Toward a fine-grained execution model of real-time tasks

Zineb Boukili
zineb.boukili@univ-brest.fr
Lab-STICC, CNRS, UMR 6285,
Univ. Brest, Brest, France

Hai Nam Tran
hai-nam.tran@univ-brest.fr
Lab-STICC, CNRS, UMR 6285,
Univ. Brest, Brest, France

Alain Plantec
alain.plantec@univ-brest.fr
Lab-STICC, CNRS, UMR 6285,
Univ. Brest, Brest, France

ABSTRACT

Verification of real-time systems in scheduling analysis tools is generally based on the validation of timing constraints by the mean of scheduling simulation or feasibility test. Real-time tasks are modeled with their worst-case execution times and deadlines together with other properties depending on the choice of scheduler. Even though the current task model provides sufficient information to perform basic schedulability tests, it is inadequate to enforce a dynamic control to guarantee the normal operation of a system under hardware/software malfunctions or malicious cyber attacks. In this article, we present an approach to extend the current task model in the Cheddar scheduling analyzer with information of internal executions and data accesses. We demonstrate the applicability of our model by providing a case study as well as its realization in the architecture analysis and design language (AADL).

1 INTRODUCTION

Real-time critical systems (RTCS) are systems in which the correctness of their behavior depends not only on the logical results of computations but also on the physical time when these results are produced. They are qualified as critical because the absence of response is at least as serious as its incorrectness and the failure of such systems has unacceptable consequences for society. Typical examples of real-time systems include flight control systems, networked multimedia systems, command control systems, and real-time monitors.

Verification methods used for RTCS are mostly based on the early validation of timing properties. These methods are qualified as early verification because they are used during the design phase. However, this type of verification is insufficient to guarantee the proper functioning of a system during operation considering hardware/software malfunctions or malicious attacks.

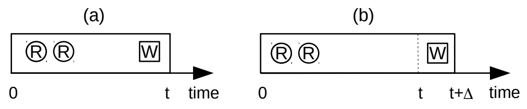

Figure 1: Invalid behavior w.r.t data accesses

Figure 1 depicts such invalid behavior, on the left (a), a valid task, and on the right (b), the same task but with a delayed write data access. On the left side, the valid behavior of the task consists in two successive read accesses and one write access (a). An application malfunction or an intrusion attack can lead to an invalid behavior scenario as shown on the right side (b) where the writing operation has exceeded the expected time with a delta delay.

In our work, we focus on the detection of timing deviations at run-time by monitoring data accesses. Our approach bases on a fine-grained execution model of real-time tasks then check dynamically all data accesses according to declared timing constraints. This article presents the first step that extends the task model to take into account these elements in a scheduling analysis tool.

Problem statement: Real-time task models in scheduling analysis tools such as [6, 14, 16] do not allow the specification of data accesses and their related timing constraints. Scheduling analysis by the mean of simulation in these tools stops at the task level and does not give insights of internal task executions. Thus, scheduling analysis does not provide meaningful data for run-time monitoring as we expect to monitor data accesses and detect timing deviations.

Contribution: In this article, we present a real-time task model that takes into account data access by decomposing tasks into smaller units called *execution units*. We show how to extend the task model exists in the Cheddar scheduling analyzer with our proposals. In addition, we explain how to integrate our model in the architecture analysis and design language AADL[7]. Experiments are conducted with the research open-source avionics and control engineering (ROSACE) [12] case study and the OTAWA [2] static analyzer to show how data can be acquired for our model.

The rest of the article is organized as follows. Section 2 presents the background and positions our work. Section 3 presents our modeling approach. Section 4 explains our experimentation method and presents the results on the ROSACE case study. Finally, section 5 concludes the article and discusses future work.

2 BACKGROUND & RELATED WORK

In this section, we provide a brief summary of task models used in scheduling analysis tools. Then, we present an overview of the existing approaches focus on decomposing task execution into smaller units or phases.

2.1 Task model in scheduling analysis tools

The real-time task model presented in the seminal work of [10] has been widely implemented in real-time scheduling analysis tools such as [14], [6], [16], [4]. A task is characterized by two attributes: capacity and deadline. The capacity represents the worst-case execution time (WCET) of the task. The task must complete before a deadline, which can be either relative or absolute. Depending on the choice of scheduler, a task can have other attributes such as priority, period, and release time.

Several extensions of the basic task model have been implemented in the tools presented above to account for new elements such as cache-related preemption delay [15], [6], energy consumption [5], and security [1]. Nevertheless, the capacity remains the only attribute representing the execution of a task. In the next section, we present the approaches extending the execution model.

2.2 Modeling task execution

The idea of decomposing tasks executions into smaller units or phases has been proposed in [3, 11, 13]. In [13], the authors present the predictable execution model (PREM) [13] which co-schedules at a high level all active COTS (Commercial-Off-The-Shelf) components in the system, such as CPU cores, and I/O peripherals to permit predictable, system-wide execution. When a typical real-time task is executed on a COTS CPU, cache misses are unpredictable. This issue makes it difficult to avoid low-level contention for access to main memory. PREM aims to solve the problem of memory interference between peripherals and CPU tasks. The code for each task is divided into a set of N scheduling intervals classified into compatible and predictable intervals. First, each predictable interval is divided into two different phases. During the initial memory phase, the CPU accesses main memory to perform a set of cache line fetches and replacements. At the end of the memory phase, all cache lines required during the predictable interval are available in last-level cache. Second, the second phase is known as the execution phase. During this phase, the task performs useful computation without suffering any last-level cache misses.

Another way to model the execution of a task is the time interest points (TIP) [3] graph. The authors describe TIPsGraphs as an intermediate representation to transform the CFG (Control Flow Graph) representing the control flow of a task into a sequence of time intervals representing the timing aspects of the task execution. Typically, a TIP can be:

- Memory instructions (stores and loads), when the static analysis cannot guarantee that they will always result in AH.
- Memory instructions addressing shared variables, or data residing in a cache block that can be written by another task.
- Instructions for which the static analysis cannot guarantee that they will always result in a hit in the instruction cache.
- Pivot instructions.

In [11], the authors propose the acquisition execution restitution (AER) model. Tasks are divided into three distinct phases, namely acquisition (A) and restitution (R) which are communication phases (i.e., phases in which accesses to the memory are performed) and a single execution phase (E) which is a computation-only phase. The AER model is a generalization of the PREM model which considers a single core environment while the AER model executing in multi-core environments.

The discussed models are designed mainly to improve scheduling analysis and to reduce timing interference in real-time applications. But none of them provide additional information about data accesses that we want to monitor in our approach.

3 APPROACH

In this section, we explain our approach to decompose task executions into smaller units. Our model is implemented in the Cheddar scheduling analyzer, we propose also an implementation for the execution unit in AADL language using the behavioral annex. Cheddar [14] is an open-source real-time scheduling analysis tool which allows users to model software and hardware architectures of real-time systems, and to check their schedulability or other performance criteria.

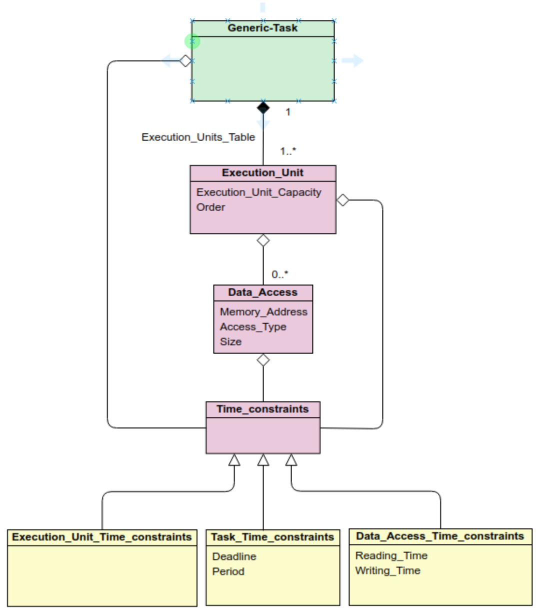

Figure 2: Suggested Task Representation Model

3.1 Cheddar Task Model Update

The original task model in Cheddar stops at the task level represented by the green box in Figure 2 and does not take into account the internal task design because the classic scheduling analysis does not require this information. In Cheddar, a *generic task* is modeled with the attributes allowing scheduling analysis on multi-processor platforms such as: capacity, priority, deadline and core mapping. Depending on the choice of scheduler, a task can have additional attributes including period, offset and release time. Interested readers can refer to the complete description of the hardware and software component supported by Cheddar in [8].

Our solution consists in representing task execution as a set of internal blocks called execution units. Each execution unit represents a portion of the task code. As depicted in Figure 2, each execution unit has a chronological order and a capacity (execution time). Within a period, a task includes a sequence of execution units, namely the compatible interval, the memory phase and the execution phase. Therefore an execution unit owns a Unit_type property that represents its particular phase.

Similarly, an execution unit can be composed of one or more data accesses, each one is characterized by a memory address of the accessed data, by the size of the data and by an access type, i.e. read or write access.

A task, an execution unit or a data access can all be associated with a time constraint specification. Thus, a time constraint is represented as an abstraction that can have three kind of concrete representations: Task_Time_Constraint specifies the Task time constraints with a deadline and a period. Data_access_Time_Constraint specifies data access time constraint with a reading or a writing time. Finally, Execution_Unit_Time_Constraint specifies execution unit kind of time constraint.

```
SCHEMA Execution_Units;
    TYPE Execution_Unit_Type = ENUMERATION
        OF (Compatible_phase,
            Memory_phase,
            Execute_phase,
            Non_identified);
    END_TYPE;
    ENTITY Execution_Unit
        SUBTYPE OF ( Named_Object );
        Unit_type : Execution_Unit_Type;
        Time_Constraint : Time_Constraint;
        Capacity : Natural;
        Order : Natural;
    END_ENTITY; ...
```

Figure 3: Execution Unit Entity

```
SCHEMA Data_Access;
    TYPE Data_Access_Type = ENUMERATION
        OF (Read_Data, Write_Data);
    END_TYPE;
    ENTITY Data_Access
        SUBTYPE OF ( Named_Object );
        Access_Type : Data_Access_Type;
        Memory_Address : Integer;
        Time_Constraint : Time_Constraint;
        Size : Integer;
    END_ENTITY; ...
```

Figure 4: Data Access Entity

To introduce these modifications into the Cheddar tool, we update its meta-model by adding the execution unit and data access entities as shown in the Figures 3 and 4[1].

3.2 Execution Unit Implementation in AADL

Our work includes the architecture analysis and design language (AADL) part, in which we implement the execution unit component using the behavioral annex. It provides a standard sub-language extension to allow behavior specifications to be attached to AADL components. The aim is to refine the implicit behavior specifications that are specified by the core of the language.

In AADL, the subprogram component may represent a portion of the source code, therefore we can implement our execution unit as a subprogram component. Then we describe the internal behavior of the execution unit as a state automaton with transitions, guards and actions as shown in the behavioral annex. To illustrate the idea, we consider an example of the execution unit with 2 data accesses (read and write). As shown, in Figure 5, and according to the behavior annex, the execution unit correspond to a subprogram, inside, we have two states each corresponds to data access operation

[1]These specifications are in EXPRESS (ISO 10303-11 [9]), the data modeling language that is used to implement the Cheddar meta-models

```
SUBPROGRAM Execution_Unit
END Execution_Unit;

SUBPROGRAM IMPLEMENTATION Execution_Unit.impl
annex Behavior_Specification {**
    states
        Read : state;
        Write: state;
    transitions
        Writing_operation:
        Read-[on dispatch]-> Write {
        computation(6ms); -- Access time to write data
        };
        Reading_operation:
        Write-[on dispatch]-> Read {
        computation(3ms); -- Access time to read data
        };
        **};
END Execution_Unit.impl
```

Figure 5: Execution Unit Implementation in AADL

that we may have in an execution unit, therefore the computation instruction is used to define the time constraint of the data access.

4 CASE STUDY: ROSACE

In this section, we describe our evaluation to measure the WCET of tasks and execution units. The evaluation is based on the ROSACE [12] case study using OTAWA [2] tool.

4.1 ROSACE Case Study

ROSACE [12] is a research open-source avionic control engineering case study, that goes from a baseline flight controller, developed in MATLAB /SIMULINK , to a multi-periodic controller executing on a multi/many-core target. In this paper, we focus on the longitudinal flight controller design, his goal is to track accurately altitude, vertical speed and airspeed commands (resp. hc, Vzc and Vac). The controller design is divided into two parts. First, the environment simulation part represents the real system that is to be controlled, that is the aircraft as well as the engines and elevators. Second, the controller part which gathers the control loops (altitude_hold, Vz_control, Va_control) as well as filters. In this paper, we use the ROSACE case study to illustrate how computing WCET for tasks, execution units and also computing the number of data accesses inside each execution unit.

4.2 WCET Task Measurement

As depicted on the controller design in [12] and according to the case study source code, each block of the Simulink controller design represent a task. Therefore, to get the WCET of each block using OTAWA tool, we proceed as follows:

- We have separated the code of each task in a separated file.

Task	WCET (cycle)	First E.U (cycles)	Data access number	Second E.U (cycles)	Data access number
Engine	2185	410	1 read & 1 write	1775	4 read & 2 write
Elevator	7080	75	2 read & 2 write	7005	6 read & 3 write
Aircraft_Dynamics	52160	31280	35 read & 16 write	20880	14 read & 16 write
Vz_Control	3815	2815	9 read & 1 write	1000	2 read & 1 write
Altitude_Hold	3475	515	3 read & 1 write	2960	5 read & 2 write
Va_Control	4430	3125	10 read & 1 write	1305	3 read & 1 write
H_Filtre	5450	4035	8 read & 4 write	1415	3 read & 3 write
Az_Filtre	2665	1250	5 read & 5 write	1415	5 read & 3 write
Vz_Filtre	3875	1720	4 read & 5 write	2155	5 read & 3 write
Q_Filtre	2665	1250	5 read & 5 write	1415	5 read & 3 write
Va_Filtre	5180	5140	9 read & 6 write	40	2 read & 2 write

Table 1: WCET Measurement

- As we have a shared data between tasks, to make computations, it was necessary to set the value of shared parameters for each task.
- We have targeted the ARM architecture and specially the ARM-linux-gnueabi-gcc compiler.
- After the compilation of the task's code, for each task, we have build a flow fact file (*.ff) to identify loop headers using the mkff utility. Called on the executable, it generates a template (*.ff) file where the loop counts are replaced by ? marks.
- Using owcet instruction which allows to use WCET computation scripts dedicated to a specific microprocessor model, we get the WCET of each task of the controller.

4.3 WCET Execution Unit Measurement

Once we have the WCET of tasks, we can go further by calculating the WCET of execution units set within each task of the system. For each we define two execution units, the division is made according to the source code data, the first execution unit processes the input data and the second provides the output data. Then we generate the WCET by following the same OTAWA steps explained above, but for the execution units this time. We also determine the number of data accesses in each execution unit. The Table 1 summarises the results found for each task of the controller system.

5 CONCLUSION

This article presents an approach to extend the classical real-time task model by decomposing task executions into smaller units. We present an approach to extend the current task model in the Cheddar scheduling analyzer with information of internal executions and data accesses. Our model can also be described in AADL. To show that it is possible to acquire data for our proposed model, we experiment with the ROSACE case study and the WCET analysis tool OTAWA.

As a short term perspective, we plan to extend this case study with more relevant measures by investigating different strategies to decompose the tasks into execution units. Future work will also focus on the dynamic control of the operations performed on the data then update the actual simulator in Cheddar to highlight possible deficiencies.

REFERENCES

[1] Ill-ham Atchadam, Laurent Lemarchand, Hai Nam Tran, Frank Singhoff, and Karim Bigou. 2020. When security affects schedulability of TSP systems: trade-offs observed by design space exploration. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. IEEE, 369–376.

[2] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An open toolbox for adaptive WCET analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 35–46.

[3] Thomas Carle and Hugues Cassé. 2018. Reducing timing interferences in real-time applications running on multicore architectures. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. 1–11.

[4] Younès Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet, and Manar Qamhieh. 2012. YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms. In *WATERS 2012*. Italy, 21–26. https://hal-upec-upem.archives-ouvertes.fr/hal-00691985

[5] Younès Chandarli, Manar Qamhieh, Frédéric Fauberteau, and Damien Masson. 2014. *Yartiss: A generic, modular and energy-aware scheduling simulator for real-time multiprocessor systems*. Ph.D. Dissertation. UPE LIGM ESIEE.

[6] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. 2014. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 6–p.

[7] Peter H Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. 2004. An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *IFIP World Computer Congress, TC 2*. Springer, 3–15.

[8] Christian Fotsing, Frank Singhoff, Alain Plantec, Vincent Gaudel, Stéphane Rubini, Shuai Li, Hai Nam Tran, Laurent Lemarchand, Pierre Dissaux, and Jérôme Legrand. 2014. Cheddar architecture description language. *Lab-STICC technical report* (2014).

[9] ISO 10303-11 1994. *STEP Part 11: EXPRESS Language Reference Manual*. ISO 10303-11.

[10] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[11] Cláudio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. 2016. A closer look into the aer model. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 1–8.

[12] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. 2014. The ROSACE case study: From Simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 309–318.

[13] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A predictable execution model for COTS-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 269–279.

[14] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. 2004. Cheddar: a flexible real time scheduling framework. In *Proceedings of the 2004 annual ACM SIGAda international conference on Ada*.

[15] Hai Nam Tran, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. 2016. Cache-aware real-time scheduling simulator: implementation and return of experience. *ACM SIGBED Review* 13, 1 (2016), 22–28.

[16] Richard Urunuela, Anne-Marie Déplanche, and Yvon Trinquet. 2010. Storm a simulation tool for real-time multiprocessor scheduling evaluation. In *15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. IEEE.