

Challenges in Calculating the WCET of a Complex On-board Satellite Application

Manuel Rodríguez¹, Nuno Silva¹, João Esteves¹, Luis Henriques¹, Diamantino Costa¹, Niklas Holsti², Kjeld Hjortnaes³

¹Critical Software SA

Banhos Secos EN1, 3040-032 Coimbra, Portugal

{mrodriguez, nsilva, jesteves, lhenriques, dcosta}@criticalsoftware.com

²Space Systems Finland Ltd

Kappelite 6, 02200 Espoo, Finland

holsti@ssf.fi

³ESA/ESTEC

Noordwijk, Netherlands

kjeld.hjortnaes@esa.int

Abstract

Calculating the WCET of mission-critical satellite applications is a challenging issue. The European Space Agency is currently undertaking the CryoSat mission, consisting of a radar altimetry satellite to be launched in 2005. This paper describes the challenges and the first experimental results of calculating the WCET of the Control and Data Management Unit (CDMU) subsystem of the satellite. This subsystem constitutes the central control unit of all the on-board data handling, as well as the attitude and orbit control system of the satellite, and must guarantee predictable behavior.

1. Introduction

CryoSat is the first satellite of the Living Planet Programme that the European Space Agency (ESA) undertakes in the framework of the Earth Explorer Opportunity Missions (see [1]). It is a three-year radar altimetry mission, scheduled for launch in 2004/2005, dedicated to the observation of the polar regions, particularly the variations in the thickness of the Earth's continental ice sheets and marine ice cover. Its primary objective is to study possible Earth's climate variability and trends, and to predict the thinning of arctic ice due to the global warming.

One of the most important on-board software applications of the CryoSat satellite is the Control and Data Management Unit (CDMU). This application constitutes the central control unit for all the on-board data handling (DH) and the attitude and orbit control system (AOCS) of the satellite. Astrium GmbH (<http://www.astrium-space.com>) is the prime contractor for the CryoSat mission, and Critical Software (<http://www.criticalsoftware.com>) is the prime contractor of the Independent Software Verification & Validation activities (ISVV, see [2]), on which the CDMU application is being screened. The ISVV activities encompass a number of static and dynamic analysis techniques (e.g., robustness and stress testing, traceability matrices, code inspections, software failure mode effects

and criticality analysis, schedulability analysis, etc.) that are applied by personnel not involved in the development process of the target product to ensure complete interdependency.

As part of the ISVV activities, schedulability analysis and WCET calculation are also to be performed. The selected tool for the WCET calculation is Bound-T [3], which is based on static code analysis techniques [4]. This paper describes the challenges of calculating the WCET of the CDMU application using this tool.

The structure of the paper is as follows. In Section 2, we present the CDMU onboard software application and the computational model used by the scheduler. The Bound-T tool is described in Section 3. Section 4 provides experimental results and describes the challenges of performing WCET analysis on the CDMU application. Section 5 concludes the paper.

2. The CDMU onboard software application of ESA's CryoSat satellite

The CryoSat's Control and Data Management Unit (CDMU) is the central control unit of all onboard data handling and of the attitude and orbit control system. Data handling functions are mainly constituted of command distribution, telemetry acquisition and timing facilities during all phases of the mission. Furthermore, the CDMU application performs monitoring functions and, depending on detected failures, provides reconfiguration and safe redundancy switchover capabilities.

The scheduler of the CDMU implements a cyclic scheduling policy. In a cyclic scheduling, the tasks are dispatched at predefined intervals. The CDMU scheduler implements a major cycle of 1 second that is divided into 10 minor cycles (called *slots*) of 100 milliseconds each. At each slot, the scheduler dispatches a predefined subset of cyclic tasks according to a task table, which is repeated every major cycle.

The CDMU onboard software consists of 24 cyclic tasks, 12 sporadic tasks and a number of background tasks. The cyclic tasks are allocated periods and offsets multiple of 100 ms (i.e., one slot), and are mainly

responsible for managing telemetry and telecommands, onboard control procedures, housekeeping, and the mission timeline. The sporadic tasks can preempt the cyclic tasks at any time, and are associated both to external events (e.g., the beginning of a slot or the arrival of a telecommand) and to internal error conditions (e.g., single bit-flip error detection). Background tasks are executed during spare time intervals (usually, at the end of a slot), and mainly perform maintenance activities (e.g., memory scrubbing).

The CDMU application is implemented in Ada 95, and the binary code is generated with the XGC Ada compiler. Neither protected objects nor the Ada tasking model are used (all the tasks are implemented in the form of procedures). The target processor used to run the application is the ERC32/SPARC V.7, running at a clock frequency of 24Mhz. Caches, pipelines and branch prediction units are not used.

3. The WCET tool: Bound-T

There are a number of methods developed for the prediction of the WCET of a program. These methods can be grouped into two main categories, namely, *dynamic* methods (e.g., testing and simulation) and *static* methods (e.g., static code analysis). The advantage of static methods is that they do not require input test sets to be defined in order to find the longest execution paths within the program code. Static code analysis [4] has become very popular and has been the object of study of many works in both academy and industry (e.g., see [5]), giving rise to a large variety of methods and supporting tools.

Bound-T [3], from Space Systems Finland (<http://www.ssf.fi>), is the tool selected by Critical/ESA to perform the WCET analysis of the CDMU onboard software application. Bound-T features static code analysis techniques for estimating the WCET of real-time programs based on their executable binary COFF or ELF code [6]. Apart from the ERC32/SPARC V7 processor targeted by the CDMU application, Bound-T also supports the Intel 8051 and the ADSP21020 processor families. Two of the most outstanding features of the tool are its arithmetic and assertion facilities. The arithmetic facility, based on the non-commercial *Omega Calculator* constraint-solving tool [7], caters for automatically bounding counter-type loops. The assertion facility implements an advanced language that allows the user to manually write assertions and bound those loops that cannot be analyzed fully automatically by the tool.

For more information about the features provided by Bound-T, refer to [3].

4. Challenges and experimental results

The main challenges raised by the analysis of the WCET of the CDMU application are related to the complexity of its loops constructions and to the XGC Ada compiler optimizations. As a consequence:

1. These issues lead to situations unexpected by Bound-T, keeping it from analyzing the application in a fully automatic way.
2. The arithmetic facility of Bound-T systematically blocks (i.e., it runs for too long) when trying to automatically bound the loops of the CDMU application. The arithmetic facility cannot thus be used, and all the loops have to be manually bounded, what can be very time consuming.

In the sequel, we first describe the problems encountered during the analysis of the WCET of the CDMU application's tasks. We then provide experimental results that are compared to the WCET estimations made at early development stages of the application.

4.1. Challenges

The analysis of the WCET of the CDMU code raised many problems that were not expected by the Bound-T tool, and that kept it from analyzing the application in a fully automatic way. The main problems encountered are reported in Figure 1.

1. *Use of "sentinels" in the loops:* A loop terminates only when a particular "sentinel" value is found in an array, rather than when the loop-counter reaches a limit value.
2. *Complex parameter-dependent loops:* The maximum number of iterations of a loop depends on the values of the input parameters. These dependencies in the CryoSat CDMU code are clearly too complex or extensive for the arithmetic facility in the current Bound-T.
3. *Complex loop-arithmetic:* Complex conditions, usually based on logical instructions "and" and "bleu" (*branch if less or equal, unsigned*), appearing in the loop-counter computations or in the loop-termination of the assembly code.
4. *Not explicit loops:* Loops appearing in the assembly code but not in the source code, specially caused by the use of array assignment constructs.
5. *Calls to error-handling code:* Ada run-time checks inserted by the compiler, triggering global error-handlers (e.g., exception *Constraint_Error*) that do not return to the call-site.
6. *Irreducible control-flow:* The longest control-flow path of a function cannot be found because of a compiler optimization or a manual coding of assembly routines leading to poorly structured loops.
7. *Recursive calls:* Recursive call sequences impairing the WCET calculation of the complete set of tasks.

Figure 1. Problems encountered

The assertion facility of Bound-T can be used in order to manually bound the unbounded loops and procedures resulting from these problems. Note that the assertions are not inserted in the application code, but are written as separated files interpreted by Bound-T. Let P be an Ada procedure, and N a natural number. Examples of typical assertions are the following:

- “subprogram “P” loop repeats N times; end loop”, which means that the single loop contained by procedure P does not repeat for more than N times.
- “subprogram “P” time N cycles”, which means that the execution time of procedure P is less than or equal to N clock cycles.

Writing assertions is a challenging issue because the user is responsible for finding bounds for the number of iterations of loops and for the execution time of procedures (i.e., value N of the previous examples). In particular, concerning the problems reported in Figure 1, the challenges raised are the following:

- For cases 1 to 4, the challenge consists in being able to understand the behavior of the CDMU application and find the maximum number of iterations of the unbounded loops. Bounds for loops in cases 1 to 3 can be found by inspecting the Ada source code. The effort required for this depends on the complexity of the Ada code related to the loops, and requires understanding of the control flow, data flow, and specific mission requirements. Bounding loops in case 4 (i.e., *not explicit loops*) requires also analyzing the assembly code of the application. It adds an additional complexity, since it might not be feasible to actually isolate the Ada instructions of the source code that lead to such assembly loops. It is worth noting that some of the unbounded loops caused by these problems might be automatically bounded by the arithmetic facility of Bound-T. Since the arithmetic facility cannot be used, these and all the other loops of the application have to be manually bounded.
- For cases 5 to 7, the challenge consists in finding a bound for the execution time of the corresponding procedures (i.e., the error-handlers, the irreducible functions and the recursive functions). Another challenging issue for case 5 (i.e., *calls to error-handling code*) consists in making assumptions about the maximum arrival rate of failures, as long as the behavior of the application in presence of faults is to be considered. In general, finding time bounds for cases 5 to 7 can be assisted by the use of techniques other than static code analysis (e.g., see [8]).

In the sequel, we analyze in more detail cases 4 to 7.

Not explicit loops

In some cases, the XGC Ada compiler generates loops that cannot be identified in the source code of the

application. It usually concerns the compilation of array assignments constructs, as the one presented in Figure 2.

```

...
Array_A(0..Foo) := Array_B(6..Bar);
...

```

Figure 2. Ada array assignment instruction

Indeed, two loops might be generated in the assembly code for the single array assignment of Figure 2. It usually occurs when the compiler cannot guarantee that the memory addresses (or *slice*) allocated to the target array are different from the slice allocated to the source array. In such a case, the compiler first copies the source array into a temporary location, and then copies the temporary location into the target array.

Apart from array assignments, conditional branches (e.g., *if-else* branches) appearing in the assembly code might be arranged by the compiler in a different order as they appear in the Ada sources. Also, the XGC Ada compiler might generate a huge amount of extra conditional branches for constraint verification purposes. This greatly increases the complexity of the resulting assembly code.

As we explained, the existence of not explicit loops constraints the user to inspect and understand the assembly code, so as to find the maximum number of iterations of the unbounded loops and write the corresponding assertions. This can be very time consuming, as it requires analyzing a large amount of assembly code, and taking into account the specific aspects of the ERC32 architecture. Indeed, some zones in the CDMU application code make a great use of arrays assignments. These array assignments usually occur within other loops or inside conditional structures. Since the compiler might rearrange loops in the assembly code, manually finding bounds can thus become even more difficult.

Calls to error-handling code

This problem is related to run-time checks inserted by the compiler, which trigger global error-handlers not returning to the call-site. More precisely, the code generated for some Ada exceptions contains jumps to addresses outside the text segment.

The XGC Ada compiler is the responsible for most of the calls to exception routines appearing in the assembly code. This is for instance the case of the *Constraint_Error* Ada exception, for which the compiler generates a function labeled “`__raise_constraint_error`”, whose implementation is shown in Figure 3.

```

<__raise_constraint_error>:
02096070:    91 d0 20 05    ta 5
02096074:    81 c3 e0 08    retl
02096078:    01 00 00 00    nop

```

Figure 3. Assembly code of the *Constraint_Error* Ada exception

Instruction at line 0x02096070 (*trap 5*) is interpreted by Bound-T as a jump outside the text segment boundaries. The reason is that the information contained in the vector trap is loaded at runtime, and Bound-T cannot keep track of it during static analysis. Static analysis is thus stopped at this point.

As long as the behavior of the application in presence of faults is not to be considered during static analysis, this issue does not represent a significant problem. Indeed, in practice, whenever an exception is raised, the nominal control flow is broken and the CDMU application is restarted. Two workarounds can be envisaged: (i) asserting the execution time of the exception handler with a non significant value (e.g., “subprogram address “02096070” time 0 cycles;”), or (ii) asserting that no calls to the error-handling subprogram are executed (e.g., “all calls to address “020976070” repeat 0 times;”). It allows Bound-T to proceed calculating the WCET of the application.

Irreducible control-flow

We observed that this problem appeared whenever the call graph contains a jump to a mathematical function in a library responsible for calculating the integer division (e.g., calls to *.div*). These functions are implemented in libraries external to the CDMU application.

The solution consists thus in measuring and asserting the execution time of every library routine leading to an irreducible control flow. For instance, concerning the *.div* routine, the following assertion can be used: “subprogram *.div* time *N* cycles;”, where *N* would correspond to the estimated execution time (expressed in cycles) of the function.

Recursive calls

A recursion involving two functions was identified in the code (the depth of this recursion is 2). This inhibits the WCET calculation in almost all the tasks because Bound-T cannot analyze recursion.

A workaround consists in calculating separately the WCET of each function responsible for the recursion. For instance, consider assertion of Figure 4, where “A” and “B” are the functions responsible for the recursion:

```
subprogram "A" time 0 cycles;
subprogram "B" call to "A" time 0 cycles; end
call;
```

Figure 4. Workaround to calculate the WCET of recursive function B

Assertion of Figure 4 allows automatically calculating with Bound-T the WCET of function “B”. A similar assertion can be written to calculate the WCET of function “A”. Therefore, when the recursion depth is known, WCET bounds for the functions involved in the recursion (e.g., “A” and “B”) can be manually computed via similar assertions, omitting thus the recursion.

4.2. Experimental results

Due to the difficulties encountered while calculating the WCET of the CDMU application with the Bound-T tool, we were not able to obtain definite results yet. Indeed, Bound-T is currently being updated, and some modifications are being performed in the compilation options of the CDMU application.

The preliminary results we obtained are based on assumptions about the execution time of some functions of the CDMU application, specially those functions containing not explicit loops (see Section 4.1). These assumptions were not validated yet. Note also that since many of the asserted functions are common to various tasks, inaccuracies in assumptions are propagated among tasks.

As an example, let us present results concerning task *Mil_Bus_Manager* (Table 1). This task manages the bus that connects the different satellite payloads, and is the largest and most complex task of the CDMU application.

Table 1. WCET calculation results

Task	WCET given by Bound-T (ms)	WCET reported in design documents (ms)
<i>Mil_Bus_Manager</i>	15.79	20

Table 1 reports two different values: the WCET estimated by Bound-T, and the WCET reported in the design documents of the CDMU application. The latter value is mainly based on in-service history information of previous missions of similar applications. As shown in Table 1, the WCET given by Bound-T (about 16 ms) was slightly under the value reported in the design documents (20 ms).

5. Conclusion

Calculating the WCET of mission critical satellite applications is a challenging issue. The Control and Data Management Unit (CDMU) application of the ESA’s CryoSat satellite is responsible for all the data exchanged between the satellite and the ground (e.g., telemetry data containing measurements and telecommands containing satellite commands). It is a large and complex satellite application using the XGC Ada compiler and running on an ERC32 architecture-based microprocessor, whose tasks must guarantee predictable worst case execution times.

In the framework of the Independent Software Verification and Validation (ISVV) program promoted by ESA, Critical Software SA is performing (among other testing activities) the WCET analysis of the CDMU application. The tool chosen by ESA for this activity is Bound-T, from Space Systems Finland, based on static code analysis.

The work presented in this paper leads us to state that fully automated tools for analyzing the WCET of large and complex safety critical satellite applications are still not mature enough. Indeed, the various unexpected problems we encountered kept us from accomplishing the WCET calculation of the entire application. These problems were described in detail in the paper: loops in the assembly code not appearing in the source code, loops depending on complex parameters, etc. To overcome these problems, it is necessary (i) estimating loop and time bounds of some parts (usually functions) of the application code by other means (e.g., code inspection of loops, testing techniques, etc.), (ii) writing annotations asserting the bounds of the concerned loops and functions, and (iii) using the annotations as inputs to the WCET tool. Note however that the reported problems can actually be solved by static code analysis techniques, but more mature tools are still needed so as to deal with these problems in an automated way. We also observed that developing complex applications with WCET in mind (e.g., systematically asserting the maximum number of iterations of every loop in the source code) could greatly help automate WCET analysis. Indeed, facilities to automatically bounding loops (e.g., the arithmetic facility of Bound-T) might be of little use for complex and large applications.

As a conclusion, fully automatic tools and good programming practices are highly required in what concerns static code analysis of the WCET of space applications. Indeed, the effort and resources in industry

must be planned in advance for every activity, and there is little place for unexpected situations that would require revisiting the planning or allocating extra resources.

References

- [1] <http://www.esa.int/export/esaLP/cryosat.html>
- [2] ECSS Secretariat, "ECSS-Q-80B, ECSS Space Product Assurance, Software Product Assurance Draft B", ESA-ESTEC Requirements & Standards Division, Noordwijk, The Netherlands, February 2002 (<http://www.ecss.nl/>)
- [3] <http://www.bound-t.com>
- [4] P. Puschner, C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs", *Real-Time Systems*, vol. 1, pp. 159-176, 1989.
- [5] C.M. Bailey, A. Burns, A.J. Wellings, C.H. Forsyth, "A Performance Analysis of a Hard Real-Time System", *Control Eng. Practice*, Vol. #, No. 4, pp. 447-464, 1995 (<http://citeseer.nj.nec.com/burns93olympus.html>).
- [6] www.pldworld.com/hdl/1/estec.esa.nl/ftp/pub/ws/wsd/erc32/doc/gcc.pdf
- [7] <http://www.cs.umd.edu/projects/omega/omega.html>
- [8] M. Lindgren, H. Hansson, H. Thane, "Using Measurements to Derive the Worst-Case Execution Time", in *Proc. of RTAS 2000*, Cheju Island, South Korea, 2000 (<http://citeseer.nj.nec.com/lindgren00using.html>).