluded from the worst-case analysis. Otherwise, change the assertion to state that the loop repeats once, or as many times as you like.
Unreachable flow to instruction at *A*	*Reasons*	Bound-T has examined the logical condition of the control-flow edge from the current instruction to the instruction at address *A* and found the condition to be false in every execution for the current context and assertions. This means that the edge is infeasible (cannot be executed). The most common reason is a **for**-loop where the number of repetitions depends on a parameter of the current subprogram and the compiler generates a separate check and branch for the case where the loop should not be repeated at all. This branch is thus infeasible when the subprogram is called with parameters that do cause loop repetitions. This warning is given only if the option *-warn reach* is on, which is the default.
	Action	Check that this result is correct.
Unreachable instruction.	*Reasons*	This instruction (flow-graph step) seems infeasible (unreachable) because the arithmetic analysis of the data values reaching this instruction signals a contradiction (impossible constraints). Thus, Bound-T excludes the path(s) with this instruction from the WCET.
	Action	See the warning “Unreachable call” for further discussion of possible reasons and corrective actions.
Unreachable loop.	*Reasons*	This loop seems infeasible (unreachable) because the arithmetic analysis of the loop counters signals a contradiction (impossible constraints). Thus, Bound-T excludes the path(s) with this loop from the WCET. This warning is given only if the option *-warn reach* is on, which is the default.
	Action	Check that this result is correct.

Warning Message		Meaning and Remedy
Unreachable loop (asserted to repeat zero times).	*Reasons*	This loop is a "bottom-test" loop where the loop body must be executed before reaching the loop termination test. This conflicts with the assertion that the loop repeats zero times. Bound-T resolves this conflict by considering the whole loop unreachable.
	Action	Check that the zero-repeats assertion is valid, that is, that the loop body really should be excluded from the worst-case analysis. Otherwise, change the assertion to state that the loop repeats once.
Unreachable loop body (asserted to repeat zero times).	*Reasons*	This loop contains a loop head and some other basic blocks that form the (rest of) the loop body. Thus, the assertion that the loop repeats zero times implies that these other blocks are unreachable (never executed). This warning is given only if the option *-warn reach* is on, which is the default.
	Action	Check that the zero-repeats assertion is valid and that this effect is intended.
Unrepeatable loop.	*Reasons*	This loop seems to be unrepeatable because the arithmetic analysis of the data on the repeat edges signals a contradiction (impossible constraints). Thus, Bound-T considers the repeat edges infeasible, which effectively means a loop-bound of zero repetitions. Still, the loop body, or parts of it, may be executed once for every time the loop is reached. This warning is given only if the option *-warn reach* is on, which is the default.
	Action	Check that this result is correct.
Unresolved dynamic memory read. *or* Unresolved dynamic memory write. *or* Unresolved dynamic memory read in condition.	*Reasons*	The actual memory address or addresses in a dynamic (indexed, pointer-based) memory-reading or memory-writing instruction could not be determined. If the instruction in question alters a variable that is involved in loop-counting, the loop-bounds derived by Bound-T may be wrong. The most common such instructions are reading or writing array elements or reading or writing a call-by-reference parameter. This warning is emitted only if the command-line option *-warn access* is used.
	Action	Inspect the target program to verify that the instruction in question does not affect loop-counting. To suppress these warnings when they are irrelevant, omit the option *-warn access*.
Unsigned operand too large, considered unknown: *V*	*Reasons*	During the constant-propagation analysis of a bit-wise logical operation, one operand has received a value *V* that is out of range for the target processor. The analysis continues but considers that this operand has an unknown value.

Warning Message		Meaning and Remedy
	Action	Check how Bound-T decodes the target program at this point (use the option *-trace effect*). The warning may indicate that an instruction operand is decoded incorrectly.

9.2 Bound-T Error Messages

Error messages use the basic output format described in section 7.2, with the key field *Error*. Fields 2 - 5 identify the context and location of the problem, and field 6 is the error message, which may be followed by further fields for variable data.

The following table lists all Bound-T error messages in alphabetical order except for:

- error messages from the assertion parser; please see section 8.16;
- target-specific errors; please refer to the Application Note for the target;
- errors specific to HRT analysis; please see section 1.4.

For each error message, the table explains the problem in more detail, makes a guess at the possible or likely reasons for the problem, and proposes some solutions. Of course, changing the target program is nearly always a possible solution, but this is not listed in the table unless it is the only solution.

As Bound-T evolves, the set and form of these messages may change, so this list may be out of date to some extent. However, we have tried to make the messages clear enough to be understood even without explanation. Feel free to ask us for an explanation of any Bound-T output that seems obscure.

Table 27: Error messages

Error Message		Meaning and Remedy
Argument is not a duration: *A*	*Problem*	A non-numeric command-line argument *A* was given to Bound-T where a numeric one was expected. The argument is expected to set a duration so it may include a decimal point and a decimal part.
	Reasons	Mistake on the command line.
	Solution	Restart with correct form of arguments. See section 6.4.
Argument is not a number: *A*	*Problem*	A non-numeric command-line argument *A* was given to Bound-T where a numeric one was expected. The argument is expected to be an integer number without a decimal part.
	Reasons	Mistake on the command line.
	Solution	Restart with correct form of arguments. See section 6.4.
At most *N* parameters allowed: *P*	*Problem*	The current patch file contains a line that has more than the maximum of *N* patch parameters, so the parameter *P* is ignored.
	Reasons	Perhaps the line is mistyped, with some extra blanks that split up parameters.
	Solution	Correct the patch file.

Error Message		Meaning and Remedy
At most *N* patch files allowed: *name*	*Problem*	The command-line contains more than the maximum of *N* *-patch* options. The patch file with this *name* is thus ignored.
	Solution	Combine the contents of some patch files to bring the total number of patch files to at most *N*.
Bit-wise result too large: *V*: *E*	*Problem*	The result *V* of applying constant propagation to the expression *E*, which includes a bit-wise Boolean operator, exceeds the range of arithmetic values in this target processor.
	Reasons	Error in Bound-T.
	Solution	This should not happen. Please report it to Tidorum Ltd.
Call matches too few entities	*Problem*	The assertion file contains an assertion on a call where the call description matches a smaller number of actual calls than expected. The matching calls (if any) are shown by appended error lines of the form "match *n*: *caller@locus=>callee*".
	Reasons	The call description is too specific, or the target subprogram contains fewer such calls than expected. Perhaps the compiler has in-lined a call.
	Solution	Improve call description in assertion file.
Call matches too many entities	*Problem*	The assertion file contains an assertion on a call where the call description matches a greater number of actual calls than expected. The matching calls are shown by appended error lines of the form "match *n*: *caller@locus=>callee*".
	Reasons	The call description is too general, or the target subprogram contains more such calls than expected. Perhaps the compiler has duplicated some code for optimization reasons.
	Solution	Improve call description in assertion file.
Callee stack-usage for *stack* not safely bounded.	*Problem*	The search for the worst-case stack path found a call to a subprogram with an unsafe (lower) bound on its usage of the named *stack*.
	Reasons	The subprogram uses the stack in a complex way that Bound-T cannot analyse exactly.
	Solution	Inspect the subprogram code to understand why Bound-T cannot bound the stack usage; then modify the code or add assertions to support the analysis. See section 3.11.
Callee stack-usage for *stack* unknown, using unsafe lower bound.	*Problem*	The search for the worst-case stack path found a call to a subprogram with an unknown usage for the named *stack*. The stack usage analysis will instead use the initial value (on entry) of the stack height as an unsafe lower bound on the stack usage in this subprogram.

Error Message		Meaning and Remedy
	Reasons	The subprogram uses the stack in a complex way that Bound-T cannot analyse at all. A special case where this error arises is when the subprogram is exluded from the analysis because an execution time is asserted for it.
	Solution	Inspect the subprogram code to understand why Bound-T cannot bound the stack usage; then modify the code or add assertions to support the analysis. See section 3.11.
Cannot create DOT file named "*name*".	*Problem*	Bound-T could not create a file called *name* to hold the DOT drawings requested by a command-line option *-dot name* or *-dot_dir dirname*.
	Reasons	Perhaps the folder or directory is write-protected, or a write-protected file by this *name* already exists, or the specified *name* is not a legal file-name on this host system.
	Solution	Change the *name* or modify file/folder permissions.
Cannot decode subprogram; using null stub.	*Problem*	The (target-specific) instruction-decoder module failed to decode the first instruction in the subprogram, leaving the control-flow graph empty.
	Reasons	The decoder module should have emitted an error message that explains the reason. Perhaps the code for this subprogram is not present in the executable file.
	Solution	Depends on the target-specific reason for the error.
Cannot integrate dynamic call to *S*. Calling by reference.	*Problem*	While resolving a dynamic call or obeying an assertion that lists the possible callees of a dynamic call, Bound-T found that one possible callee is the subprogram *S* which, however, is defined as a subprogram to be "integrated" with its callees and not analysed on its own. Integrated analysis is not possible for a dynamic call so this call of *S* will be analysed as a normal, non-integrated or "reference" call. Subprograms marked for integrated analysis usually violate normal calling conventions which means that the analysis of *S* through this call is likely to fail.
	Reasons	A mistake in the assertion files (perhaps *S* should not be integrated, or should not be listed as a possible callee) or an error in Bound-T's analysis of the possible callees.
	Solution	Correct the assertions, or if Bound-T's analysis is in error, work around it by asserting the true list of callees for this call.
Computation model did not converge in *n* iterations and may be unsafe	*Problem*	After *n* iterations of various analyses and consequent updates of the arithmetic computation model of this subprogram, the model is still not stable; more iterations might be needed.
	Reasons	The subprogram probably contains many dynamic data references (pointers and pointers to pointers, or indexed array references) that are simple enough for Bound-T to resolve, but nested in such a way that successive analyses are required to resolve them all.

Error Message		Meaning and Remedy
	Solution	Try to increase the iteration limit using the command-line option *-model_iter number*.
Could not be fully bounded.	*Problem*	This root subprogram could not be fully bounded, because Bound-T could not bound some dynami behaviour in it or in its callees. Dynamic behaviour includes loops (for WCET analysis) and dynamic stack usage (for sstack analysis).
	Reasons	The loop(s) or stack usage are too complex to be automatically bounded, and were not bounded by assertions.
	Solution	Inspect the rest of the output to find out which dynamic behaviours are unbounded. Bound them with assertions or modify the program to make them boundable automatically.
Could not open the patch file "*name*".	*Problem*	Bound-T could not open the patch file with the given *name* as specified by the command-line option *-patch name*.
	Reasons	The file name may be wrong (file does not exist) or the user may not have read access to the file.
	Solution	Correct the file name on the command-line, or correct the permissions of the file.
Dynamic flow needs arithmetic analysis.	*Problem*	This subprogram contains dynamic jumps for which arithmetic analysis is required, but arithmetic analysis is disabled. Bound-T will not be able to bound the execution of this subprogram with these options and assertions.
	Reasons	The command-line contains the option *-no_arithmetic* which disables arithmetic analysis generally, or the assertion file uses the option **no arithmetic** to disable it specifically for this subprogram.
	Solution	Recode the subprogram to avoid dynamic jumps, or change the command-line options or the assertion options to allow arithmetic analysis of this subprogram.
Dynamism bounding did not converge in *N* iterations	*Problem*	The flow-graph still contains unresolved dynamic branches, and the iterative resolution process has used the maximum allowed number *N* of iterations of data-flow analysis alternated with resolving dynamic accesses and extending the flow-graph.
	Reasons	The subprograms under analysis may contain sequences of dynamic branches such that resolving the first branch leads to the discovery of a next branch, and so on.
	Solution	Increase the maximum number of iterations with the option *-dynamic_limit*.
Full time bounding needs arithmetic analysis.	*Problem*	This subprogram contains some features (loops or calls to subprograms with loops) for which the execution time can be bounded only by arithmetic analysis, but arithmetic analysis is disabled. Bound-T will not be able to bound the execution time of this subprogram with these options and assertions.

Error Message		Meaning and Remedy
	Reasons	The command-line contains the option *-no_arithmetic* which disables arithmetic analysis generally, or the assertion file uses the option **no arithmetic** to disable it specifically for this subprogram.
	Solution	Add assertions to bound the loops, or change the command-line options or the assertion options to allow arithmetic analysis of this subprogram. It may also be necessary to enable arithmetic analysis for some of the callees.
Infeasible execution constraints	*Problem*	Bound-T cannot find any execution path in this subprogram that obeys all the computed and asserted constraints. Therefore no WCET bound is found.
	Reasons	The assertions may be contradictory, in particular assertions on the number of loop repetitions or call executions can conflict.
	Solution	Check the assertions that apply to this subprogram, including relevant global assertions.
Irreducibility prevents arithmetic analysis	*Problem*	The subprogram cannot be analysed arithmetically because the control-flow graph is not “reducible”, that is, it cannot be divided into properly nested loops.
	Reasons	See the error message “Irreducible flow-graph”.
	Solution	Ditto.
Irreducible flow-graph	*Problem*	The control-flow graph of the subprogram under analysis is not "reducible", that is, it cannot be divided into properly nested loops where each loop has a single point of entry (the loop head). The loops intersect one another in some way, or there are jumps into loops that by-pass the loop head. Bound-T can only analyse execution time and arithmetic for reducible flow-graphs. Stack usage analysis is possible even for an irreducible subprogram, providing that arithmetic analysis is not needed. In this case, this error should be considered a warning only.
	Reasons	The subprogram is coded in this way, either by the programmer directly or by the optimising code generator in the compiler. The usual reason is that there is a jump into the body of a loop from outside the loop.
	Solution	Change the subprogram’s source code if the problem is there, or change the compiler options (reduce optimization level). If the subprogram calls other routines that do not return (for example, routines for handling fatal errors) it may help to assert these routines as **no return**.
Local stack height for *stack* at call is not bounded.	*Problem*	The arithmetic analysis found no upper bound on the space allocated locally by the current subprogram in this *stack* at the point of this call. Thus, even if the stack-usage of the callee is known, the total *stack* usage cannot be computed for this call.

Error Message		Meaning and Remedy
	Reasons	The amount of space allocated on the stack (the change in the stack pointer) is computd in some way that Bound-T cannot analyse.
	Solution	Change the target program to use an analysable amount of stack space.
Local stack height for *stack* is not bounded.	*Problem*	The current instruction modifies the pointer of this *stack*, allocating or releasing *stack* space locally for the current subprogram, but the arithmetic analysis found no upper bound on the allocated space.
	Reasons	The amount of space allocated on the stack (the change in the stack pointer) is computd in some way that Bound-T cannot analyse.
	Solution	Change the target program to use an analysable amount of stack space.
Loop matches too few entities	*Problem*	The assertion file contains an assertion on a loop where the loop description matches a smaller number of actual loops than expected. The matching loops (if any) are shown by appended error lines of the form "match *n*: *locus*".
	Reasons	The loop description is too specific, or the target subprogram contains fewer such loops than expected. Perhaps the compiler has in-lined (unrolled) some loop.
	Solution	Improve loop description in assertion file.
Loop matches too many entities	*Problem*	The assertion file contains an assertion on a loop where the loop description matches a greater number of actual loops than expected. The matching loops are shown by appended error lines of the form "match *n*: *locus*".
	Reasons	The loop description is too general, or the target subprogram contains more such loops than expected. Perhaps the compiler has created some loops for its own purposes such as copying data in an assignment statement.
	Solution	Improve loop description in assertion file.
Match *n*: *caller@locus=>callee*	*Problem*	This message follows an error message of the type "call matches too few/many entities" and shows the locus (code address and/or source-line number) in the target program of one of the calls matches the call description in the assertion file. The matches are numbered; this is match number *n*. See the error messages "call matches too few/many entities" for the possible reasons and solutions.
Match *n*: *locus*	*Problem*	This message follows an error message of the type "loop matches too few/many entities" and shows the locus (code addresses and/or source-line numbers) in the target program of one of the loops that matches the loop description in the assertion file. The matches are numbered; this is match number *n*.

Error Message		Meaning and Remedy
	Reasons *Solution*	See the error messages "loop matches too few/many entities" for the possible reasons and solutions.
Maximum analysis time exceeded.	*Problem*	The analysis has taken longer than the limit specified with the option *-max_anatime* so the analysis is aborted.
	Reasons	The requested analysis needs more computation than is possible in the allowed analysis duration. The most common time-consumer is the arithmetic analysis for loop-bounds.
	Solution	Increase the allowed duration or reduce the analysis tasks, for example by using assertions instead of arithmetic analysis for loop-bounds.
Maximum call-dependent analysis depth reached.	*Problem*	The context (call-path) under analysis is deeper (has more call levels) than the maximum set by the option *-max_par_depth*. The current subprogram (the final callee in this context) will not be analysed further. It will remain "not fully bounded".
	Reasons	The subprogram's loops have a form that Bound-T cannot analyse (they are not counter-based or have counters that use computations that Bound-T cannot analyse); or the bounds for the loop counters are passed from a still higher-level caller (the context is not deep enough); or in a way that Bound-T cannot track (as elements of arrays, for example).
	Solution	Assert bounds on the loops or change the target program to use more locally defined loop bounds or to pass loop bounds in a way that Bound-T can track. It is also possible to increase *-max_par_depth* but this probably increases analysis time considerably so do it only after checking that *-max_par_depth* really is the obstacle.
No feasible execution path.	*Problem*	There is no feasible (reachable) path through the control flow-graph of this subprogram, in this context, so no WCET bound can be computed for it.
	Reasons	So many parts of the flow-graph have been marked as unreachable, due to assertions or analysis, that the pruning process (see section 6.5) has made the entry node unreachable.
	Solution	Check the assertions to see if this result is expected. If so, remove the subprogram (or this call of the subprogram) from the analysis by asserting that it is never called.
Option conflict: HRT analysis requires time bounds	*Problem*	The command-line options request HRT analysis (*-hrt*) but disable time analysis (*-no_time*). This is contradictory.
	Reasons	The only purpose of the HRT analysis is to provide execution-time bounds for an HRT model. Thus an HRT analysis with *-no_time* is useless.
	Solution	Correct the command line.

Error Message		Meaning and Remedy
Patch address invalid: *A*	*Problem*	The current patch file has a non-comment line that starts with the token *A* which is not a valid patch address (according to the target-specific syntax).
	Reasons	Error in the patch file.
	Solution	Correct the patch file.
Patch data missing.	*Problem*	The current patch file has a line with a valid patch address but no data (nothing after the address).
	Reasons	Error in the patch file.
	Solution	Correct the patch file.
Patch line too long (over *N* characters).	*Problem*	The current patch file has a line that contains more than the maximum of *N* characters.
	Reasons	Error in the patch file.
	Solution	Correct the patch file by shortening the line, perhaps by removing leading or trailing whitesparce or other redundant whitespace.
Patch parameter invalid: *P*	*Problem*	The current patch file has a parameter P that is neither the name of a subprogram or a label nor a valid code address (in the target-specific format).
	Reasons	Error in the patch file, or perhaps name-mangling by the compiler or linker.
	Solution	Correct the patch file.
Recursion detected	*Problem*	The subprogram is part of a recursive cycle of calls, either directly (*Sub* calls *Sub*) or indirectly (*Sub1* calls *Sub2*, *Sub2* calls *Sub1*, etc.) This error message is followed by two or more *Recursion_Cycle* output lines that describe one recursion cycle in the program (there may be more).
	Reasons	The target program was written in that way.
	Solution	Modify the target program, removing the recursion.
	Work-around	Give an assertion on the WCET of some subprogram in the cycle. This will keep Bound-T from analysing that subprogram at all, and will thus hide the recursion. You must then manually combine the computed WCET values with your understanding of how the recursion works, to get an upper bound on the execution time that includes the recursive calls. See section 5.18.
Recursive integrated call to *S* at *A* changed to normal (recursive) call	*Problem*	This call to subprogram *S*, with entry address *A*, would create a recursive “integration” of *S* (as defined in section 8.5) and thus the analysis would not terminate. To ensure termination Bound-T analyses the present call of *S* as normal (not integrated) call. However, the call-graph is still recursive so the analysis will fail in a later phase.
	Reasons	Subprogram *S* is defined to be integrated but is part of a recursive cycle of subprograms.

Error Message		Meaning and Remedy
	Solution	Change the target program to remove the recursion or change the analysis approach to break the recursion, for example as suggested in section 5.18.
Response line *n*: *line*	*Problem*	This message follows an error message of the form "calculator did not give the expected echo/empty line" and displays the (unexpected) response *line* from the Omega Calculator, as well as the sequential number *n* of this line.
	Reasons and Solution	See the error messages referred to in the *Problem* row.
Root subprogram cannot be “unused”.	*Problem*	An assertion defines this root subprogram as an “unused” subprogram (see section 8.5). This is a contradiction because it prevents the analysis of the subprogram.
	Reasons	Perhaps a mistake in the assertion file, or a mistake in the command line (naming wrong subprogram as root).
	Solution	Correct the assertion file or the command line.
Root subprogram name is ambiguous	*Problem*	The name (symbol, identifier) given on the command line matches more than one actual subprogram. The name is thus ambiguous.
	Reasons	The program contains several subprograms with similar names although in different scopes. The scope part of the name on the command line is not complete enough to separate between these subprograms.
	Solution	Add scope levels to the name on the command line. For example, if the program contains two modules, *Err* and *Pack*, that contain the same subprogram *Foo*, write either “*Err*|*Foo*” or “*Pack*|*Foo*” on the command line to say which of these two functions is to be analysed.
Root subprogram not found or address in wrong form.	*Problem*	A root subprogram named on the command line was not found in the target program, nor could the given name string be understood as a valid code address (entry address for the root subprogram).
	Reasons	Error in the name given as command argument; or an entry address in incorrect syntax; or some name mangling by the compiler and linker; or some other error in command-line syntax that makes Bound-T try to interpret this argument as the name of a root subprogram although this was not meant.
	Solution	Correct the command to use the subprogram name as in the target program executable. See Chapter 6. If the root subprogram was meant to be identified by its entry address, refer to the Application Note for this target for the correct syntax of code addresses.
Stack usage needs arithmetic analysis.	*Problem*	This subprogram uses the stack in such a way that arithmetic analysis is required to bound the stack usage, but arithmetic analysis is disabled. Bound-T will not be able to bound the (local) stack usage of this subprogram with these options and assertions.

Error Message		Meaning and Remedy
	Reasons	The command-line contains the option *-no_arithmetic* which disables arithmetic analysis generally, or the assertion file uses the option **no arithmetic** to disable it specifically for this subprogram.
	Solution	Recode the subprogram to avoid dynamic stack usage, or change the command-line options or the assertion options to allow arithmetic analysis of this subprogram.
Stack usage for calls needs arithmetic analysis.	*Problem*	This subprogram contains calls for which stack usage can only be bounded by arithmetic analysis, but arithmetic analysis is disabled. Bound-T will not be able to bound the (total) stack usage of this subprogram with these options and assertions.
	Reasons	The command-line contains the option *-no_arithmetic* which disables arithmetic analysis generally, or the assertion file uses the option **no arithmetic** to disable it specifically for this subprogram.
	Solution	Recode the subprogram (and perhaps some of its callees) to avoid dynamic stack usage, or change the command-line options or the assertion options to allow arithmetic analysis of this subprogram. It may also be necessary to enable arithmetic analysis for some of the callees.
stack : Stack usage undefined	*Problem*	The stack usage analysis could not find any bound on the usage of the named *stack* in the current subprogram and current context.
	Reasons	The subprogram manipulates the stack in some way that Bound-T cannot analyse, or makes use of values (parameters or globals) that are not bounded by the context.
	Solution	Inspect the subprogram code to understand why Bound-T cannot bound the stack usage; then modify the code or add assertions to support the analysis. See section 3.11.
Target-program file-name not specified	*Problem*	The Bound-T command line does not give the target program file name; all arguments on the command line were interpreted as options.
	Solution	Check and correct the command-line syntax against chapter 6.
This program has no stacks.	*Problem*	Stack usage analysis was requested (*-stack* option) but the target program does not use any stacks.
	Reasons	The target processor or the cross-compiler have no stacks (that Bound-T can analyse).
	Solution	Check the relevant Application Notes for specifics on stack usage analysis for this target processor and compiler. Perhaps stacks are used only with specific compilation options. If there are no stacks, do not ask Bound-T for stack analysis on this target.
Too few arguments	*Problem*	Too few arguments given to Bound-T at start-up.

Error Message		Meaning and Remedy
	Solution	Restart with correct number of arguments. See section 6.
Unknown -arith_ref choice: *choice*	*Problem*	On the Bound-T command line, the *choice* argument that follows the option *-arith_ref* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -const_refine item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-const_refine* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -draw item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-draw* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -imp item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-imp* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -lines item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-lines* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -show item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-show* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -source item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-source* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -trace item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-trace* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -virtual item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-virtual* is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unknown -warn item: *item*	*Problem*	On the Bound-T command line, the *item* argument that follows the option *-warn* is not recognised.
	Reasons	Mistyped command line.

Error Message		Meaning and Remedy
	Solution	Correct the command line. See section 6.4.
Unrecognized option: *argument*	*Problem*	The Bound-T command line contains an option *argument* that is not recognised.
	Reasons	Mistyped command line.
	Solution	Correct the command line. See section 6.4.
Unresolved dynamic control flow	*Problem*	The actual memory address or addresses in a dynamic (indexed, pointer-based) jump instruction could not be determined. Bound-T is unabled to continue the control-flow analysis past this instruction and will interpret the instruction as a return from the subprogram under analysis.
	Reasons	The most common cause is a switch/case statement that is implemented using an indexed jump or an address table for which Bound-T could not determine the target addresses, perhaps because it needs arithmetic analysis but that analysis was disabled.
	Solution	Beware that the WCET given for this subprogram omits all code (and calls) that could have been reached (only) from the problematic instruction. Modify the target program to avoid such instructions, for example by using an if-then-elsif structure instead of the switch/case.
Use -help for help.	*Problem*	A reminder that the *-help* option makes Bound-T display help for the command-line syntax and options.
	Reasons	There were some errors in the Bound-T command line.
	Solution	Correct the command line.
Value of -output_sep must be a 1-letter string	*Problem*	On the Bound-T command line, the argument following the *-output_sep* option is invalid. It should be one letter or special character (punctuation).
	Solution	Correct the command line. See section 6.4. Remember to "escape" or "quote" special characters that may be significant for your command shell. For example, if you want to change the output separator to a semicolon under Linux, you should quote it (*-output_sep* ';') or escape it (*-output_sep* \;)
Worst-case path not found.	*Problem*	The search (in the *lp_solve* auxiliary program) for the longest execution path in the current subprogram, in the current context, failed for some reason.
	Reasons	No common reasons for this are known.
	Solution	Please contact Tidorum Ltd.

10 GLOSSARY

ABI — Application Binary Interface. A definition of how subprogram calls work on a specific target processor. Usually defines which registers (if any) are used for passing parameters and return values, which register (if any) is used as a stack pointer, and how the stack (if any) is laid out in memory.

Arithmetic analysis — The (optional) part of a Bound-T analysis that models the computations of the target program as a set of equations and inequations expressed in *Presburger Arithmetic* and then queries the model to find *loop counters* and bounds on the number of loop iterations. The *Omega Calculator* plays an essential part.

Assertion — 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. An assertion has two parts, the asserted *fact* and the *context* in which the fact holds. See chapter 5.

Basic block — The normal meaning is a maximal sequence of consecutive instructions in a program such that there are no jumps into the sequence or within the sequence, except possibly in the last instruction. In Bound-T, the meaning is a maximal sequence of flow-connected steps (see this term) in a control-flow graph such that the sequence is entered only at the first step and left only after the last step. Note in particular that a step in the sequence may correspond to an unconditional jump instruction in the target program. Bound-T also considers each "call step" (see this term) as its own basic block. In detailed output from Bound-T (see section 7.5) the term "node" is often used for basic blocks.

Branch — A jump or a call.

Call — 1. Static meaning: An instruction that suspends the execution of the current, or calling subprogram, executes another, or called subprogram, and then returns to the calling subprogram to continue the execution of the calling subprogram. The return point is often (but not always) the next instruction in the calling subprogram. The calling subprogram is also knowns as the *caller* and the called subprogram is also known as the *callee*. See also *call site*.

2. Dynamic meaning: The execution of a call instruction, transferring execution control from the call instruction (possibly after some delay instructions) to the first instruction (the entry point) of the callee.

Call path — A sequence (list) of *calls* (more precisely, *call sites*) such that, for each call in the list, the callee is the caller in the next call (if any). A call path represents a chain of nested subprogram calls that can define the *context* for the analysis of the callee of the last call in the path.

Call site — A point in the program (an instruction) that is a *call*, in the static meaning that term.

Callee — The subprogram that is called from another subprogram (the caller).
See *call*.

Caller — A subprogram that calls another subprogram (the callee). See *call*.

Call step — A special step in the control-flow graph of a caller subprogram that models the execution of a callee. Used to model the effect of a call. See *call* and *step*.

Cell — See variable.

Constant propagation — A method of simplifying a sequence of computations by executing any computation with known (constant) operands and propagating the known (constant) result to any computation that uses it. See section 6.5.

Context — 1. The call path (sequence of calls) that leads to a given invocation of a subprogram. Sometimes the analysis of a subprogram is context-dependent, for example the loop-bounds and WCET may depend on the context.

2. The part of the target program to which an assertion applies.

Context-free bounds — Execution bounds (bounds on the execution time and/or stack usage) for a subprogram that apply to all executions of the subprogram. Such execution bounds are derived by analysing the subprogram in isolation, without considering the context of a particular call or call path leading to the subprogram. Same as *universal bounds*.

Delay instruction — An instruction that statically follows a jump or call (that is, the next instruction in address order) but is executed before the transfer of control happens. That is, the jump or call takes effect only after the delay instruction is executed. Delay instructions are used in some pipelined processors to avoid disrupting the pipeline state. The alternative is to make a jump or call flush the pipeline, discarding some fetched and perhaps partially (speculatively) executed instructions.

Destination — The (address of) the instruction that is indicated by a branch instruction as the next instruction to be executed.

DOT — 1. A program for drawing graphs; part of the *GraphViz* package. See *www.graphviz.org* and section 7.6.

2. The textual language for describing graphs for the DOT program. Bound-T writes the *-dot* file in this language.

Dynamic call — A call in which the destination address (the address of the called subprogram) is not given statically in the instruction, but is computed at run-time.

Dynamic jump — A jump in which the set of potential destination addresses is not presented statically in the instruction, but is computed at run-time.

Entry address — The machine address of the first instruction in a subprogram – the first instruction that is executed when a call instruction transfers control to the subprogram.

ESF — Execution Skeleton File. The text file generated by HRT-mode analysis of an HRT target program and containing the information from the TPOF supplemented with execution skeletons containing WCET values. See chapter 1.4.

Eternal loop — A loop that cannot possibly terminate, either because there is no branch that can exit the loop or because all exit branches have been found to be infeasible. See section 5.17.

Executable file — A file that contains the compiled and linked form of a *target program*. Such a file contains the machine-code instructions and the constant data that will be loaded into the target processor as the initial memory state before the target program is started. The file usually also contains symbolic debugging information that connects source-level entities such as subprogram names and variable names to the machine-level properties such as the entry address of the subprogram or the memory address or register number of the variable.

Execution count — The number of times some part (node or edge) of a flow-graph is executed, usually referring to a worst-case execution path.

Fact — When discussing *assertions*, the condition or relation that is asserted, as opposed to the *context* of the assertion.

Full context — The sequence of nested calls (call sites) that lead from the a root subprogram to a particular execution of another subprogram is the full context of this execution. See also *suffix context* and *context*.

Function — A subprogram that returns a value as the meaning of the call, so that the call can occur in an expression.

Host platform — The computer on which Bound-T is run, as distinct from the target processor.

HRT — Hard Real Time; a principle for real-time program architecture, and a theory and tool-set for analysing such programs. An HRT program consists of threads and protected objects. See section 1.4.

ILP — Integer Linear Programming is an area of mathematical optimization in which the unknowns are integer variables, the objective function to be maximized or minimized is an affine expression of the variables and the variables are constrained by affine equalities or inequalities. Bound-T uses ILP, as implemented in the *LP_Solve* program, for the *IPET* stage of the analysis.

Implicit Path Enumeration Technique – see *IPET*.

Infeasible code — A part of a program that cannot be executed because it is conditional and the condition is always false (in the context under analysis).

Infeasible path — An path through a program or subprogram that cannot be executed because it contains conditional parts and the conditions cannot all be true in the same execution (in the context under analysis).

Input (variable) — A parameter or a global variable such that its value on entry to a subprogram is used in the subprogram and, in particular, has an effect on the execution time or stack usage of the subprogram. An input is *necessary* if its value must be known or bounded in order to find bounds on the execution time or stack usage of the subprogram.

Integer Linear Programming – see *ILP*.

IPET — The Implicit Path Enumeration Technique uses *ILP* to find the worst-case (or best-case) path in a flow-graph without explicitly trying (enumerating) all possible paths. IPET generates an ILP problem in which the unknown variables are the number of times each part (node or edge) of the flow-graph is executed and the objective function is the total execution time which is the sum of the times spent in each node or edge. The time spent in a node or edge is the product of the number of times this node or edge is executed (an unknown) and the constant worst-case (or best-case) time for one execution of the node or edge. The unknown execution counts (also called execution frequencies) are constrained by the structure of the flow-graph, by loop bounds and by other computed or asserted conditions on the execution. Solving this ILP problem gives one set of execution counts that leads to the worst-case (or best-case) execution time but does not give an explicit execution path; indeed there are usually many execution paths that give the same execution counts.

Jump — An instruction that explicitly specifies the address of the next instruction to be executed (without implying a suspension of the current subprogram, as in a call). There may be more than one potential successor instruction, from which the actual successor is chosen at run-time by a boolean condition or an integer-valued index expression.

Local stack height — The amount of stack space that a subprogram is using for its own data, at a given point in its execution. Usually includes any obligatory stack data such as the return address. Usually excludes stack space used by parameters for this subprogram; that space is instead counted in the local stack height of the caller. Vice versa, stack space for parameter that this subprogram provides to its callees is usually included in the local stack height measure. See section 3.11.

Loop body — In a natural loop, all the other nodes except the loop head. The only way for execution to reach a loop-body node is through the loop head. From any loop-body node there is at least one execution path that returns to the loop head.

Loop counter — A variable that grows on each iteration of the loop, such that the loop terminates when the counter reaches or exceeds some value. Of course, the counter may as well be decreased on each iteration, and terminate the loop when it reaches or falls below some value. The former is an *up-counter* and the latter a *down-counter*.

Loop head — In a natural loop, the unique node (basic block) that dominates (in the graph-theoretic sense) all the other nodes in the loop, which form the loop body. Any execution path that enters the loop does so at the loop-head.

LP_Solve — A support program that solves Integer Linear Programming (*ILP*) problems. Bound-T uses LP_Solve for the *IPET* phase of the WCET analysis. The executable program is called *lp_solve* or *lp_solve.exe*.

Natural loop — In a control-flow graph, a set of nodes (basic blocks) that forms a loop as defined under loop body and loop head.

Necessary input — See *input (variable)*.

Node — 1. In general, any node or vertex in a graph.

2. In a Bound-T control-flow graph, the term node means a *basic block*, which see.

Non-rectangular loop — A loop-nest in which the number of repetitions of the inner loop is not constant but depends on the current repetition of the outer loop. For erxample, a loop-nest that traverses the upper (or lower) triangle of a square matrix. Non-rectangular loops pose problems for Bound-T; see section 5.3.

Non-returning subprogram — A subprogram that never returns to its caller. For example, the C *_exit* function. See section 5.11.

Null context — See *universal context*.

Omega Calculator — A support program used for the *arithmetic analysis*. The Omega Calculator evaluates expressions and solves queries using systems of equations and inequations expressed in *Presburger Arithmetic*. The executable program is called *oc* or *oc.exe*.

Presburger Arithmetic — A form of algebra that deals with affine expressions of integer-valued variables and thus includes the operations of addition, subtraction and multiplication by an integer constant but excludes the multiplication of two or more variables with each other. Expressions can be compared for equality or inequality, relations can be combined with conjunction or disjunction, and both existential and universal quantification are available. Problems in Presburger Arithmetic are decidable (can be solved in a finite time) but the worst-case complexity is multiply exponential, as far as is known. Bound-T uses Presburger Arithmetic, as implemented in the *Omega Calculator*, for the *arithmetic analysis* of loop bounds.

Procedure — A subprogram which is not a function; it does not return a value as the meaning of the call, so the call can only occur as a statement, not as an expression.

Property — 1. A target-specific value or configuration setting that can be defined with a property assertion. See section 5.10.

2. A feature or characteristic of a loop or a call that can be used to identify the loop or call in an assertion. See sections 5.15 and 5.16.

Protected object — A component of an HRT program that is a passive entity and acts as a communication and synchronisation point for threads. See chapter 8.

Pruning — Simplifying a control-flow graph by removing parts (nodes and edges) that are infeasible. See section 6.5.

Rate-Monotonic Analysis — A way to analyse the schedulability of a multi-threaded program where the threads are periodic and scheduled by priority with pre-emption. Rate-Monotonic Analysis (RMA) assigns priorities to threads monotonically in order of thread period so that a short-period, high-rate threads have higher priorities than long-period, low-rate threads. With such a priority assignment the WCETs of the threads can be plugged into mathematical formulae that show if the thread set is schedulable (each thread can execute to completion without overrunning its period).

Rectangular loop — A loop-nest in which the inner loop repeats for the same number of times on each repetition of the outer loop. For example, a loop-nest that traverses all elements of a rectangular matrix in order by rows or columns.

Recursion — A cyclic chain of calls between subprograms. See section 5.18.

Reducible — A control-flow graph in which any loop is entered only through a unique loop head, and any two loops are either nested or entirely separate.

Resolving a jump/call — The analysis that determines the possible target addresses of a *dynamic jump* or a *dynamic call*.

RMA — See *Rate-Monotonic Analysis*.

Scheduling — The allocation of processor resources (execution time) to the several threads in a concurrent program. Specifically, the selection of which thread shall be running at every moment.

Stack — An area of memory that subprograms can use for their local variables and for parameters passed to their callees. Subprograms allocate and release stack space in a last-in-first-out way.

Stack usage — The (largest) amount of stack space that a subprogram uses, when stack space used by its callees is included. Also called *total* stack usage.

Static Single Assignment – see *SSA*.

Step — An vertex in a Bound-T control-flow graph that represents the smallest unit of program flow. A step usually models one machine instruction in the target program, but some complex instructions may be modelled by several steps and some special instruction sequences may be combined into one step. Steps are connected by (step) edges that model the flow of control from one instruction to the next. A maximal linear sequence of steps that can be entered only at the first step and left only from the last step is a *basic block*, often called a *node* in output from Bound-T.

SSA — Static Single Assignment. See *value-origin analysis* in section 6.5.

Subprogram	A callable (closed) subroutine: a function or a procedure.
Suffix context	A call path (which see) that leads to a particular subprogram (the callee of the last call on the path) is a suffix context for this subprogram and, in particular, for any execution of this subprogram in which the full context (which see) ends with this call path.
Take-off height	The caller's local stack height at the point of a call. Usually includes all stacked parameters for the call but excludes the return address (which is considered to belong to the callee's stack frame). See section 3.11.
Target processor	The processor that will (eventually) run the target program being analysed by Bound-T. Bound-T, however, is run on the host platform, which is usually, but not always, different from the target processor.
Target program	The real-time program that runs (or will run) on the target processor. The execution time of the target program is of interest. The target program may or may not be an HRT program. The program must be compiled and linked for execution on the target processor and stored in an *executable file* before it can be analysed by Bound-T.
Task	See thread.
Thread	An active component of a program, executing program statements sequentially. Some programs have a single thread of exection, but many real-time programs are multi-threaded, i.e. several threads are executing concurrently. The number of threads that can be (truly) executed in parallel depends on the number of processors in the target system. For Bound-T, the usual assumption is that there is one processor, which is shared among the threads via thread scheduling. See section 1.4.
TPO file	Threads and Protected Objects File. See *TPOF*.
TPOF	Threads and Protected Objects File. The user-supplied text file that lists and describes the structure of an HRT program, for HRT-mode analysis by Bound-T. See section 1.4.
Triangular loop	A special case of *non-rectangular loop*, which see.
Universal bounds	See *context-free bounds*.
Unreachable code	See *infeasible code*.
Universal context	Synonym for null context, applied when a subprogram is analysed in isolation, without considering the context of a particular call or call path leading to the subprogram.
Unresolved jump/call	A *dynamic jump* or *dynamic call* that has not been fully resolved. Thus, Bound-T may not know all the possible target addresses. Some parts of the target program may be missing from the analysis and the analysis results may be unsafe, for example the WCET bound may be less than the true WCET.
Value-origin analysis	A form of data-flow analysis that determines the possible origins of the value of a variable at a point where that variable is used. The origin is often an instruction that stores the result of some computation in the variable. However, an instruction that simply copies the value of another variable is not considered to be the origin of a value (copy propagation is applied). If the variable has no origin in the current subprogram then the variable is an input to this subprogram. Section 6.5 explains how Bound-T uses value-origin analysis. Value-origin analysis is similar to *SSA*.
Variable	A memory location or register in the target processor in which the target program stores some value. For Bound-T, the same as a *cell*, which see.

WCET
: Worst-Case Execution Time of a subprogram. The maximum time required to execute the subprogram in the target processor, when any initial execution state (parameter values, global values) is allowed.

  It is not defined if the WCET also allows any pattern of interference from interrupts and thread scheduling. Such interference could affect the performance of the processor cache, and increase (or decrease) the subprogram's execution time, even when the execution time of the interfering threads is excluded.

Worst-case execution time – see *WCET*.

Worst-case stack path
: The call-path, starting at a root subprogram, that consumes the largest amount of stack space when the stack usage of all path levels is included. See section 3.11. There can be several different call-paths that consume the same maximal amount of stack space.