Bound-T timing analysis tool

User Guide

TR-UG-001

Version 6.2
2008-02-20

Tidorum Ltd.

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

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

Document reference:	TR-UG-001
Document issue:	Version 6.2
Document issue date:	2008-02-20
Bound-T version:	3
Last change included:	BT-CH-0106
Web location:	*http://www.bound-t.com/user-guide.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. Some versions of Bound-T emit XML data with the *XML_EZ_Out* package written by Marc Criley at McKae Technologies.

Contents

Tables

Figures

This page is intentionally blank except for this note.

1 INTRODUCTION

1.1 What Bound-T is

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

The main function of Bound-T is to compute an *upper bound* on the *worst-case execution time* (WCET) 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. Such *schedulability analysis* is not a function of Bound-T, but many schedulability analysis tools are available. Some tools are listed at *http://www.tidorum.fi/bound-t/scheduling-tools.html*.

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 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

Figure 1 below illustrates the context in which Bound-T is used. The inputs are the compiled, linked executable target program, an optional file of assertions, and 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).

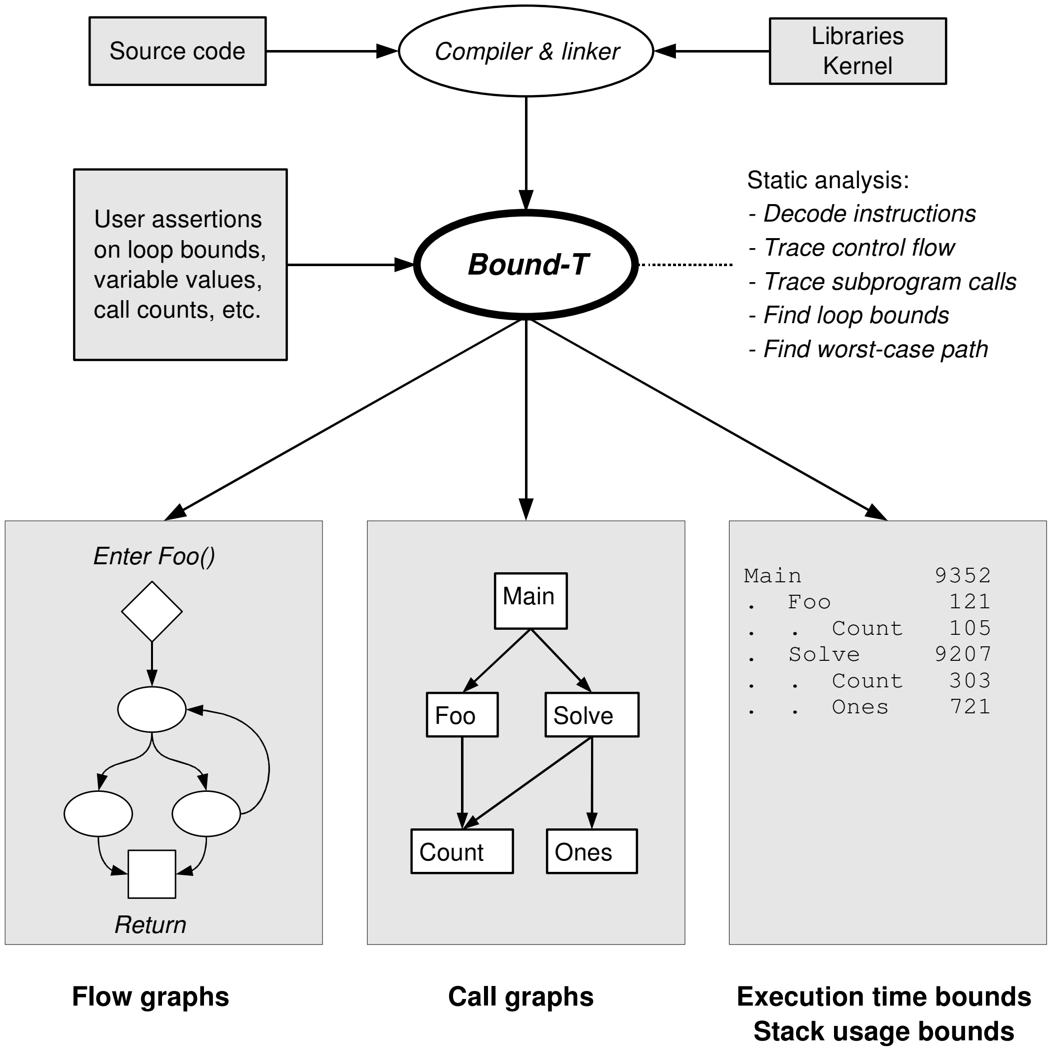
Target program, target processor, target computer

To use Bound-T effectively, you must know the structure of the *target program* – the program being analysed. In some cases, you may have to understand the architecture of the *target processor* that will run the target program, and perhaps of the *target computer* – the computer that contains the target processor plus various peripherals, memories and other devices.

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 Guide. Additional information for specific targets is provided in separate Bound-T *Application Notes* on which more below.

Host computer

The Bound-T tool itself is installed and executed on a *host computer* – your PC or workstation. Since Bound-T works entirely by static analysis, not by measurement or profiling, it needs no access to the target computer. You can use Bound-T to analyse a target program before the target computer even exists, and before the target program is complete enough to be executed on the target computer. All you need is a cross-compiler and linker that can generate the machine code for the target processor.



```
Main           9352
.  Foo          121
.  .  Count     105
.  Solve       9207
.  .  Count     303
.  .  Ones      721
```

Figure 1: Inputs and outputs

1.2 Overview of this User Guide

What the reader should know

This User Guide 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, not on the source code.

User Guide overview

This document is a *User Guide* that introduces Bound-T in a tutorial and informal way. The guide is organised into chapters as follows:

- Chapter 2 explains how Bound-T is used. It introduces the basic methods, options and results for execution-time analysis. The next-to-last section tells how to get started with the analysis of a real program.
- Chapter 3 gives more detail on *context-dependent* analysis, in which the execution time or stack usage of a subprogram can depend on actual parameter values or other context inherited from a particular chain of calls leading to that subprogram.
- Chapter 4 covers the analysis of stack usage. Stack overflow can be fatal to embedded programs; stack-usage analysis can make sure that overflow never happens.
- Chapter 5 suggests how to write programs that Bound-T can analyse.
- Chapter 6 is a glossary of terms and concepts related to Bound-T.

1.3 Other Bound-T documentation

This User Guide is not an exhaustive documentation of Bound-T. It is supplemented by other documents as follows.

Reference Manual

For full and detailed information on command-line options and output formats, look to the Bound-T Reference Manual at *http://www.bound-t.com/ref-manual.pdf*. The Reference Manual also outlines the analysis process and lists and explains the warning and error messages.

Assertion Language manual

Most users of Bound-T need to write *assertions* to guide and constrain the analysis. Assertions are written as text. This User Guide gives several examples of assertions, but you should refer to the Bound-T Asssertion Language manual at *http://www.bound-t.com/assertion-lang.pdf* for the full syntax and meaning of the assertion language. The possible warning and error messages from the assertion parser are also described there, not in this User Guide nor in the Reference Manual.

Target-specific Application Notes

This User Guide, the Reference Manual, and the Assertion Language manual describe the general, target-independent usage and features of Bound-T. For specific target processors there are usually additional command-line options, perhaps additional forms of analysis and outputs, additional warning and error messages, and target-specific aspects of the assertion language, such as the names of the processor registers. All of these are described in the Bound-T Application Notes for your target processor at *http://www.bound-t.com/app_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.

Hard Real Time programming model

Bound-T contains special high-level support for target programs that follow the *Hard-Real-Time (HRT)* programming model, an architectural style for concurrent, real-time programs originally defined by the European Space Agency.

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.

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.

This User Guide describes how Bound-T is used in its basic mode, without the special HRT features. There is a separate manual that explains how to use Bound-T in HRT mode. See *http://www.bound-t.com/hrt-manual.pdf*.

1.4 Typographic Conventions

We use the following fonts and styles to show the role of pieces of the text in this guide:

-option	A command-line option for Bound-T.
symbol	A mathematical symbol or variable.
`text`	Text quoted from a source file or a command.
keyword	A keyword (reserved word) in a programming language or in the Bound-T assertion language.

2 USING BOUND-T

2.1 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 generate executable forms of your programs.

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.

2.2 Installing Bound-T

Host and target computers

This section explains briefly and in general how to install Bound-T on your computer. The computer on which Bound-T is installed and used is known as the *host* computer. It is usually not the same as the *target* computer on which the target program runs or will run. The host and target computers usually have different types of processor, for example a powerful RISC processor on the host computer and a small 8-bit microcontroller on the target.

Note that you do not need to have a target computer available in order to use Bound-T, and even if you have one it does not need to be connected to your host computer. Bound-T works purely by static analysis of the machine code of the target program, and this static analysis is done by running Bound-T on the host computer. Bound-T does not use any form of measurement or execution of the target program on the target computer itself.

Delivery medium and specific instructions

Specific installation instructions for each supported type of host computer and host operating system are included with each delivery of Bound-T, on the delivery medium or as hard-copy enclosed with the medium.

On a typical host computer, 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 the *bin* folder. The auxiliary programs are the same for all targets.

The main steps in the installation are to copy the *bin* folder to the host computer and to set up the PATH variable, or an equivalent mechanism, to make the contents of *bin* available on the command line.

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

Disk space requirements

The disk space consumed by Bound-T depends on the host computer but allowing 30 MB should be ample for one type of target processor. Each additional target processor type needs about 15 MB more disk space.

Processor and memory requirements

Bound-T usually places about the same demands on the host computer'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 asserting loop-bounds manually, or by simplifying the target program.

Host-specific usage instructions

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

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.

2.3 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 and/or the stack usage of the named subprograms within the given executable target program file. The Reference Manual describes all the command-line options, but you will see several examples later in this guide.

This computation can either succeed fully automatically, or succeed only after some additional assertions are given. The Assertion Language manual describes the full assertion language, but there are several examples in this guide.

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 1.3 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. When you install Bound-T you can also change the name, or define an alias or abbreviation for it.

2.4 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 an upper bound on the worst-case execution time (WCET) of *sum_vector* as the last field of the output line:

```
Wcet:summer.exe:sum.c:sum_vector:4-8:1103
```

The WCET is given as the number of instruction cycles (1103 in this example, when the function is compiled for the SPARC ERC32 processor). 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, 4 – 8, are the source-code line-numbers of the subprogram. Bound-T gets these line-numbers from the compiler-generated mapping between code addresses and source-code lines, which often leads to a little fuzziness, for example by omitting the line numbers for syntactical brackets like the '{' and '}' that enclose the body of the function.

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*).

2.5 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" subprograms named 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, as will be explained in Chapter 3. 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:17-18, offset 00000014
```

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 bound, for example:

```
Wcet:summer.exe:sum_two.c:sum_two:4-12:9119
```

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*. Chapter 3 explains how such context-dependent analysis works in more detail.

The results for each call are reported in the following form:

```
Loop_Bound:summer.exe:sum.c:sum_two@8-=>sum_vec_n:17-18:39
Loop_Bound:summer.exe:sum.c:sum_two@9-=>sum_vec_n:17-18:1233
Wcet_Call:summer.exe:sum.c:sum_two@8-=>sum_vec_n:14-18:445
Wcet_Call:summer.exe:sum.c:sum_two@9-=>sum_vec_n:14-18:8647
```

The fourth colon-delimited field shows the call-path context for each result. For example, the first `Loop_Bound` is valid when *sum_vec_n* is called from *sum_two* at line 8 of *sum.c*, while the second `Loop_Bound` is valid for the call from line 9.

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*.

2.6 Drawing call-graphs and control-flow graphs

DOT drawings

Bound-T does not have a graphical user interface, but it can draw the call-graph and the control-flow graphs of the subprograms that are analysed.

The graphs are generated as output files that describe the structure of the graphs for the DOT tool. The DOT tool is a part of the GraphViz suite that you can download from *http://www.graphviz.org/*. The DOT tool reads the graph structure file, decides how to lay out the nodes, edges, and labels in a two-dimensional drawing, and generates the drawing in some graphical file format, for example PDF or JPEG. You can then view the drawings using a tool such as Acrobat Reader for PDF files.

The DOT description of a graph is in text form. You can make small tweaks to the graphs by simple text editing, for example to change from portrait format to landscape format.

Command-line options for graph drawing

To make Bound-T draw call-graphs or flow-graphs, you must

- use the option *-dot* or the option *-dot_dir* to say where the drawings go, and
- use some *-draw* options to say what the drawings should show.

The *-dot* option puts all drawings into one file. You write the name of this file after the *-dot*, separated by whitespace (so it becomes the next argument on the command line). For example, the following command (where the ... stands for more options and arguments) puts all the drawings (all the DOT text) into the file *graphs.dot*:

```
boundt -dot graphs.dot -draw ...
```

The *-dot_dir* option puts each drawing in a separate file of DOT text, and puts these files in one directory folder. You write the name of the directory folder after the *-dot_dir*, separated by whitespace. For example, the following command puts each drawing into its own file in the directory *graphdir*:

```
boundt -dot_dir graphdir -draw ...
```

With *-dot_dir*, Bound-T names the drawing files automatically as explained in the Reference Manual. The names of call-graph drawings start with "*cg_*" and the names of flow-graph drawings start with "*fg_*". Bound-T tries to include the name of the relevant subprogram in the name of the drawing file, but it may have to remove or change special characters into ordinary letters in the file-name, so don't expect a perfect match for all names.

The *-draw* option is followed by a keyword that selects or excludes a particular form of drawing or particular information that may be present in a drawing. The Reference Manual explains the *-draw* options fully; here we focus on examples.

The most common use of *-draw* options is probably the simple form *-draw total* which draws the call-graph in the default form (each subprogram seen as one node) and one flow-graph for each analysed subprogram, giving the total number of executions and the total execution time for each basic block in each subprogram.

First example: sum_vector

For a first example of control-flow graphs, take the subprogram *sum_vector* from section 2.4 (page 8). Add the *-dot* and *-draw* options to the Bound-T command:

```
boundt -dot grf.dot -draw total summer.exe sum_vector
```

This command puts the call-graph diagram and the flow-graph diagram in the file *grf.dot*. You can then convert this file to a PostScript file, for example, with the DOT command:

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

Assume, for example, that *sum_vector* is compiled for the SPARC processor. In this case the call-graph is trivial because *sum_vector* calls no other subprograms. Figure 2 below shows the drawing of the flow-graph of *sum_vector* from this analysis.

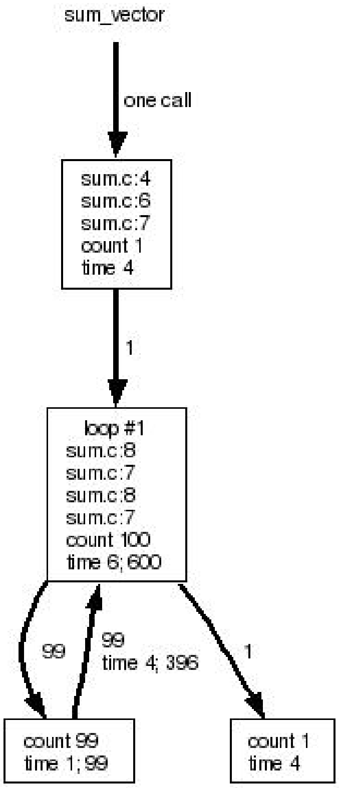


Figure 2: Flow-graph of *sum_vector* on SPARC

The flow of execution in flow-graph drawings is mostly top-down. The entry point is identified by the entry arrow at the top, labelled to show how many calls of the subprogram are included in the total (because this drawing comes from *-draw total*, remember). The rectangular nodes are the basic blocks and are labelled with the source-code lines that correspond to each instruction in the block. The last two lines in the node-label show the execution count and execution time of the node. For example, the first node (the entry point) in Figure 2 is executed once ("count 1") and takes 4 cycles to execute ("time 4").

The edges (arcs) represent flow of execution from one node to another, in the direction of the arrow-head. Edges are labelled with the execution count and with the execution time (if not zero). The edges on the worst-case execution path are drawn with a greater thickness (bolder) than other edges. In Figure 2, however, all edges are on the worst-case execution path.

The second node and the lowermost node on the left side of Figure 2 represent the loop in *sum_vector*. The node that forms the loop head is labelled with the loop number ("loop #1"). The "count" labels show that the loop head is executed 100 times while the other node in the loop is executed only 99 times.

For nodes and edges that are executed more than once the "time" labels show two numbers separated by a semicolon. The first number is the time for one execution; the second number is the total time for all executions. For example, the edge that returns to the loop head from the other node in the loop is labelled "time 4; 396" meaning that one execution of the edge can take up to 4 cycles, so the 99 executions take up to 4 × 99 = 396 cycles in total. (As an aside, this edge has such a large time, 4 cycles, because it is a short path from one SPARC floating-point addition instruction to another such instruction, which may force the program to wait for the SPARC floating-point unit to complete the first instruction.)

The lowermost node on the right-hand side of Figure 2 is reached when the loop terminates. There is no edge out from this node, so the subprogram returns after executing this node.

Disassembled code in the flow-graph

The following draw options will show the disassembled machine instructions, instead of the source-code lines, in the flow-graph:

```
boundt -dot grf.dot -draw total -draw decode -draw cond
   -draw no_line -draw no_count -draw no_time summer.exe sum_vector
```

The resulting flow-graph drawing for *sum_vector* on a SPARC is shown in Figure 3 below.

Compared to Figure 2, the source-line references are removed (*-draw no_line*) and so are the execution counts and times (*-draw no_count* and *-draw no_time*) and instead (*-draw decode*) the drawing shows the disassembled SPARC instructions, one per line, in execution order, with the address of each instruction in the left margin. Thanks to *-draw cond* the edges are labelled with the logical condition for taking the edge, which is *true* for unconditional edges and otherwise some logical formula using the *Z* (zero) and *N* (negative) condition flags from the SPARC status register.

Call graph of sum_two

Consider again the subprogram *sub_two* from section 2.5 (page 10). The following command makes Bound-T analyse it and draw the call-graph and control-flow graphs:

```
boundt -dot grf.dot -draw total summer.exe sum_two
```

The call-graph is no longer trivial because *sum_two* calls *sum_vec_n* (twice, in fact). If the program is compiled for the Atmel AVR processor, the call-graph looks even more interesting, as Figure 4 below shows. The AVR has no hardware floating-point unit, so the compilers instead use library routines to implement floating-point operations. In this example, both *sum_two* and *sum_vec_n* call the library routine for floating-point addition, *?F_ADD_L04*, which is not visible at all in the source code of this program in section 2.5. This library routine in turn calls three other library routines: *?TEST_P_L04* and two anonymous routines that Bound-T identifies with their entry addresses, 0000E2 and 00002E hex.

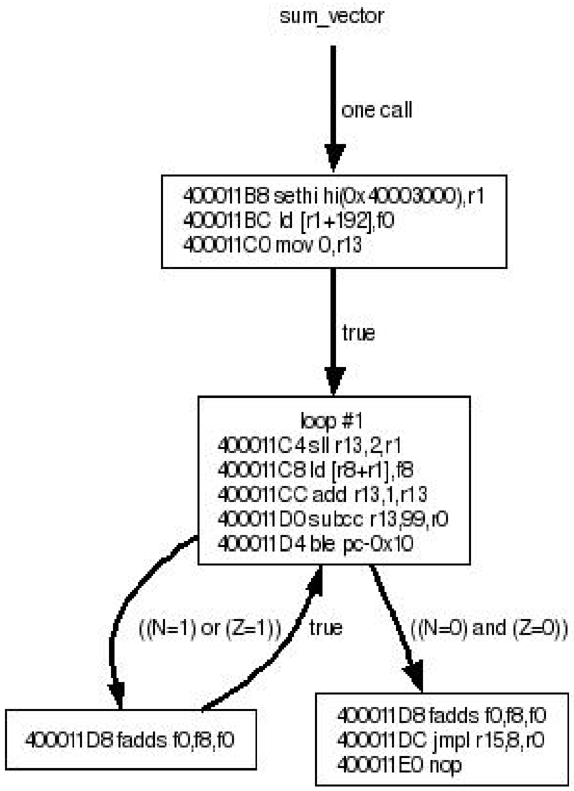


Figure 3: Flow-graph of sum_vector with SPARC instructions

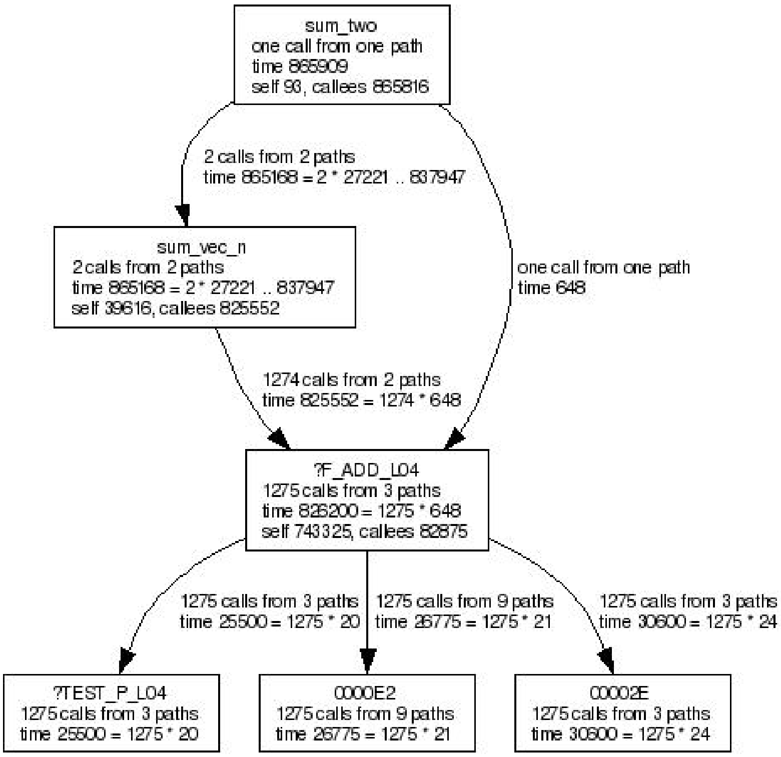


Figure 4: Call-graph of *sum_two* on AVR

The nodes (rectangles) in the call-graph represent subprograms. The root subprogram, *sum_two* in this example, is on the top. The node label gives the name of the subprogram, the number of calls of this subprogram that are executed on the worst-case execution path, the number of different call-paths that can lead to these calls, and the total time (upper bound) of these executions.

For example, Figure 4 shows that *sum_two* is executed once ("one call") from one call path, which is always and naturally the case for the root subprogram. Furthermore, the total time is the WCET bound for *sum_two*, 865 909 AVR clock cycles ("time 865909").

The last line in the label for a subprogram shows the division of the total execution into time spent in the subprogram itself ("self") and time spent in other subprograms that are called from this subprogram ("callees"). For *sum_two* only 93 cycles of the total time is spent in *sum_two* itself, the rest being spent in the callees: *sum_vec_n* and the floating-point routines.

Looking at the lower levels of the call-graph, the library routine *?F_ADD_L04* is called (executed) 1275 times, along three call paths: firstly, from *sum_two* directly; secondly, from *sum_two* via the first call of *sum_vec_n*; and thirdly, from *sum_two* via the second call of *sum_vec_n*.

When a subprogram is executed more than once, the "time" label shows how the total time is the product of the number of executions and the time per execution. Thus, the "time" label for *?F_ADD_L04* shows that the total time, 826 200 cycles, equals the product of 1275 executions and the time per one execution, 648 cycles. Here each execution of *?F_ADD_L04* is given the same, context-independent, execution time bound; in reality, however, the execution time of this subprogram depends on the values of the floating-point numbers to be added.

For a subprogram that has context-specific bounds, the time per execution is shown as a range *min .. max*. For example, the "time" label for *sum_vec_n* shows that the total time, 865 168 cycles, comes from 2 executions where the time per execution ranges from 27 221 cycles to 837 947 cycles.

The edges (arcs) in the call-graph represent calls (caller-callee relationships). They point from the caller subprogram to the callee subprogram. The edge labels show the number of call paths and the execution time bounds.

Drawing context-dependent execution bounds: the bounds graph

For suprograms with context-dependent execution-time bounds, such as *sum_vec_n*, you may like to draw a graph that is like a call-graph but separates the different execution bounds. Use the option *-draw bounds* to create such "bounds-graph" drawings. For example:

```
boundt -dot grf.dot -draw bounds summer.exe sum_two
```

This command makes a bounds-graph drawing in the file *grf.dot*. It makes no call-graph drawing (because call-graph and bounds-graph drawings are mutually exclusive options) and no flow-graph drawings (because the command has no *-draw* option for them). Figure 5 below shows the bounds-graph drawing for *sum_two* on the SPARC target processor.

A call-graph drawing would show *sum_vec_n* once, covering both calls of this subprogram and showing a range of execution-time bounds, 445 .. 8647 cycles. In contrast, the bounds-graph drawing in Figure 5 shows the two calls separately, because they have different bounds on execution time.

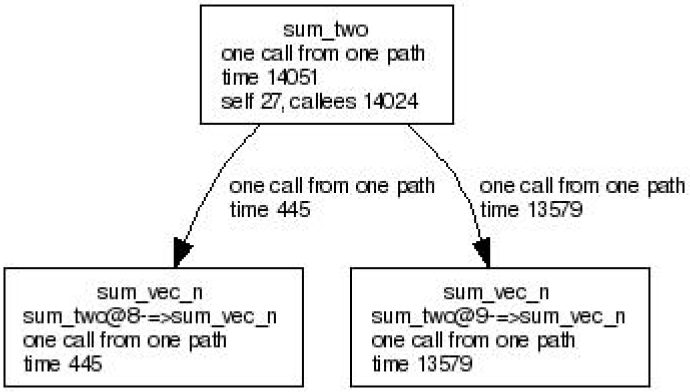


Figure 5: Bounds-graph drawing of *sum_two* on SPARC

Drawing context-dependent flow-graphs

The earlier examples of flow-graph drawings all use the option *-draw total* which, for each subprogram, creates a single flow-graph drawing that summarises all executions of that subprogram on the analysed worst-case path for the root subprogram. For subprograms with context-dependent execution-time bounds, such as *sum_vec_n*, you may like to draw a separate flow-graph for each context or for chosen contexts. Bound-T provides six different *-draw* options to choose which flow-graphs to draw. The earlier examples in this section used *-draw total*; the example below uses *-draw all*; the Reference Manual explains the other four options.

Use the option *-draw all* to make a separate flow-graph drawing for each context in which a subprogram has been analysed and given context-specific execution bounds. For example:

```
boundt -dot grf.dot -draw all summer.exe sum_two
```

This command makes a call-graph drawing in the usual form and then makes flow-graph drawings for each subprogram in all contexts. For this example, on the SPARC processor, the only subprogram with context-specific bounds is *sum_vec_n*, which has two bounds corresponding to the two calls of this subprogram from *sum_two*. Figure 6 below shows the resulting two flow-graph drawings, side by side.

In this example, the only difference between the two flow-graph drawings in Figure 6 is the indication of the call-path, given at the very top and showing that the calls lie on different lines (numbers 8 and 9) in *sum_two*, and the execution counts of nodes and edges in the loop, reflecting the different value of the parameter *n* in the two calls.

You may wonder why Figure 6 shows that *sum_vec_n* has an execution path that bypasses the loop entirely, going from the entry node via another node to the return node. There is no *if* statement in the source-code (page 10) that could make the loop conditional. However, the number of loop iterations depends on the parameter *n*, which may be zero or negative, in which case the loop should not be executed at all. The compiler inserts a test and branch for this case because the compiler can then generate a bottom-test loop body (do until) instead of a top-test (while do). Since the actual value of n in both contexts is positive, this bypass path is not on the worst-case path and therefore shows zero execution counts in Figure 6 and the edges on the bypass path are drawn as thin lines.

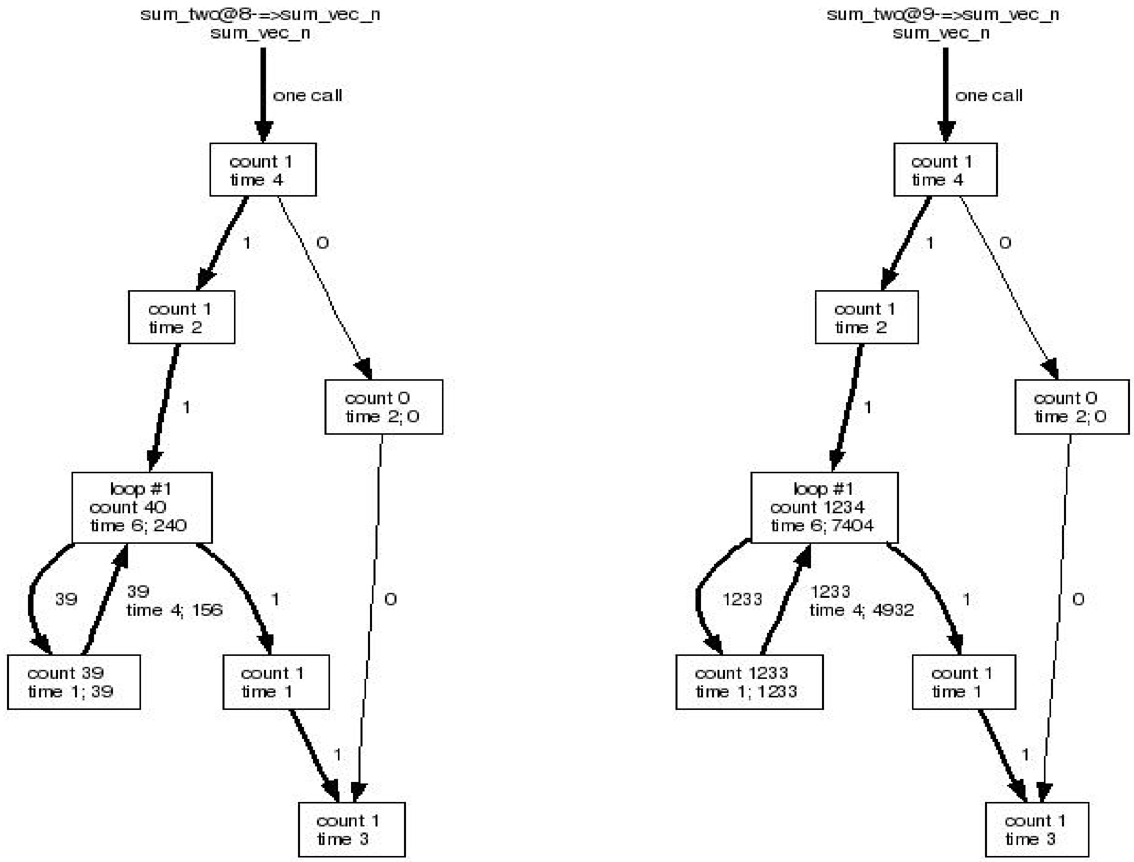


Figure 6: Context-specific flow-graphs for *sum_vec_n* on SPARC

2.7 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*. The Assertion Language manual 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. The Assertion Language manual has more to say on eternal loops.

2.8 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 earlier in this chapter 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 5 advises on programming styles that help Bound-T analyse the program.

2.9 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 2.7, 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 may become "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 instructions 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 (including processor registers) 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 auxiliary 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 form $2^n - 1$ and 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. (Except that some C++ compilers do store enough information in the executable file about the class hierarchy and virtual functions for Bound-T to discover all possible targets of a given virtual function call.)

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 analyse loops only in 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.

When a subprogram has an irreducible flow-graph Bound-T cannot find the loop structure. Thus, it cannot find loop repetition bounds by analysis and cannot accept assertions on loop repetition bounds. This does not hamper stack usage analysis, but it does pose a problem for execution-time analysis.

There is a work-around, however. If the repeated execution paths in an irreducible flow-graph always pass through calls, and if you can assert bounds on how many times each such call can be repeated, then Bound-T may be able to compute execution-time bounds even in the absence of a reducible loop structure. However, you must specifically tell Bound-T to attempt this, by asserting that the assertions on the calls are **enough for time**. Refer to the Assertion Language manual for details.

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. The method is explained the Assertion Language manual.

2.10 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. The Assertion Language manual shows some examples.

2.11 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. The Assertion Language manual 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 CONTEXT-DEPENDENT ANALYSIS

3.1 Introduction

This chapter will talk about subprograms that have context-dependent bounds. It usually talks about bounds on execution time (WCET) but applies as well to bounds on stack usage. Stack-usage analysis is described in Chapter 4.

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. 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.

3.2 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.

Call sites

In the kind of context-dependency or input-dependency that Bound-T uses, the context is defined by the call sites, defined as follows:

- 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.

Call paths

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 7. The calls are numbered *call1* to *call6* for identification.

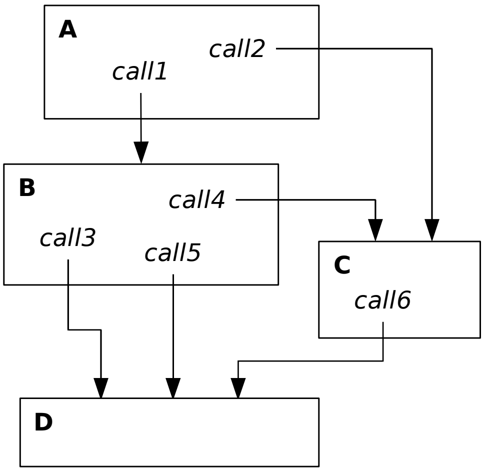


Figure 7: Example of calls and call paths

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.

3.3 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.

Full context

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. We define:

- 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 7, 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.

Suffix context

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 7 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*.

3.4 Searching for the necessary context

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 7. The description is long; if you feel that you understand the idea you can skip this subsection.

The analysis of the program in Figure 7 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 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.

3.5 Summary

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

3.6 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.

3.7 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.

4 STACK USAGE ANALYSIS

4.1 Stacks and stack overflow

A *stack* is an area of memory that holds data that the currently executing subprogram needs, but that can be discarded when the subprogram ends and returns to its caller. This releases memory space for use by another subprogram, in its turn.

On the other hand, when the current subprogram itself calls some other subprogram, the stack data for the *calling* subprogram remain in the stack, and the new data for the *called* subprogram are allocated or “pushed” on “top of” the caller's data. When the called subprogram returns, its data are discarded or “popped” from the stack. So the “top” of the data in the stack grows and shrinks dynamically as subprograms are called and return. Space in the stack is allocated and released in last-in-first-out (LIFO) order.

What is in the stack?

Implementations of procedural languages like C and Ada typically use stacks for:

- the *local variables* (“automatic” variables in C) of a subprogram,
- the *parameters* of subprograms,
- the *return addresses* for use by the return instructions,
- *intermediate results* of computations, and
- other things that may be required by a particular processor and its coding rules, for example space to save registers when a trap or interrupt happens.

Although most programs use stacks, stacks are not absolutely necessary for most programs. All types of data listed above could also be kept in processor registers or in statically allocated data memory (at fixed addresses) – if there are enough registers or enough memory. Stacks become very useful, perhaps unavoidable, for recursive programs and for subprograms that must be reentrant.

How much stack space does a subprogram need?

Looking at the C or Ada source-code of a subprogram gives only a rough idea of the amount of stack space that the subprogram needs. The actual amount depends on how the compiler allocates memory: which parameters are passed in the stack; which local variables are stored in the stack; which intermediate results are stored in the stack; and how much stack memory is used for a value of a given type. In addition, some processors have coding rules that force the compilers to allocate more stack space, even if the compiler-generated code does not use this space directly.

Example

Here is a C function that has a parameter and some local variables. The function takes a null-terminated string as its only parameter, computes the number of times each decimal digit character '0' .. '9' occurs in the string, and returns the maximum number of occurrences. The loop counter variable *d* for short loops (0 .. 9) is declared with a type *count_t*, defined elsewhere as *unsigned int* or *unsigned char* depending on the target processor. The source-lines are numbered in the left margin for reference.

```
unsigned int max_freq (char *str)
/* The frequency of the most frequently occurring
* digit '0' .. '9' in the string str.
*/
{
  unsigned int freq[10];
  unsigned int max;
  char c;
  count_t d;
  char *p = str;

  /* Initialize frequencies to zero: */

  for (d = 0; d <= 9; d++) freq[d] = 0;

  /* Count the frequency of each digit in the string: */

  while (*p)
  {
    c = *p;
    if ((c >= '0') && (c <= '9')) freq[c-'0'] ++;
    p++;
  }

/* Find the maximum frequency: */

  max = freq[0];
  for (d = 1; d <= 9; d++)
  {
    if (freq[d] > max) max = freq[d];
  }

  return max;
}
```

How much stack space does this function need? On most target processors, the return address is put in the stack. For a small processor the return address may be only 16 bits, or even less, while a processor with a larger code memory may need 24, 32, or even 64 bits for the return address. On most processors the parameter *str* is passed in a register, but some processors have very few registers and so may pass it in the stack. Depending on the size of the data memory, a *char** may be only 8 bits, or 16 bits, or even 64 bits.

The cross-compiler can place any one of the local variables *freq*, *max*, *c*, *d*, *p* in the stack, or in statically allocated memory, or in registers (but register allocation is unlikely for an array like *freq*). If some of these variables are placed in the stack, the stack space required depends on the size of the types *unsigned int*, *char*, *count_t*, and *char** for the chosen target processor and the chosen cross-compiler.

Table 2 below shows the stack space that this *max_freq* function needs on some target processors and cross-compilers.

Some cross-compilers can report the amount of stack space that each subprogram needs. But all compilers do not do this, and those that do can usually report only the *static* stack space, not the *context-dependent* usage on which more later.

Table 2: Stack space for *max_freq* on several processors

Processor	Compiler	Stack space octets	Remarks
Intel 8051	IAR Systems 7.30B	0	These compilers use the processor stack only for return addresses. Data are by default stored in statically allocated memory (and by default subprograms are not reentrant).
Intel 8051	Keil C51 8.09	2	
Intel 8051	SDCC (Small Device C Compiler)	2	
Renesas H8/300	GCC 3.3.1 *count_t = unsigned int*	20	The stack holds both return addresses and some data. The only variable of type *count_t* is held in a register.
Renesas H8/300	GCC 3.3.1 *count_t = unsigned char*	22	Stack space is larger than in the case above, although the *count_t* variable is smaller and is still in a register.
Renesas H8/300	IAR Systems *count_t = unsigned int*	34	Stack space is 70% larger than for the GCC compiler. However, compilation options may play a role, and should be investigated.
Renesas H8/300	IAR Systems *count_t = unsigned char*	32	
SPARC V7 (ERC32)	Gaisler Research BCC	144	Most of the space is a buffer for storing registers in case of a “register file overflow” trap. The SPARC programming rules require such a buffer for most subprograms.
Atmel AVR	IAR Systems	2, SP stack 24, Y stack	This compiler uses the processor stack (SP) only for return addresses, and uses a compiler-defined stack (Y) for data.

For most target processors, Bound-T considers the space for the return address as a part of the stack usage of the calling subprogram, not of the callee. Thus, although the table shows zero stack space for *max_freq* on the Intel 8051 processor with the IAR compiler, a call of *max_freq* uses 2 octets of stack space for the return address. The non-zero stack usage of *max_freq* on the same processor for the Keil and SDCC compilers is due to the return addresses for calls from *max_freq* to some compiler-provided library routines. The IAR compiler for the Intel 8051 does not generate any library calls from *max_freq*.

The IAR Systems compiler for the AVR processor uses *two* stacks. The processor's “native” stack is accessed by the Stack Pointer register (SP) and is automatically used by the call and return instructions to store and retrieve the return address. For data variables the compiler defines a “software” stack and uses the AVR register Y as the stack pointer. Either or both of these stacks can overflow if too little space is allocated for it at link time. Such software stacks are common for 8-bit processors. For example, both the IAR and Keil compilers for the Intel 8051 use software stacks when some subprograms must be reentrant.

Context-dependent stack usage

Some programming languages allow local variables, typically arrays, of a *dynamic* size. That is, the number of elements of the array is not a static constant, but is computed by a dynamic expression, perhaps depending on the subprogram's parameters or on the current values of global variables. If such dynamically sized arrays are stored in the stack (which is not always the case) the stack space needed by the subprogram becomes dynamic and context-dependent, too.

Stack overflow and how to avoid it

The memory area allocated for a stack typically has a fixed size that is set at link-time and cannot be increased at run-time. *Stack overflow* happens if the program executes a chain or nesting of subprogram calls such that the total stack space used by all these subprograms exceeds the size of the stack area.

Stack overflow often makes the program fail (for example, through a *stack-overflow trap*) or behave randomly when non-stack memory is wrongly *overwritten* and its content destroyed. You can reduce the risk of stack overflow by allocating a large memory are for the stack. However, some small processors have stacks of a hardware-defined, fixed size – for example holding at most six return addresses – and even processors where the stack is set at link-time may not have memory to waste on safely oversized stacks, especially since each thread usually needs its own stack area. How can you know what size the stack area should be? If the stack size is fixed by the hardware, how can you be sure that the size is large enough?

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.

4.2 Stack analysis to avoid stack overflow

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 an optional function of Bound-T and is activated by the command-line options *-stack* or *-stack_path*. Stack usage can be analysed together with execution time (*-time*, the default) or separately (*-no_time*).

For example, the following command analyses (only) the stack usage of the subprogram *max_freq*, within the executable file *freq.exe*, generated by GCC for the the Renesas H8/300 processor, and using the corresponding version of Bound-T:

```
boundt_h8_300  -stack  -no_time  -lego  freq.exe  _max_freq
```

The option *-lego* specifies the particular type of H8/300 processor in use. It is not important for stack usage analysis. The underscore '_' in front of the subprogram name is added by GCC.

Results

Bound-T reports the stack-usage bounds in output lines that starts with the word *Stack*. For example, the following line reports the stack usage bound of the *max_freq* subprogram:

```
Stack:freq.exe:freq.c:_max_freq:1-33:SP-stack:22
```

This output line shows that the subprogram *max_freq*, together with its callees if any, needs at most 22 units of space on the stack called *SP-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 like the H8/300 the above *Stack* line means 22 octets of stack space, while on a processor built around 32-bit words it could mean 22 words = 88 octets. The unit is of course defined in the Application Note for the target, as are the names of the stacks used on that target.

How did it do that?

Bound-T finds an upper bound on the stack usage for a subprogram by analysing and modelling the computations that change the value of the *stack pointer* – the processor register that points to the top of the stack. For compiler-defined software stacks the stack pointer is some compiler-defined global variable that may be held in a register or in memory.

The analysis of changes to the stack pointer is similar to the analysis of loop-counter variables in the execution-time analysis, but is usually simpler. *Constant propagation* analysis is often enough, and it is seldom necessary to invoke the costly Presburger arithmetic analysis.

Still, there are certainly programs that change the stack pointer in ways that are too complex for Bound-T's current methods of analysis. Bound-T then reports an unbounded stack usage.

The -stack_path option

The examples above used the *-stack* option to activate stack usage analysis. With this option Bound-T reports only an upper bound on the stack usage – a single number per subprogram and stack – but gives little information on how this total amount is consumed by the various subprograms in the call-graph. The *-stack_path* option also shows the *worst-case stack path*: the sequence of calls that would actually use this amount of stack space, if the sequence happens during execution. We will return to this after more explanation of stack analysis across calls.

4.3 Stack usage in loops

Loops must pop as much as they push

You may have noticed that the example above, concerning the stack usage of the *max_freq* subprogram, did not talk at all about bounds on the loops in *max_freq* although loop-bounds are so important in the analysis of execution time. Loop-bounds are not needed for stack-usage analysis because Bound-T assumes that *the repeated execution of a loop body has no net effect on the stack height*.

In other words, while the code in a loop body can push data onto the stack, it must pop the same amount of data before going back to repeat the loop. Likewise, if the loop body pops data from the stack, it must push the same amount of data before repeating the loop.

It is very unusual for a compiler to generate a loop body that does not balance pushes with pops, because then the stack would grow or shrink continuously as the loop runs. Compilers generally do not know how many times loops repeat, so the compiler would not know the height of the stack within the loop or after the loop.

Except on the last iteration

However, the *last* iteration of a loop body – the one that leads to exit and termination of the loop – *can* have a net effect on the stack height.

Example of push and pop in loops

Common high-level languages like C and Ada have no statements that (directly) push or pop the stack, so for such examples we must use an assembly language. The following subprogram is written in the assembly language for the Atmel AVR processors and contains a loop that executes push and pop operations. Source-lines are numbered in the left margin, for reference, and commented in the right margin, after the semicolon ';' sign.

```
 1   xubaloo:                 ; The subprogram name and entry point.
 2           ldi    r16,0     ; Initialize the loop-counter r16 to zero.
 3   xloo:                    ; The start (head) of the loop.
 4           push   r1        ; Push register r1 on the stack.
 5           cpi    r16,17    ; Compare the loop-counter r16 to the literal 17.
 6           breq   xend      ; If r16 equals 17, exit from the loop (go to xend).
 7           pop    r1        ; Pop register r1 from the stack (pushed in line 4).
 8           inc    r16       ; Increment the loop-counter r16.
 9           rjmp   xloo      ; Jump back to repeat the loop once more.
10   xend:                    ; The loop exits to this point in the code.
11           pop    r1        ; Pop register r1 (pushed in line 4).
12           ret              ; Return from the subprogram xubaloo.
```

There are no stack operations before the loop, so the loop is started (going from line 2 to line 3) with the local stack height at zero. Disregarding the loop-counting instructions, the loop body pushes register *r1* (line 4), thus increasing the stack height to 1 octet, then pops *r1* (line 7), and repeats (going from line 9 to line 3). There is no net effect on the stack height because the pop balances the push, so this loop is suitable for stack usage analysis in Bound-T.

However, the loop-termination test is placed between the push and the pop, so when the loop exits, the last push of *r1* has not been balanced by a pop. Thus the stack height after the loop (at line 10) is 1 octet; the net effect of the execution of the loop is to increase the stack height by one octet. This octet is removed by the last pop (line 11) and the subprogram can return normally. As already said, Bound-T can analyse loops with such unbalanced exit paths, as long as pops and pushes balance on all paths that repeat the loop.

This command uses the AVR version of Bound-T to analyse the stack usage of this subprogram:

```
boundt_avr  -stack  -no_time  -at90s8515  prg.exe  xubaloo
```

The result for the processor (SP) stack usage is the following:

```
Stack:prg.exe:luups.s90:xubaloo:1-12:SP:1
```

The 1 octet of stack usage for *xubaloo* consist of the one octet (*r1*) that is explicitly pushed and then popped.

4.4 Stack usage in calls

The stack height profile

We have spoken of “the” stack usage of a subprogram, but in fact the stack pointer can change several times during the execution of a subprogram, so the stack usage can vary – and usually does vary – as execution progresses. Figure 8 below illustrates this for one execution of one subprogram. The horizontal axis shows execution flow (time) and the vertical axis shows the height of the stack, relative to its height on entry to the subprogram – the *local stack height* in this subprogram. This boxy curve that shows stack height as a function of time is called the (local) *stack-height profile* of the subprogram.

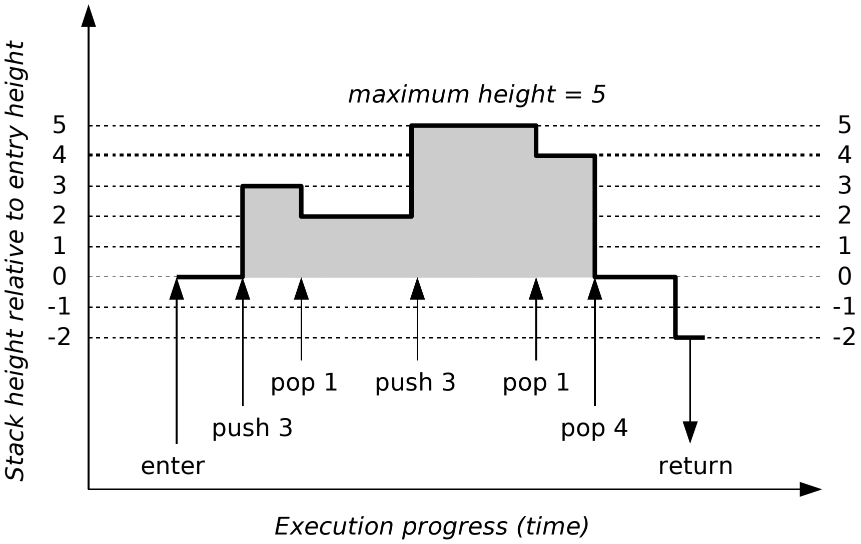


Figure 8: Stack height profile

Let's go through the significant points in Figure 8 in the order they happen – from left to right. First, on entry to the subprogram the stack height is zero by definition (remember we are talking about the local stack height, not the total stack usage at this point). Then, after perhaps executing some other instructions, the subprogram pushes 3 octets on the stack, increasing the stack height to 3. Later, the subprogram pops 1 octet, decreasing stack height to 2. After a while, the program again pushes 3 octets to reach a stack height of 5, which is also the maximum (local) height in this execution of this subprogram. Later still, the subprogram first pops one octet, leaving a stack height of 4 octets, and then pops 4 more octets, returning the stack height to zero. When the subprogram finally returns, the return instruction pops the return address from the stack, leaving a *final stack height* of –2 octets if we assume that a return address uses 2 octets of stack space.

In this example we assumed that the subprogram calls no other subprograms. This means that the largest local stack height, 5 octets, is also the largest total stack usage of this subprogram (not including the stack used by possible callers).

Stack usage at a call; the take-off height

Now consider another subprogram that calls the subprogram shown in Figure 8. We can combine, or stack together, the local stack-height profile of the caller and the local stack-height profile of the callee (from Figure 8) to get their sum, as shown in Figure 9 below. The sum becomes the stack height and usage profile for the caller.

Going through the significant points in Figure 9 in the same way as above, the first point is the entry to the caller subprogram, where the local stack height of the caller is zero by definition. Next the caller pushes 2 octets on the stack, and then calls the subprogram shown in Figure 8. The call instruction pushes the return address on the stack, which increases the caller's stack height to 4, and then execution flows from the caller to the callee. The *take-off (stack-) height* of the call is defined as the local stack-height in the caller when execution passes to the callee. In this example it is 4 octets.

While the callee is executing, the total stack usage of the caller and callee is the sum of the take-off height and the local stack height in the callee. This sum starts from 4 on entry to the callee, increases to a maximum of 9 (take-off height 4 plus callee maximum stack height 5) and decreases to 2 (take-off height 4 plus final callee stack height –2) when the callee executes a return instruction that pops the return address from the stack. The caller then pops 2 octets and returns, so the final stack height of the caller is –2 octets.

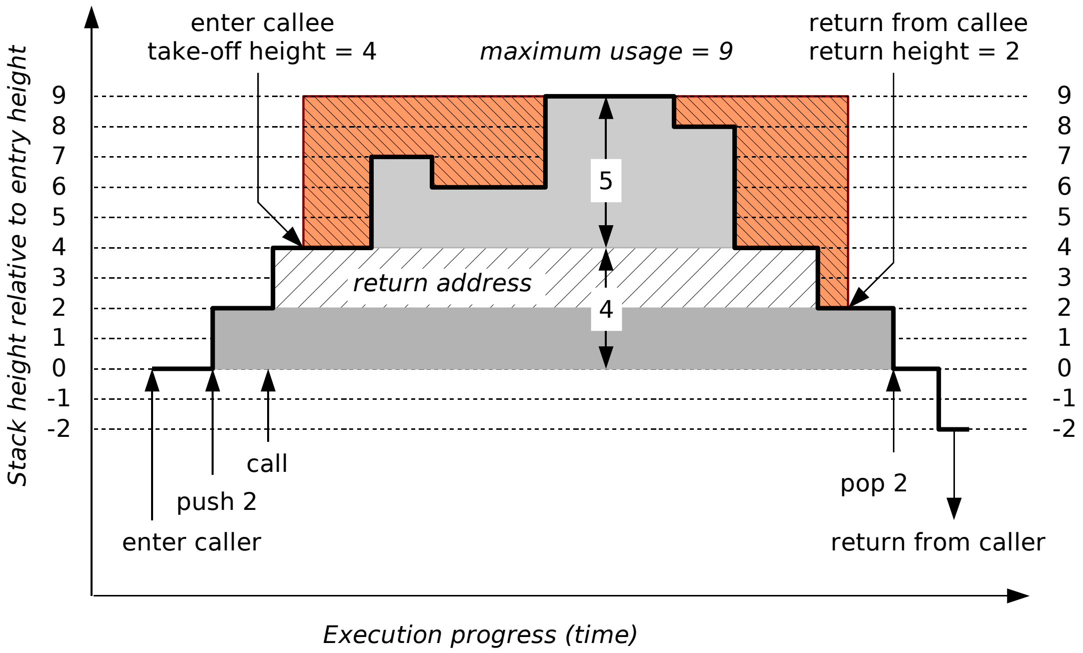


Figure 9: Stack height and usage over a call

The take-off height and parameter passing

As an aside, note that the take-off height of a call is important not only for stack usage analysis, but also for any context-dependent analysis of the callee when some inputs are passed in the stack. The reason is that the take-off height defines the mapping from the caller's stack frame (offsets relative to local stack-height zero in the caller) to the callee's stack frame (offset relative to local stack-height zero in the callee).

For this reason, it is generally best if the take-off height for a given call is constant – the same for any execution of the call during one and the same execution of the callee. The take-off height can be context-dependent – for example, it can depend on some of the caller's parameters – but it should not, for example, depend on the loop counter, when the call is in a loop.

The worst-case stack path

In general a subprogram calls several other subprograms. Each call can have a different take-off height, and the stack usage of each callee can be different, too. The total stack usage of a call is the sum of the take-off height and the maximum stack usage of the callee at this call.

Figure 10 below shows the stack-usage profile of a subprogram – let's give it the name *A* – that calls other subprograms *B*, *C* (twice), *D*, and *E*. For clarity the figure omits the profiles of the callees and instead shows each callee as a rectangle with a height equal to the maximum stack usage of the callee.

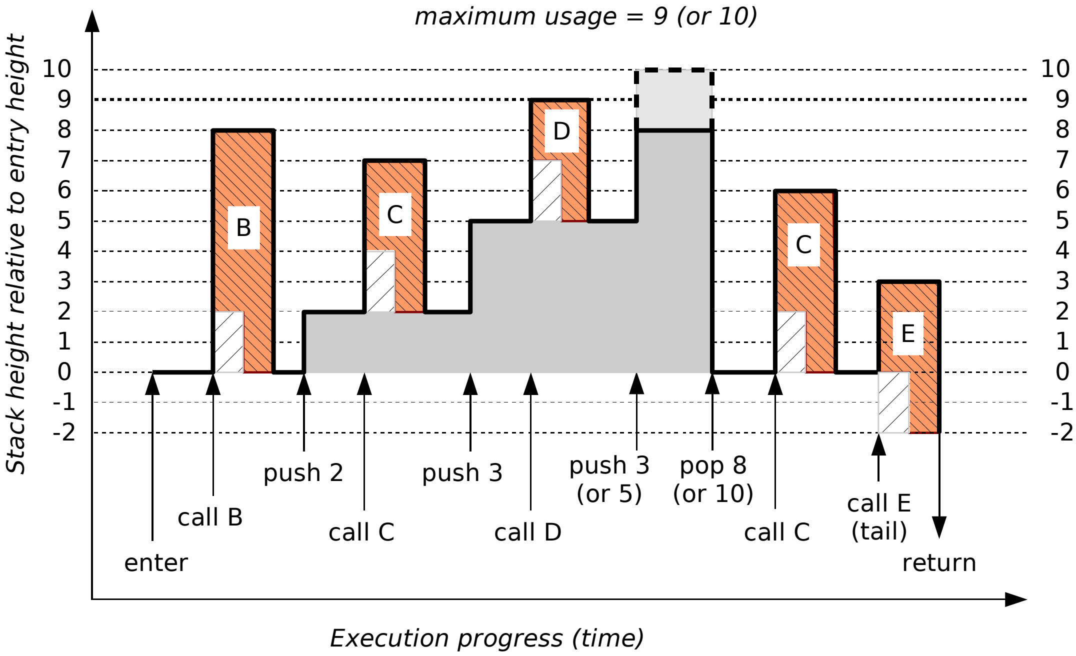


Figure 10: Stack-usage profile of subprogram *A*

You can see how the total stack usage of each call depends (is the sum of) the take-off height and the callee's stack usage. For example, the first call of *C* causes a total usage of 7 octets, from a take-off height of 4 octets plus 3 octets in *C*. The second call of *C* uses only 6 octets, from a take-off height of 2 octets plus 4 octets in *C* – evidently, *C* has a context-dependent stack usage. The call with the largest total stack usage is the call to *D*, using a total of 9 octets: 7 octets for *A* (including the return address) and 2 octets for *D*.

Figure 10 shows two different stack-usage profiles. The first (solid) profile happens when the third push instruction in *A* pushes only 3 octets; the second (dashed) profile happens when this instruction pushes 5 octets.

In the first (solid) profile, the highest stack-usage overall is in the call to *D*, which means that the call $A \rightarrow D$ is the *first* call on the *worst-case stack path* for *A*. The next call on this worst-case path, if any, is found by looking at the stack-usage profile of *D*, in the same way; if the maximum usage happens at a call $D \rightarrow F$ then this is the *second* call on the worst-case stack path for *A*, and so on.

In the second (dashed) stack-usage profile in Figure 10, the highest overall stack-usage does not happen at a call, but at the third push instruction in *A*, which increases the local stack height of *A* to 10 octets. In this case, the worst-case stack path for *A* ends at *A* itself.

In summary, the worst-case stack path for a subprogram shows the sequence of calls starting from this subprogram that forms the highest point in the stack-usage profile of this subprogram. The path ends when the highest point comes from local usage, not from a call. 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, and only when you use the -*stack_path* option on which more shortly.

Tail calls

The last call shown in Figure 10, the call to subprogram *E*, may look peculiar because the return address (shown as a diagonally hatched white rectangle) occupies the same stack location as the return address for the calling subprogram *A*. This illustrates a *tail call*: a call that is the last action in the caller and which can be implemented by a simple jump from the caller to the callee. There is no need to save a return address here, because there is no need to return to the caller (*A*). Indeed this means that when the callee (*E*) returns, execution flows to the return address of the caller (*A*), bypassing the caller itself.

Bound-T may not detect tail calls *as* calls, because in many processors they look exactly like ordinary jumps, not like calls at all. In Figure 10 the call to *E* would then be analysed as the execution, within the caller subprogram *A*, of all instructions in *E*, including a push of 3 octets, a pop of 3 octets, and a return. If a subprogram is called only by (undetected) tail calls, Bound-T may not even see it as a separate subprogram, only as a part of each caller subprogram.

Example of a nested calls and a worst-case stack path

For an example of nested calls and the worst-case stack path, consider the following C functions that call each other and the function *max_freq* shown earlier (page 38):

```
char* more_freq (char *x, char *y)
/* That one of x or y that has the higher max_freq. */
{  unsigned int fx, fy;
   fx = max_freq (x);
   fy = max_freq (y);
   return (fx > fy ? x : y);
}

unsigned int length (char *s)
/* The length of the string up to a terminating null char. */
{  unsigned int n = 0;
   while (*s) {s++; n++; }
   return n;
}

unsigned len_more (char *a, char *b)
/* The length of that string, a or b, with the higher max_freq. */
{  return length (more_freq (a, b));
}
```

Figure 11 below shows how these functions call each other, starting from *len_more*.

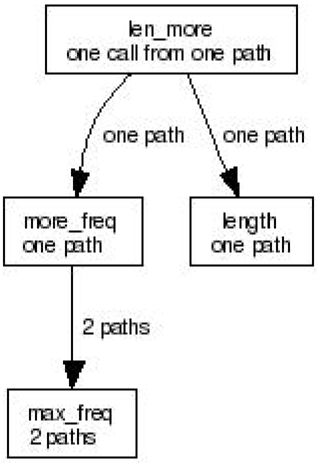


Figure 11: Call graph of *len_more* and callees

Suppose we compile these functions for the Renesas H8/300 processor with the GCC compiler, and then analyse their stack usage with the Bound-T command:

```
boundt_h8_300  -stack  -no_time  -lego  prg.exe  _len_more
```

The resulting stack usage bounds of each subprogram are the following, where the important data are shown in bold style for clarity:

```
Stack:tp_um_stack_call.exe:subs1.c:_max_freq:17-46:SP-stack:22
Stack:tp_um_stack_call.exe:subs2.c:_length:25-29:SP-stack:0
Stack:tp_um_stack_call.exe:subs2.c:_more_freq:13-20:SP-stack:30
Stack:tp_um_stack_call.exe:subs2.c:_len_more:34-36:SP-stack:32
```

The *Stack* output line for *len_more* shows that this subprogram, together with the subprograms that it calls, needs at most 32 octets of stack space. But it does not show which, if any, of the callees contributes to this upper bound. For example, if you would like to change the code to use less stack, you would not know which subprograms to attack first, in other words, which subprograms are "critical" for the stack usage. In this simple example there are not many subprograms to choose from, but a real application may have hundreds or thousands of subprograms and call paths.

The option *-stack_path* activates stack usage analysis and also shows a worst-case stack path, a kind of "critical path" for the stack usage. You simply use this option instead of *-stack*:

```
boundt_h8_300  -stack_path  -no_time  -lego  freq.exe  _len_more
```

This produces the same Stack lines as above (with -stack) but also shows the worst-case stacl path by a sequence of output lines starting with *Stack_Path*, except for the last line, which starts with *Stack_Leaf*. There will be one *Stack_Path* line for each subprogram (each level) in the worst-case stack path and these lines traverse the path in top-down order. The lines are rather long, however:

```
Stack_Path:freq.exe:freq.c:_len_more@34-35=>_more_freq:34-35:SP-stack:32:2:2:30
Stack_Path:freq.exe:freq.c:_more_freq@16=>_max_freq:16:SP-stack:30:8:8:22
Stack_Leaf:freq.exe:freq.c:_max_freq:17-46:SP-stack:22:22::
```

The following table formats the important information more legibly; each row corresponds to one of the above output lines, in the same order, and thus to one call on the worst-case stack path. The last row represents the last level of the path. There is no deeper callee in that row because here the maximum local stack height is also the maximum stack usage.

Table 3: Worst-case stack path for *len_more* on H8/300 with GCC

Caller	Callee	Stack usage, caller	Max local height, caller	Take-off height	Stack usage, callee
len_more	*more_freq*	32	2	2	30
more_freq	*max_freq*	30	8	8	22
max_freq		22	22		

Figure 12 below illustrates this worst-case stack path as a stack-usage profile in the same style as Figure 9. However, for clarity the space used for return addresses is not hatched, and the vertical axis (stack space) is not to scale.

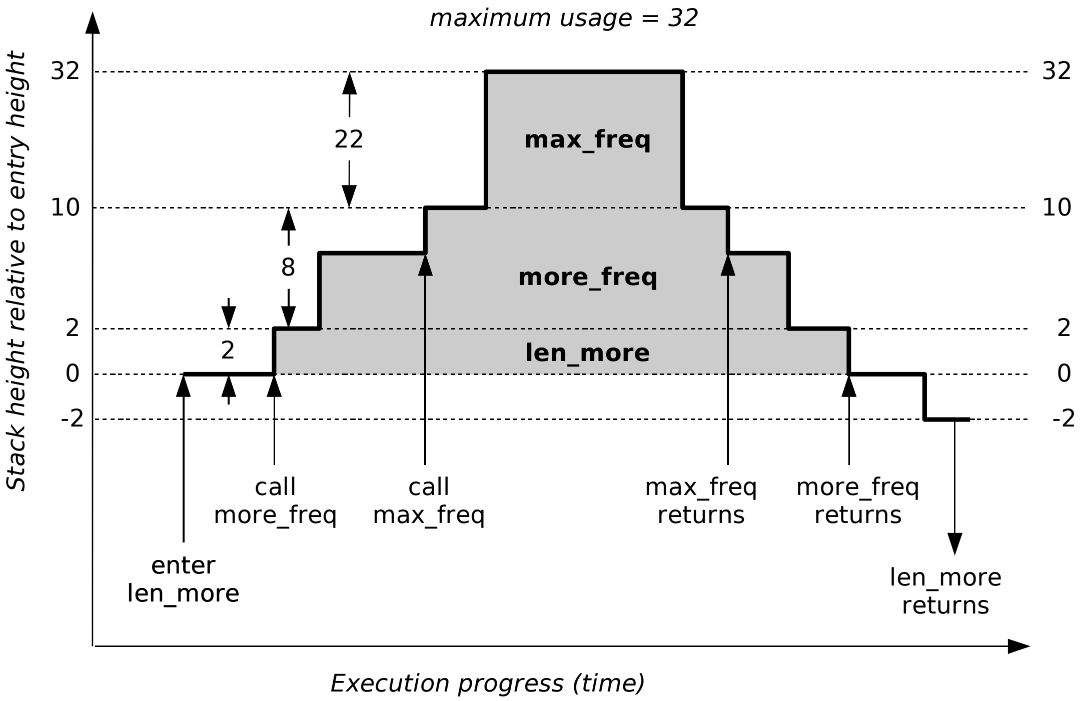


Figure 12: Stack-usage profile (worst path) for *len_more*, H8/300, GCC

The worst-case stack path is often also the longest (deepest) 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.

Making the stack usage smaller

You can often reduce the total stack-usage bound of a root subprogram by reducing the stack usage (local stack heights and take-off heights) in the subprograms on the worst-case stack path. However, this does not always work on the first try, because the worst-case stack path that Bound-T shows may be only one of several call-paths that have the same stack-usage bound. If that happens, and you reduce the stack usage of the worst-case path that Bound-T shows in the first analysis, a new analysis may give the same stack-usage bound and show another worst-case path that uses this amount of stack space. You may have to repeat the reduction and analysis for the new path, to get a real reduction in the overall stack-usage bound.

Another annoying thing that may happen to you is that you manage to reduce the stack usage of some subprogram(s) on the worst-case path by a large number, say 95 octets in all, but a new analysis shows a much smaller reduction of the total bound, for example by only 2 octets. This happens if there is another call-path that uses only 2 octets less stack space than the worst-case stack path in the first analysis, and thus this other call-path becomes the new worst-case stack path in the second analysis. You must now look at the subprograms on this call-path and try to reduce their stack usage, and so on.

Worst-case stack paths for several stacks

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 different worst-case paths.

Different worst-case paths for time and stack

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.

Context-dependent analysis of 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-path suffixes. 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: $\ldots \rightarrow A \rightarrow B$.

Assertions for stack usage

Earlier in this guide you saw how to assert an upper bound for the execution time of a subprogram when, for some reason, you do not want Bound-T to find the bound by analysing the subprogram or when Bound-T is unable to analyse the subprogram.

In the same way and for the same reasons you can assert the stack usage of a subprogram, instead of analysing it. Here is how to assert that the subprogram *piler* uses at most 71 units of stack space:

```
subprogram “piler”
  stack usage 71;
end “piler”;
```

Simple, wasn't it? But wait, that assertion only works when the program has exactly one stack. For example, a program compiled with the IAR Systems C compiler for the Atmel AVR processor uses two stacks: the hardware stack, called the “SP” stack, and a compiler-defined software stack, called the “–Y” stack. Bound-T will then reject the assertion above because the assertion does not specify which of the two stacks uses 71 units. Not to worry, just add the name of the stack. If you are talking about the “SP” stack, say:

```
subprogram “piler”
  stack “SP” usage 71;
end “piler”;
```

If you are talking about the "–Y" stack, say:

```
subprogram "piler"
  stack "-Y" usage 71;
end "piler";
```

But on processors with several stacks, if you want to avoid analysing a subprogram you must usually assert bounds on both stacks:

```
subprogram "piler"
  stack "SP" usage 12;
  stack "-Y" usage 31;
end "piler";
```

Assertions on final stack height

It is also possible to assert the final stack height of a subprogram. Such assertions can be necessary to let Bound-T find the stack-height profile of the callers of this subprogram, because the final stack height determines the net effect of the call, on the caller's stack height. To assert the final stack height, simply use the keyword **final** instead of **usage**. For example, this says that the *piler* subprogram lowers the "SP" stack height by 2 units:

```
subprogram "piler"
  stack "SP" final -2;
end "piler";
```

5 WRITING ANALYSABLE PROGRAMS

5.1 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-scase 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.

5.2 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*.

5.3 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.

5.4 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.

5.5 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.

5.6 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.

5.7 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.

5.8 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.

5.9 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, as explained in the Assertion Language manual.

5.10 Final words

We hope that this guide has explained enough to get you started with Bound-T. If you meet problems, need more advice, or have suggestions for changes or extensions to Bound-T, do contact Tidorum; we will be pleased to hear from you.

6 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. The Assertion Language manual explains the syntax and meaning of assertions.

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 the term "node" is often used for basic blocks, as shown in the Bound-T Reference Manual.

Bounds graph

A variant of *call-graphs* (which see) where each node represents a particular analysis (execution bounds) of a subprogram. Thus, if a subprogram has different execution bounds, in different contexts, the subprogram is represented by as many nodes in the bounds-graph. See section 2.6 (page 16) for an example.

Branch

A jump or a call. Sometimes specifically a *conditional* jump which has more than one possible successor instruction, as opposed to an *unconditional* jump which has exactly one successor.

Call

1. Static meaning: An instruction that suspends the execution of the current, or calling subprogram, and directs execution flow to another, or called subprogram. When the called subprogram finishes it usually 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 graph

A graph that represents the flow of execution between subprograms in a program. The graph nodes represent subprograms and the edges represent calls (call sites) between subprograms. In other words, the edges represent caller-callee relationships between subprograms. There is no explicit representation of returns from calls. It is implicitly assumed that each call returns to the caller, or to the caller's caller, or to some point even higher in the call-path. Bound-T can create call-graph drawings. See section 2.6 for examples.

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 of 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 *storage cell*.

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. The Bound-T Reference Manual has more to say on constant propagation.

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. Section 3 explains how Bound-T uses context information.

2. The part of the target program to which an assertion applies. See the Assertion Language manual.

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*.

Control-flow graph

See *Flow graph*.

Delay instruction

An instruction that statically *follows* a jump or call (that is, the next instruction in address order after the jump or call) 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. In pipelined processors that lack delay instructions a jump or call usually flushes the pipeline, discarding some fetched and perhaps partially (speculatively) executed instructions. This happens especially for conditional jumps and calls.

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 section 2.6 in this guide.

2. The textual language for describing graphs for the DOT program. Bound-T writes the *-dot* file in this language. See section 2.6 in this guide.

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 destination address (where execution is to continue) is not given 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 section 1.3.

Essential input

A input variable for a subprogram such that bounds on the execution time or stack usage of the subprogram cannot be found without knowing the value of this variable, or bounds on the value. See *input*, *context*, and Chapter 3.

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. The Bound-T Reference Manual discusses various forms of loops and their properties.

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 bounds

Bounds on the execution time (WCET) and/or the stack usage, for a given subprogram, and perhaps for a given *context*. Execution bounds can be derived by Bound-T or asserted by the user.

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. See the Bound-T Assertion Language manual.

Final stack height

See *Stack height, final*.

Flow graph

A graph that represents the flow of execution (flow of control) within a subprogram. Each node in the graph stands for a *basic block* (which see), and each edge stands for execution flow, or transfer of control, from one basic block to another block (or to the same block, for a small loop). Section 2.6 shows several examples of flow-graphs as drawn by Bound-T.

Frame pointer

A register or variable that points to the start of a *stack frame*, which see.

Full context

The call-path that leads all the way from 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.

Ghost loop

A loop that appears to be repeated for some number of times although it is never started (entered from outside the loop). In some special cases, Bound-T's use of the IPET method can create ghost loops in the IPET solution for the worst-case execution path of a subprogram. To create a ghost loop all the following conditions must hold: The subprogram contains an unbounded loop (a loop without loop-repetition bounds); other assertions on the number of repetitions of parts of the loop body bound the number of executions of the whole loop body; the loop is in a conditional branch of the flow-graph; the

enough for time assertion forces Bound-T to apply the IPET method in spite of the unbounded loop; and the resulting worst-case execution path does not start the loop, either because some assertion makes the loop unreachable, or because the branch with the loop has a smaller execution time (bound) than an alternative branch. Under these conditions the loop becomes a ghost loop because the IPET ILP constraints allow a positive execution count for the loop body even if the loop is not started. The remedy is to add a loop-repetition assertion, which can be sloppily overestimated without harm.

Host computer
The computer on which Bound-T is run, as distinct from the *target processor* that runs (or eventually will run) the target program under analysis.

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 1.3.

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*.

Indirect call
See *dynamic call*.

Indirect jump
See *dynamic jump*.

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 *essential* 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.

Irreducible
A control-flow graph that is not *Reducible*, which see.

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

See *Stack height, local.*

Loop

A part of a subprogram that can be executed more than once in a single call of a subprogram, thanks to a jump “back” to an instruction that was already executed. In other words, a cycle in the control-flow graph of the subprogram.

Loop body

All the flow-graph nodes in a *loop* and all edges between these nodes. Edges that enter the loop from the outside, or that leave the loop, are not included.

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, ghost

See *Ghost loop*.

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. Any execution path that enters the loop does so at the loop-head. See also *Reducible*.

Loop repetition

A loop repeats when execution flows from within the loop back to the loop head.

Loop start

A loop starts when execution flows from outside the loop into the loop (for a natural loop, into 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* with a *loop head*. See also *Reducible*.

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. Such a node contains a sequence of *steps*, 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. The Bound-T Reference Manual discusses various forms of loops and their analysis.

Non-returning subprogram

A subprogram that never returns to its caller. For example, the C *_exit* function.

Null context

See *universal context*.

Omega Calculator

A support program used for the *Presburger 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, inequality, less, or greater. Relations can be combined with conjunction or disjunction. 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 and other dynamic behaviours (dynamic jumps and calls and dynamic memory addressing).

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 the Assertion Language manual. For example, the number of wait states to be assumed for memory accesses.

2. A feature or characteristic of a loop or a call that can be used to identify the loop or call in an assertion. For example, the property that the loop contains a call to a given subprogram. See the Assertion Language manual.

Protected object

A component of an HRT program that is a passive entity and acts as a communication and synchronisation point for threads. See section 1.3.

Pruning

Simplifying a control-flow graph by removing parts (nodes and edges) that are infeasible (logically unreachable). Bound-T normally prunes all flow-graphs as one step in the analysis, as explained in the Bound-T Reference Manual.

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. Bound-T cannot analyse recursive programs automatically. The Assertion Language manual explains how you can use assertions to cut the recursion chain into analysable parts.

Reducible

A control-flow graph in which any loop is entered only through a unique *loop head* node, 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 a given time.

SSA

Static Single Assignment. See *value-origin analysis* and the Bound-T Reference Manual.

Stable stack

See *Stack, stable*.

Stack

An area of data memory in which storage space is allocated in a last-in-first-out fashion. The memory area for a stack is usually statically created at link-time, contiguous in the address space, and of a fixed size. At run-time, space in the stack is usually allocated starting from one end of the area (the *bottom* of the stack, although not always the low-address end) and proceeding toward the other end; a *stack pointer* shows the boundary between the allocated part and the unallocated (free) part of the area. Stacks are commonly used for storing subprogram return addresses and the local variables ("automatic" variables) of subprograms. Upon a subprogram call a part of the stack is allocated for this (invocation of the) subprogram; this part is called the *stack frame* of this (invocation of the) subprogram. Upon return from the subprogram the stack frame is usually deallocated, although some compilers delay the deallocation until a later point. Thus, the amount of space allocated in the stack, or the *stack usage*, varies dynamically during a program's execution. *Stack overflow* happens when the program tries to allocate more stack space than the stack area can hold. Stack overflow can make the program abort, compute wrong results, or fail in some other way. Static program analysis can compute an upper bound on the stack usage; if the size of the stack area is defined accordingly, stack overflow is safely avoided.

Stack frame

The section of a stack that is allocated for (an invocation of) a subprogram. Commonly used to hold the return address and local variables and working space of the subprogram (invocation). Data in the stack frame is usually addressed (accessed) by means of a static offset from the (dynamic) stack pointer. The size of the stack frame for a given subprogram can be static (same for all invocations of the subprogram) or dynamic (depend in some way on the parameters or other context of the invocation). When the size of the stack frame is dynamic programs often use a secondary pointer, the *frame pointer*, that points at the start of the stack frame. The stack pointer itself points to the end (or beyond the end) of the stack frame, and thus the offsets from the stack pointer to statically placed data in the frame would be dynamic, while offsets from the frame pointer are still static.

Stack height, final

The local stack height (which see) on return from a subprogram. Equals the net amount of stack space allocated (positive) and deallocated (negative) during the execution of the subprogram. The execution of a subprogram call changes the local stack height of the caller by an amount equal to the final stack height of the callee.

Stack height, local

The amount of stack space that is allocated for a particular (invocation of a) subprogram, at a particular point in the execution of the subprogram. Formally, the difference between the current value of the stack pointer and the value of the stack pointer on entry to the current subprogram, counted positive in the direction of stack growth and expressed in some processor-specific units (usually octets, on octet-addressed processors). Local stack height usually excludes stack space used by parameters for this subprogram; that space is instead counted in the local stack height of the caller, where it is allocated. Vice versa, stack space for parameters that this subprogram provides to its callees is usually included in the local stack height measure. See *stack* and section 4.

Stack, stable

A stack is called *stable* if the final stack height is always zero for this stack, for any subprogram. In other words, the stack pointer for this stack has the same value before and after any call to any subprogram. See *Stack height, final*.

Stack, unstable — A stack is called unstable if it is not a stable stack or, in other words, the final stack height may be non-zero. See *Stack, stable* and *Stack height, final*.

Stack usage — The amount of space that is used (allocated) in a stack, at a particular point in the execution of a program. The *stack usage of a subprogram* (*call*) is the largest amount of space allocated in the stack for this subprogram (call) including space allocated in lower-level callees but excluding space allocated by higher-level callers. Also called *total* stack usage, as opposed to local stack usage which is another word for local stack height, which see. See *stack*.

Static Single Assignment – see *SSA*.

Step — A part (a vertex) of 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 Bound-T.

Storage cell — A part of the target processor that can store a numeric of Boolean value; a register, flag, or memory location. Instructions use the values of storage cells to compute expressions, and may store the computed expressions in the same or other cells, overwriting the old values of those cells. The values stored in cells can determine the flow of execution through conditional or dynamic jumps and calls.

Stub — A subprogram that is not analysed and is instead seen as a "black box" that consumes some execution time, uses some stack space, and may have some effect on storage cells (variables, registers). A subprogram becomes a stub in one of three ways: by an **unused** assertion, by an **omit** assertion, or by asserting bounds on the execution time and/or stack usage so that analysis of the subprogram is not necessary.

Subprogram — A callable (closed) subroutine – a function or a procedure – in the target program.

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 and also the return address. See section 4.

Target (address) — See *Destination*.

Target processor — The processor that will (eventually) run the target program being analysed by Bound-T. Bound-T, however, is run on the *host* computer, which is usually different from the target processor.

Target program — The real-time program that runs (or will run) on the target processor. The execution time or stack usage of the target program are 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 1.3.

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 1.3.

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 then be missing from the analysis and the analysis results may be unsafe, for example the WCET bound may be less than the true WCET.

Unstable stack

See *Stack, unstable*.

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. The Reference Manual 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 *storage 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.

Although the abbreviation WCET in principle means the *true* worst-case execution time, in practice it is often used for an *upper bound* on the true value. For example, the number of cycles given in a *Wcet* output line from Bound-T is usually an upper bound, not the true value.

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.

Virtual function

In object-oriented programming (and particularly in C++), a function (subprogram) that is defined with the same name and parameters for a class and its subclasses (thus *inherited* from the class to the subclasses), but may have a different implementation (different code) for the class and each of the subclasses. A subclass may thus *override* the implementation it inherits from its superclass.

Virtual function call

A call to a *virtual function* (which see) for an object of a dynamically defined class. Thus, while the compiler knows the class hierarchy to which the object belongs (usually identified by the root class), at run-time the object may belong to any class in that hierarchy (any subclass of the root class). At run-time the call must therefore *dispatch* by inspecting the actual class of the object and invoking the corresponding implementation of the virtual function. In C++ this is done by associating a virtual function table with each class. At the machine level a virtual function call becomes some kind of *dynamic* or *indirect call*, which see.

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 Chapter 4. There can be several different call-paths that consume the same maximal amount of stack space. For programs that use more than one stack each stack may have a different worst-case stack path or paths.

Tidorum Ltd

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