APPLYING WCET ANALYSIS AT ARCHITECTURAL LEVEL

Olivier GILLES, Jérôme HUGUES[1]

Abstract

Real-Time embedded systems must enforce strict timing constraints. In this context, achieving precise Worst Case Execution Time is a prerequisite to apply scheduling analysis and verify system viability. WCET analysis is usually a complex and time-consuming activity. It becomes increasingly complex when one also considers code generation strategies from high-level models.

In this paper, we present an experiment made on the coupling of the WCET analysis tool Bound-T and our AADL to code generator OCARINA. *We list the different steps to successfully apply WCET analysis directly from model, to limit user intervention.*

1. Introduction

Distributed Real-time and Embedded (DRE) systems must enforce strict timing constraints. Run-time mechanisms exist to control the execution time of each processing thread, and eventually detect overrun, like Ada 2005 execution time timers. Yet, these techniques are usually resource consuming, and not sufficient when building critical systems [11].

A better option for resource-constrained or critical systems is to rely on precise WCET analysis techniques. These techniques rely on the careful examination of the source code and/or assembly-level code to extract longest execution path. However, they are hard to master, and time-consuming. Yet, one needs evaluation of the WCET of some functions early in the definition of the software system.

At the same time, model-based development, the idea that a system can be described in a high-level formalism and then leads to fully generated systems emerge. We claim that this approach is interesting provided that the modeling process fully integrates engineering concerns for real-time systems, including performance analysis.

In this paper, we present a process for evaluating DRE systems WCET metrics relying on both architectural model and binary code analysis. In the second section, we present a taxonomy of relevant information for evaluating a system's performance and how one can model the whole system and its properties. In the third section, we present a full process to analyse models performance in an integrated framework. In the last section, we provide a use case, showing how to achieve full system evaluation and conclude on the needs for more advanced analysis techniques.

[1] GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
olivier.gilles@enst.fr, jerome.hugues@enst.fr
This work is funded in part by the Flex-eWare and MOSIC projects

2. Defining a toolchain for building critical real-time systems

In this section we list contextual information on our research work, prior to illustrating the benefits of automating WCET analysis in an automated toolchain.

2.1. A taxonomy of performance criteria

DRE systems must comply to multiple and contradictory constraints. In this section, we present a taxonomy of those constraints and then classify them relatively to their analyzability.

Deterministic behavior : Ensuring a fully deterministic behavior is a complex issue. As a general rule, one should select a computational model that is amenable to full analysis. This usually implies restricting the set of constructs allowed at either model or programming levels.

Schedulability : Once deterministic behavior is achieved, one can apply scheduling analysis on the set of tasks using scheduling-related data (eg. periodicity, deadline, execution time) of each thread in the software model. Furthermore, interferences of critical section accesses must be analyzed too, and thus expressed in the model.

WCET : this is the key factor for determining schedulabity of a system. Worst-Case Execution Time provides a hint on the duration of some computations. A wrong estimate leads either to pessimistic usage of resource, or some runtime errors. While direct analysis of the binary code can be used in order to extract some values (such as a subprogram WCET), more complete evaluations will need to capture the system semantics. Architectural-level relations, in particular, must be captured in order to make scheduling and latency analysis easier. (cf. [6] and [12]). Still, the analysis of the WCET should be also compatible with both the modeled system, and the code that will actually execute it.

Therefore, we claim that these constraints are better expressed at model-level, and then enforced in an automatic code generation and then analysis process process. Code generation process would enforce deterministic behavior and also ensure generated code is amenable to WCET analysis at object or source code level. We selected the AADL, an architectural description language, to describe framework-level and program-level properties. We then present the code generator we developed, and show how to combine it with the WCET analysis tool Bound-T [7].

2.2. AADL

AADL (*Architecture Analysis and Description Language*) [8] aims at describing DRE systems by assembling components. AADL allows for the description of software and hardware parts. It focuses on the definition of interfaces, and separates the implementations from these interfaces.

An AADL description is made of *components*. The AADL standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`) and execution platform components (`memory`, `bus`, `processor`, `device`) and hybrid components (`system`). Components describe well identified elements of the actual architecture. *Subprograms* model procedures as in C or Ada. *Threads* model the active part of an application (such as POSIX threads). AADL threads may have multiple operational modes. Each mode may describe a different behavior and property values for the thread. *Processes* are memory spaces that contain the *threads*. *Processors* model micro-processors and a minimal operating system (mainly a scheduler). *Memories* model hard disks, RAMs, *buses* model all kinds of networks, wires, *devices* model sensors, etc.

An AADL model also describes non-functional facets: embedded or real-time characteristics of the components (execution time, memory footprint...), behavioral descriptions, etc. Description can be extended either through new property sets, or through annexes. Annexes are extensions to the core language. A complete introduction to the AADL can be found in [2].

We have developed the OCARINA [13] tool-suite to manipulate AADL models. OCARINA proposes AADL model manipulation based on a compiler-like API. "Back-end" modules can generate formal models, perform scheduling analysis and generate distributed high-integrity applications in Ada. Generated code relies on the POLYORB-HI middleware to ensure communications and task allocation. POLYORB-HI ensures that a minimal and reliable middleware is generated for a given distributed application. Furthermore, it relies on strong design patterns to ensure the code is compatible with the requirements from the High-Integrity domain, easing further analysis. For a complete description of the code generation strategy enforced by POLYORB-HI, please refer to [3].

2.3. Setting framework properties

As explained above, AADL not only offers a language to describe architectural relations, but also allows to define run-time oriented properties such as a subprogram WCET, the worst-case duration of a context switch, etc. However, the user must provide those values, which is definitively not a trivial task, since such values are highly architecture-dependent and often non-deterministic.

To our knowledge, no reliable method exists to compute such values from high-level architectural level. In order to obtain those values, we proceeded to an analysis of the binary code generated from the architectural model by OCARINA, using tools such as Bound-T, which allows to extract a local subprogram upper bound on the WCET in terms of processor cycles.

3. Evaluation tool suite

A design-level software model cannot directly compute all of system properties, as some of them are highly OS-specific, hardware-specific or compiler-specific. In particular, subprogram WCET or code size cannot be guessed reliably at design level. In order to perform a realistic evaluation, one should first generate the exact binary code for the system, and then perform analysis on the code and then using together the software architecture and the binary code metrics to refine the initial model. In this section, we present the structure of the tool suite, and then we show a use case with a simple example.

3.1. The evaluation pipeline

As illustrated in figure 1, we divided the evaluation into 3 stages : Code generation, model annotation and model evaluation.

1. Code generation

   This stage takes as input the native (ie. user-designed) architectural model plus the legacy code, and build the binary code. Code generation relies on OCARINA and POLYORB-HI [14], which is a middleware generator for high-integrity environment. Of course, the same tools must be used for the final binary code generation to preserve computed properties.

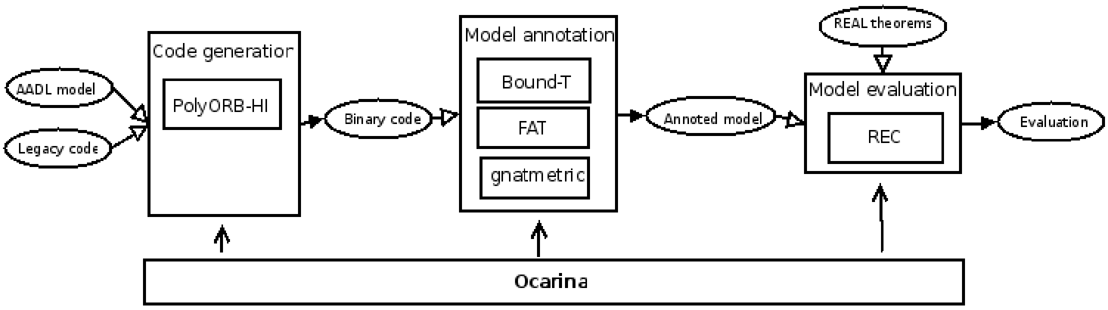

Figure 1. Evaluation tool suite process

2. Model annotation

   This stage takes as input the binary code and the native architectural model, and build an annotated architectural model which takes account of code-specific properties. It relies on tools such as : gnatstack or Bound-T for computing an upper-bound on thread's stack size; Bound-T for extraction of the subprograms' WCET.

   Finally, we use the AADL model manipulation capacities of OCARINA in order to build the annotated model. In case of external tools (Bound-T, gnatstack), glue code has been developed in order to set the tool parameters and extract the results.

3. Model evaluation

   Model evaluation is performed via third-party tools like Cheddar [9] that perform schedulatiblity analysis on a complete AADL model; OSATE [10] for computing latency in the system or performing processor assignment based on thread requirements for CPU usage.

In the following, we illustrate how the toolchain effectively proceeds to evaluate precisely the WCET, and why automation is interesting in that setup.

4. Use case

As an illustration, we selected the `SunSeeker` AADL model, from the OCARINA distribution[2]. `SunSeeker` models a rocket whose goal is to be directed to the sun. It exhibits a traditional subsystem in charge of Guidance, Navigation and Control. This system is made of periodic threads exchanging information on the systems, and commands to be sent to actuators.

This system is amenable to RMA analysis, but requires precise WCET analysis to do so. Besides, even if the overall architecture is already known, the actual computation functions may change as the system evolve. Automating WCET analysis would remove such a time consuming operation, while allowing for early detection of errors when dimensioning the system. In the following, we show how combining stringent code generation strategies and tool can help solving this issue.

4.1. Code generation

Sunseeker relies on AADL semantics for concurrent execution. This model exhibits some periodic execution, on the same processor. Data exchange occurs through Ada protected data component.

[2] See `http://aadl.enst.fr/polyorb-hi/examples.html` for more details

AADL strong semantics allow us to precisely derive supporting runtime entities. OCARINA uses all information in the model to generate minimal code that will support its execution.

Furthermore, in order to be amenable to analysis, the initial model and the code patterns used are both compatible with the Ravenscar profile [1], but also restricts all usage of dynamicity at source-code level (no pointers to procedure, no object orientation, etc.). This ensures the code is deterministic and amenable to analysis. The code to be analyzed encompasses user specific code, execution glue code generate (e.g. threads deduce from application needs), and the supporting kernel and driver. This code, and its compiled counterpart are not enough to allow for WCET analysis.

4.2. Performing WCET analysis

To ensure that WCET analysis can be performed, we must ensure the supporting tools has enough information to proceed. Bound-T inspects object files produced after compilation to evaluate the WCET of a set of function. This analysis relies on a performance model of the processor. We retained the ERC32 processor model, a derivative of the SPARC processor used by ESA. The tool computes all execution path in the system, and return the time to traverse the longest one, if it converges. The latter is the hard point of the analysis.

The code generated by OCARINA strictly follows requirements for high-integrity domain and use the corresponding design patterns for some task artifacts (periodic, sporadic activation patterns, inter-ask communication, . . .). In this setting, Bound-T offers a Hard Real-Time mode that can carefully analyse these patterns. In this context, Bound-T requires the name of the root subprograms on which it must perform WCET analysis: it is the name of the subprogram called at each dispatch time.

Moreover, some of the subprograms used by either legacy code or generated code can be unbounded in term of WCET. To try to analyze them would impede complete WCET analysis. Bound-T allows to specify an *assertion* file that contains the list of subprograms that must not be analyzed, either by providing a well-known WCET (e.g. upper bound for printing a data) or by simply forbidding their analysis (e.g. exception handler). Since generated code follows regular patterns which can be deduced from the AADL model, we can predict which subprograms will ultimately have an unbounded WCET in the binaries. Thus, we generate the assertion file along with the root subprograms from the information found AADL model and the code patterns used by the code generator.

Unbounded WCET occurs when there exist loops in the source code whose upper-bound depends on interactions with other software or hardware elements. We first ensured no portion of generated code has unbounded WCET time per Bound-T analysis. Yet, a typical example is a device driver which completes its work after receiving a signal for the device. We set its value to zero at this level, we tackle this case in a further analysis step.

4.3. Building the annotated model

Once Bound-T completes, we annotate back the AADL model with the information computed by the tool. In hard real-time mode, Bound-T returns an *Execution Skeleton File* (ESF) that stores the result of the analysis. We parse the file to fill the WCET value of each thread. First, we deduce the corresponding AADL thread from the name of its root function's name; then, we compute the exact WCET using the cycles information provided by Bound-T and the the actual processor speed.

Unbounded code cannot be analyzed through Bound-T, we noted it can be either device driver code or user-provided code. The latter is under the responsibility of the designer. The sooner can be accommodated thanks to communication patterns. All task patterns follow a “read-execute-write” cycle. Therefore, we know when a call to the device driver is made. Furthermore, we can deduce from the thread’ interface the size of the data to be sent and add it to the WCET of a thread thanks to device metrics information (latency, bandwidth). This ensures one can provide a complete WCET analysis of all the code generated plus user code.

Then, we complete the AADL model with the newly-acquired WCET in the corresponding threads using the `Compute_Execution_Time` property, as seen in code example 1. In this context, we show how Bound-T and Ocarina can be linked in an efficient way. Let us note the process is limited by Bound-T scalability and the complexity of the generated code. Our experiment illustrates it performs correctly in HRT mode thanks to the careful patterns used for inter-task communications.

```
thread implementation Plant_Type.Plant
calls
    plant: subprogram Sunseekerplant_Subprogram.Beacon;
connections
  parameter Controllerinput -> plant.Controllerinput;
  parameter plant.Outputfeedback -> Outputfeedback;
properties
  Dispatch_Protocol  => Periodic;
  Period             => 10 Ms;
  Compute_Execution_Time => 120 Us .. 120 Us; -- Computed by Bound-T
end Plant_Type.Plant;

thread implementation Controller_Type.Controller
calls
  ctrl: subprogram Sunseekercontroller_Subprogram.Impl;
connections
  parameter Outputfeedback -> ctrl.Outputfeedback;
  parameter ctrl.Controllerinput -> Controllerinput;
properties
  Dispatch_Protocol => Periodic;
  Period            => 10 Ms;
  Compute_Execution_Time => 15 Us .. 15 Us;  -- Computed by Bound-T
end Controller_Type.Controller;
```

Listing 1. Sunseeker Threads Implementation

4.4. Completing evaluation

In order to ensure real-time constraints are met, a system schedule must be established. In the context of preemptive tasks, Rate Monotonic Analysis is a quite common solution. Others scheduler such as Earliest Deadline First (EDF), while less common, offer a more optimal usage of the CPU cycles.

An efficient solution in order to verify schedulability is to use Cheddar, a real time scheduling tool which supports popular scheduling methods such as RMA or EDF and takes as inputs AADL files. In order to perform the analysis, Cheddar will need 2 declared properties in the AADL files: `Cheddar_Properties::Fixed_Priority`, which defines the priority assigned to the task; `Compute_Execution_Time`, which defines the WCET of the thread root subprogram, as computed by Bound-T and the analysis of the communication pattern. To complete some analysis, one may also to define other time-impacting non-functional properties such as `Period`, `Deadline` or `Dispatch_Protocol`. Now that the thread root program WCET has been computed in the annotation phase, Cheddar can perform scheduling analysis and conclude to system schedulability.

Another solution in order to analyze the scheduling is to perform a Response Time Analysis (RTA). While RMA indicates whether the tasks can meet their deadline or not, RTA gives a value for the total task WCET, including perturbating ones such as context switch due to preemption, blocking, clock signal, etc. [4]. The worst-case response time of a process is defined by the time it takes for that process to complete its most demanding set of activities in response to a single activation event occurring under maximum contention from the rest of the system. The worst-case response time of a given activity (ie. a task) can be computed by a recurrence on the sum of three components, which are: the worst-case execution time of the task; the interference incurred by the task; the blocking experienced by the task.

Authors in [12] give a complete definition of each term involved in Response Time Analysis (RTA), and provides formal expressions in order to compute them within a Ravenscar-compliant model. An important thing about these expressions is that they involve the knowledge of the subprogram WCET and the worst-case time of some framework primitives or actions such as a context switch or an clock interrupt handler duration. The run-time environment we used is based on the same mechanisms as the one developed in this analysis. We can then reuse these equations and validate the system. In this context, knowing precisely when the different runtime primitives are used is relevant. We have developed REAL[3], a domain-specific language that allows one to perform queries and computations on AADL models. We implemented a set of REAL subprograms which allows to proceed to full RTA by taking into account the different impacting factors of an RTA analysis. Similar extensions can be written, to take into account other platform-specific or domain-specific analysis.

4.5. Assessment and Related work

The complete process proved to be efficient to analyse directly model. This is mostly due to the fact that Bound-T and the Ocarina code generator relies on the same family of patterns. Writing the annotations for Bound-T is therefor natural.

The complete automation leads to a shortened development time: developer needs only to focus on a high-level design representing its system, joint work by Ocarina, Cheddar and Bound-T allows the designer to validate its system directly. Yet, the validation is done atomically for a complete system. Incremental validation on subparts is a current work in progress.

The combination of a modeling toolsuite and WCET analysis tools is an interesting feature for the system designer: he can test various configurations automatically. This has been already tested in different settings.

Authors in [5] discuss the integration of a WCET tool in Matlab/Simulink. The approach developed is notionally equivalent. The key difference resides in the code to be analyzed. In the case of Matlab, the code is highly sequential whereas our code generator introduces Ada concurrent entities (threads and protected objects). In both cases annotations are built from an a priori knowledge of the code generated. Our approach exploits a higher level description language, AADL, that can integrate multiple languages to implement its functional blocks (e.g. SCADE, C or Ada). We extend this work to a more complete engineering framework.

[3] REAL sources and documentation can be accessed from http://aadl.enst.fr/ocarina/real.html

5. Conclusion

In this paper, we discussed the issue of performing WCET analysis for complex systems. We noted that such analysis is required, yet it is time consuming. We linked this activity to schedulability analysis, usually performed on a high-level model of a system.

In this context, we proposed to use the same model to 1/ generate code, and 2/ perform WCET analysis on the code generated to 3/ refine the model with precise WCET values for the system. We proposed a complete process based on the AADL modeling notation and the AADL toolsuite we developed. AADL defines precise semantics for all its constructs, this allows one to derive precise code and provide roots for analysis.

Typical WCET analysis tools require "hints" to point the code executed by the processing threads, the potential infinite loops of these threads, path irrelevant to the analysis, etc. By computing these information as part of the code generated, and then passing it to the Bound-T analysis tool, we shorten the distance between model and executable system, allowing for precise analysis as the model evolves.

This provides one step forward a complete toolchain to build critical real-time systems from high-level models, combining precise descriptions of the system resources, code generation with high-integrity restrictions enforced, precise WCET analysis, and the capability of performing schedulability analysis or performance evaluation in a uniform framework.

Future work will consider the extension of this toolsuite to exploit all computed information to perform model-based optimizations of the system, e.g. computing precisely the minimal number of threads to provide an semantically-equivalent system, reducing latency, etc.

References

[1] B. Dobbing, A. Burns, and T. Vardanega. Guide for the use of the of the Ravenscar Profile in High Integrity Systems. Technical report, 2003.

[2] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, CMU, 2006. CMU/SEI-2006-TN-011.

[3] J. Hugues, B. Zalila, and L. Pautet. Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina. In *Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07)*, pages 106–112, Porto Alegre, Brazil, May 2007. IEEE Computer Society Press.

[4] M. Josef and P. Pandya. Finding response times in real-time systems. *BCS Computer Journal 29(5)*, pages 390–395, 1986.

[5] R. Kirner and P. Puschner. Integrating WCET analysis into a matlab /simulink simulation model. In *Proceedings ot the 16th IFAC Workshop on Distributed Computer Control Systems, Sydney, Australia, School of Computer Science and Engineering, UNSW.*, November 2000.

[6] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in hard-real-time environment. In *Journal of the ACM*, january 1973.

[7] Tidorum Ltd. Bound-T Execution Time Analyzer, url: `http://www.bound-t.com`.

[8] SAE. Architecture Analysis & Design Language(AS5506). sep 2004.

[9] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar : a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, New York, USA, December 2004. ACM Press.

[10] SEI AADL team. Osate : an extensible source aadl tool environment. Technical report, SEI, December 2004.

[11] S. Ureña, J. Pulido, J. Zamorano, and J. A. de la Puente. Handling Temporal Faults in Ada 2005. In Springer Verlag, editor, *Proceedings of the 12th Reliable Software Technologies AdaEurope'07*, 2007.

[12] T. Vardanega, J. Zamorano, and J. A. de la Puente. On the dynamic semantics and the timing behavior of ravenscar kernels. *Real-Time Syst.*, 29:59–89, 2005.

[13] T. Vergnaud and B. Zalila. Ocarina: a Compiler for the AADL, `http://aadl.enst.fr`. Technical report, Télécom Paris.

[14] B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. In *Proceedings of the 11th IEEE International ymposium on Object-oriented Real-time distributed Computing (ISORC'08)*, Orlando, Florida, USA, May 2008. IEEE Computer Society Press.