CORTEVA

A Correct-by-construction methodology for supporting execution time variability in real-time systems

ANR PRCE – Appel à projets générique 2017
Défi 7, Société de l'information et de la communication

Abstract

In real-time safety critical systems, it is of paramount importance to guarantee that computation is performed within certain time bounds, otherwise a critical failure may happen. Avionics and aerospace systems, electronic automotive systems, train control systems, etc, are all examples of real-time safety-critical systems. To guarantee correctness, the designer needs first to compute bounds on the execution time of every block of code, and then to guarantee that, in the worst-case, every block is scheduled by the operating system to complete before its deadline.

Today, it is difficult to build efficient and predictable real-time systems on modern processors, because the execution time of a piece of code exhibits a large variability. The worst-case can be hundreds of times larger than the best-case, due to dynamically varying parameters such as the state of cache memories for instance. Therefore, the designer needs to greatly over-provision the computational capacity of the processors, leading to a higher cost of the system. The continued demand for additional functionalities makes the situation unsustainable in the long term. While some methods have been proposed to deal with such large variations, they are not immediately applicable because they focus on scheduling without considering the functional aspects of the application.

The overall objective of this project is to contribute to the design and development of the next generation of safety critical embedded real-time systems. In particular, we aim at solving the problem of the large variability of execution times by using sound and provably correct programming models that combine functional and timing aspects.

The main idea can be summarised as follows. First, we will use parametric Worst-Case Execution Time analysis techniques for computing off-line a WCET formula. The formula is parametrised with respect to input values of the code block and to state of the processor cache. Then, we plan to use the formula at run-time to dynamically estimate a tighter upper bound to the execution time. The execution time estimation will be used at run time to dynamically select the application behaviour so as to avoid deadline misses. The designer will specify the behaviour of the system by using a synchronous language to formally guarantee at the same time functional and timing correctness. Finally, we propose to use a design methodology to help the designer configure the system in the best way.

Consortium

Partner	Member	Position	PM	Role
CRIStAL	Giuseppe Lipari	Professor	12.6	Coordinator
	Clément Ballabriga	Assoc. Prof.	9	Participant
	Julien Forget	Assoc. Prof.	9	Participant
ONERA Toulouse	Frédéric Boniol	Researcher	9.3	ONERA Sc. leader
	Luca Santinelli	Researcher	9	Participant
RealTime-at-Work	Loïc Fejoz	Res. Engineer	12	RTaW Sc. leader
	Lionel Havet	Res. Engineer	12	Participant

1 Context, positioning and objectives

1.1 Context

Hard real-time safety critical systems are systems whose correctness depends not only upon the correctness of the produced results, but also on the time at which they are produced. A missed deadline may compromise

the functionality of the system, and in some extreme cases, even cause loss of human lives.

Such critical real-time systems can be found in avionics, aerospace, automotive, public transportation systems, etc. For example, in the control of an aircraft subsystem (e.g. the control of the plane engines), it is of utmost importance that sensor inputs are read periodically with a precise sampling rate, and that the control law is computed and results are sent to the actuators within a certain maximum delay, otherwise some malfunctioning may impair engine functionality and even compromise the aircraft stability.

A complex critical real-time system consists of several concurrent periodic tasks executed on a microprocessor using a Real-Time Operating Systems (RTOS). The RTOS scheduler is in charge of selecting which task to execute at each instant, typically based on the task priority.

Therefore, it is important to perform a *schedulability analysis* of the system: given a set of concurrent *tasks*, with their scheduling parameters (e.g. period and priority), their timing constraints (e.g. deadlines) and their **Worst-Case Execution Times - WCETs**, schedulability analysis can formally verify that all tasks will meet their timing constraints (e.g. deadlines) under all conditions.

1.1.1 Execution time variability

The execution time of a block of sequential code depends on two different classes of parameters. On one hand, the code may contain branches (*if-then-else*) and loops (*do-while*) whose execution depends on the run-time values of variables, which in turn may depend on external inputs or on the internal state of the code. Therefore, one source of variability is due to the structure and the semantics of the algorithm implemented by the code under analysis.

On the other hand, the execution time depends also on the characteristics of the hardware platform on which the code executes. Over the course of the years, progress in chip manufacturing has greatly increased the performances of microprocessors. However, processor designers have focused their efforts on reducing the *Average Execution Time* of a sequential piece of code. Thanks to the use of pipelines and cache memories, performance has increased at an exponential rate, following the well-known Moore law. However, the side effect of this performance increase is the **variability of the execution time**. For instance, [Wil12] reports a slide by AbsInt (a company providing tools for WCET analysis) that shows that the worst-case execution time of a piece of assembly code, consisting of only three instructions, can be hundreds of time higher than the best-case, on modern architectures, due to pipelines and cache memories. However, the "*always-miss* assumption" is very pessimistic and such a situation may occur very rarely during execution.

The first source of variability (program variables) may be analyzed statically and, to a certain extent, even taken under control by the developer. The second source of variability (hardware platform) is more difficult to analyze and control because it depends on hardware mechanisms that are difficult to model and to control, and because it depends on the interference of the other tasks in the system, and on the RTOS scheduler.

Representing the execution time of a task as a single value (the WCET) is certainly pessimistic. At the same time, for safety reasons it is necessary to cater for the worst-case. As technology progresses, the situation is getting worse: the introduction of multicore processors has further widened the gap between BCET and WCET. A task executing on one core may now suffer interferences from the other tasks executing on the other cores due to shared caches and shared buses. This is one of the reasons why multicore processors have not yet been widely adopted for the development of safety critical real-time systems. The high variability of the execution times forces developers to reserve resources for the worst-case, thus greatly under-utilizing the computational resources of the processor.

One attractive idea for reducing execution time variability is to build a *predictable processor* adapted to the requirements of safety critical real-time systems. One example of such architecture is the Precision Time (PRET) machine [EL07], proposed by the CHESS group at the University of Berkeley. The PRET machine avoids the use of caches and uses scratchpad memory instead; it reduces contention on the bus memory by using appropriate hardware scheduling, etc. However this approach is currently unfeasible from an economic point of view: the high costs of development and production of a special dedicated processor (and of all the development chain on top of it) will not be covered by the low production volumes. The PRET project ended in 2014 without being taken over by industry.

1.1.2 Impact of timing variability on functional correctness

The difficulty in proposing solutions at the hardware level has convinced academic researchers to investigate techniques for **coping with execution time variability** at the level of the RTOS scheduler. However, most research on this topic focuses on timing analysis (execution time estimation and scheduling problem), without considering the **functional semantics** of the tasks. In other words, a task is usually modelled as a set of numerical parameters, without regards to the semantics of the executing code. For example, the popular Mixed Criticality (MC) scheduling approach [BD13] assumes that low criticality tasks can be *dropped* when a high criticality task exceeds its initial execution time budget. However, a task that is dropped before completion must not produce side-effects, otherwise the functional correctness of the system may be compromised.

From a software point-of-view, handling WCET overrun is a complex problem [BW09]. First, execution-time monitoring mechanisms must be provided, either by the programming language or by the operating system. Guaranteeing the accuracy of these mechanisms when preemptions occur can be difficult. Second, an error-recovery procedure that will be executed in case of overrun must be provided. There are many possible ways to implement the recovery. In the simplest case, we may afford to let the task continue its execution (if the overrun does not lead to a deadline-miss). Assessing at run-time whether this is possible can be complex though. In other cases we may have to stop the current task execution. If the task is stopped, we may have to undo its side-effects or start a recovery task. The recovery procedure can also require a more general reconfiguration of the system, creating new tasks or altering existing ones. Finally, we must plan for means to transit back to the system initial behaviour. This complex process is mostly ignored by existing approaches in the real-time research community, since system models generally abstract from the system functional behaviour. By contrast, our project focuses precisely on this aspect.

1.2 Objectives

The overall goal of the project is to *safely* and *efficiently* deal with variable execution times in safety critical systems. We propose to do so in a **precise, semantically sound and efficient** manner by combining existing techniques with new ones and by considering this problem from a software programming point of view. Existing programming techniques usually focus on recovering from a deadline-miss when it occurs. We propose a different approach, which aims at preventing the occurrence of a deadline-miss, by dynamically adapting the system behaviour when it detects an upcoming WCET overrun. This will be achieved through three sub-objectives:

Objective 1 - Parametric WCET: We will rely on a parametric WCET technique to dynamically predict the occurrence of an overrun. Instead of producing a single WCET value for a task, a parametric WCET approach produces a formula, which depends on a set of parameters, such as the cache state or the input values of the task. At run-time, more precisely at each task release, we can obtain the actual values of these parameters and thus instantiate the WCET formula to obtain an estimate of the WCET that is tighter than the result of a classic WCET analysis performed completely off-line;

Objective 2 - Programming constructs for deadline-miss prevention: Thanks to parametric WCET, we can dynamically forecast an overrun before it occurs and modify the system behaviour to either prevent the overrun from occurring or to prevent it from causing a deadline-miss. We will define programming language features that enable a sound functional specification of the behaviour of the application when an upcoming execution-time overrun is detected. These features will be introduced in synchronous data-flow languages, which have been proven well-adapted for the programming of safety critical systems;

Objective 3 - Iterative design and analysis methodology: After the system has been designed, an analysis must be carried out to guarantee that no deadline will be missed (schedulability analysis). If the response is negative, the designer must go back and modify the system to avoid the deadline miss. However, the high number of parameters, and the variability in the execution times make it difficult for the designer to chose the most appropriate set of parameters to configure the system. Therefore, in this project we will propose an iterative design and analysis methodology, supported by a software analysis tool, that integrates the previous techniques. The tool will guide the designer in the choice of the scheduling parameters and in the re-design if the system happens to be un-schedulable.

1.3 Originality

1.3.1 State of the art

This project consists of two main tasks: (1) analyzing the execution time variability of real-time tasks, and (2) exploiting the results of this analysis. Therefore, we will present various works related to the execution time variability analysis. Then, we will discuss papers related to the support and exploitation of these results in real-time operating systems.

Analyzing execution time variability Execution time variability in real-time applications can be caused by hardware effects (such as instruction or data cache), by the content of the program (dependency on arguments or input), or by system-related effects (such as preemption, system calls, etc.). In this section we present various solutions proposed in the past to address these sources of variability.

The main cause of hardware-related execution time variability is the cache. F. Nemer et al. [Nem+07] attempt to address this problem by studying the inter-task cache effects in real-time systems, for example the impact on WCET of cache block sharing between tasks. This approach works by statically analyzing the task code and the schedule. Therefore, it is limited to off-line scheduling. Moreover, the presented analysis suffers from great pessimism.

A different but related attempt to handle this problem is presented by C. Ballabriga et al. in [BCS08]. The method consists in statically computing the *cache damage*, a safe abstraction representing the effect of a task execution on the state of the cache. This cache damage enables to get an estimate of what the cache state will be after task execution, provided an approximation of the cache state before task execution.

Loop bounds are another significant cause of execution time variability. They are often difficult to statically determine, which produces execution time estimates with large pessimism. An attempt to address this problem has been made by S. Mohan et al. in [Moh+05]. This paper describes a method to take into account the variability of the WCET of loops in real-time systems to improve the energy saving (by using methods such as DVFS). Unfortunately, the timing model of the loops is quite simple, and does not take into account important effects such as the cache, the different possible paths in the loop body, etc.

These papers address several sources of execution time variability. Unfortunately they do not propose a full framework capable of integrating all these parameters into a single analysis. We propose such a general framework in [BFL15; BFL17], where we present an approach to parametrically compute the WCET of an application, producing a WCET formula that depends on arbitrary parameters. The parameters can, for instance, represent loop bounds or cache content. The WCET formula can then quickly be evaluated at run-time to produce a tight WCET estimate.

Supporting execution time variability Several methods have been proposed to deal with large variability between worst-case and average case execution times. First, more precise task execution time models that take variability into account have been proposed. For instance, the probabilistic approach [Cuc+12; MNS13] represents execution times using probability distribution functions. The parametric WCET approach [BL08; CB02] represents execution times as formulas that depend on parameters, such as task input values or hardware cache content.

Mixed Criticality (MC) Scheduling [Ves07] has been proposed as a way to deal with execution time variability, focusing on the *Mixed Criticality Scheduling Problem*, i.e. how to schedule a set of tasks at different levels of criticality so that high criticality tasks are always guaranteed. The main idea behind this technique is to change the **criticality mode** of the system upon detection of an extreme value of the execution time, and then execute only the high criticality tasks. The MC Scheduling Problem has received much attention from the research community, as testified by the large number of research papers and also by the number of funded research projects which deal with such a problem [BD13].

In the context of real-time scheduling, sensitivity analysis [GC11; BDB08] aims at determining acceptable deviations from the specifications of a problem. It consists in studying the consequences, in terms of deadline-misses, of deviations from the specified task real-time characteristics (WCET, deadline, period). For instance, it is possible to determine up to which level an estimated WCET can be exceeded at run-time, while still respecting all system deadlines. This is an important topic for our project since it can help us determine under which conditions run-time error-prevention measures must be considered.

Reservation-based systems [Raj+98; AB98] and hierarchical scheduling [LB05] have been proposed as a way to provide temporal isolation in real-time systems. Temporal isolation prevents a task that exceeds its WCET from impacting other tasks of the system by suspending it and delaying its completion. A careful system design combined with such techniques can be used to limit the impact of exceeding some task WCET. For instance, we can ensure that there is no impact on the critical functionalities of the system. However, the task that exceeds the WCET is delayed and may miss its deadline. In other words, while providing isolation we cannot entirely avoid deadline misses.

As mentioned in [BW09], a possibility to handle WCET overrun is to divide the WCET of a task in two parts: the first part corresponds to the execution time of the task under normal conditions, the second part corresponds to the time dedicated to the execution of error recovery mechanisms in case the task exceeds its normal execution time. The main problem of this approach is that it can introduce important pessimism in the schedulability analysis.

Since timing-faults may not only impact the task that caused the fault but also other tasks, real-time programmers usually rely on low-level asynchronous mechanisms to handle them. This includes classic POSIX event handling mechanisms, which can be used to plan a recovery procedure, should a WCET overrun occur. More complex asynchronous transfer of control mechanisms are available in Java [**burns2007Java**], Ada [BW07] or Real-time Euclid [KS86], which enable the programmer to specify the behaviour of a function in case it gets interrupted due to an overrun occurring in a different function.

Synchronous languages [Ben+03] have successfully been applied to the implementation of real-time systems for many years now. Thanks to a high-level of abstraction and to formally defined semantics, programs written with such languages are easier to analyze and to prove correct. Furthermore, their compilation is also defined formally, which ensures that properties proved on the synchronous program are preserved in the generated code (*correct-by-construction* programming). However, execution time variability is not a well-covered topic in this area of research, since synchronous languages abstract from execution-times and mostly ignore them. A notable exception is the introduction of *Futures* in Lustre [CGP12], which enables to use computations with unbounded execution durations in a synchronous program. More generally, many Globally Asynchronous Locally Synchronous (GALS) approaches have been proposed to desynchronize the execution of several synchronous sub-systems [Gir05], though they are less directly related to our concerns.

1.3.2 Novelty of the proposed approach

As discussed above, the problem of extreme variability of execution times is well-known in the real-time research community. However, existing system models either focus on timing aspects and abstract from the system functionality or on the contrary focus on functionality and abstract from timing aspects. Our approach is fundamentally new because it considers functionality and time together, and because we consider this problem from a software programming point of view.

The key novelty that lies at the center of our approach is the ability to detect dynamically an upcoming WCET overrun before it occurs and to adapt the system behaviour to prevent the overrun altogether. By contrast, existing programming techniques focus on correcting the system behaviour to deal with the consequences of an overrun after it occurs. We believe this singularity in our approach is of major importance in the context of safety critical systems: we ensure that overruns will not occur and thus prevent deadline-misses. In more details, the novelties related to the different objectives of our approach can be summarized as follows:

Parametric WCET: Approaches to deal with parametric WCET computation have been proposed before. However, these approaches focus on a single aspect related to execution time variability, for instance, *either* the cache state *or* loop bounds. Furthermore, these methods usually either suffer from important pessimism, or they are intractable for non-trivial programs. Our project will introduce a complete method for *accurate* and *efficient* parametric WCET computation. Our parametric model will enable to represent various sources of variability (hardware and program effects) simultaneously.

Programming constructs for deadline-miss prevention: Currently, program adaptation in reaction to WCET-overrun is described using low-level languages. Instead, our approach will introduce dedicated constructs in synchronous languages, which have a higher-level of abstraction. The semantics of these constructs will be defined formally, as well as their compilation into lower-level code (C code). As a result, our approach will be the first to provide high-level correct-by-construction programming for systems with extreme execution time variability;

Iterative design and analysis methodology: The designer of a safety critical real-time system faces the problem of selecting the values of a great amount of parameters: WCET bounds, priorities, deadlines, periods, operating modes, etc. Exploring the space of solutions is all but trivial. In this project we will propose novel algorithms and methods to guide the designer in the development of a safety critical real-time system. In particular, we will extend the sensitivity analysis methodology [BDB08] to other parameters like selection of preemption points [Ber+11] and mode change [RC04]. We will apply techniques to explore the parameters space using optimization techniques combined with hints provided by the designer.

1.3.3 Related research structures

The project thematics fall within the scope of several national and regional research structures. For the Hauts-de-France region, this includes the *iTrans* competitivity cluster, the *Railenium* IRT, and the *ELSAT 2020* CPER on the topic of secure and automated transports (trains or cars). For the Occitanie region, the project is tightly related to the activities of the *Aerospace Valley* competitivity cluster and the *Saint Exupery* IRT, on the topic of critical aerospace embedded systems. For the Grand East region, the project is in the technological scope of the Materalia Cluster.To the best of our knowledge, the use of programming constructs for addressing the problem of execution time variability in real-time systems is not studied by any current or past national or European research project.

2 Organization

2.1 Scientific coordinator

The project coordinator is Giuseppe Lipari. He is full professor at University of Lille 1 since 2014. His research interests are in real-time systems, real-time operating systems, scheduling algorithms, embedded systems. He has published more than 120 peer-reviewed publications on these topics. He has been partner leader in many EU projects on real-time embedded systems in the FP5, FP6 and FP7 framework programs of the European Commission (FRESCOR, OCERA, RI-MACS, ACTORS, SoOS, and others). A more detailed Curriculum Vitae is available in the annex.

2.2 Consortium

The consortium is composed of two research institutions (CRIStAL and ONERA Toulouse) and one enterprise (RealTime-at-Work) for a total of 7 permanent staff members.

CRIStAL, Univ. of Lille 1

Giuseppe Lipari is full professor at University of Lille 1. His full CV is available in the annex.

Julien Forget is associate professor at University of Lille 1 since 2010. He received his Ph.D. degree from Univ. Toulouse in 2009. He also worked as an engineer from 2002 to 2006. His research interests are centered on the programming of critical systems, including formal languages, compilation and real-time scheduling. He has a total of 15 peer-reviewed publications on these topics.

Clément Ballabriga is associate professor at University of Lille 1 since 2014. He received the Ph.D degree from University of Toulouse in 2010. His research interests include embedded and real-time systems, static analysis, and WCET computation. He has co-authored a total of 13 peer-reviewed publications on these topics.

ONERA, Toulouse

Frédéric Boniol is full professor at University of Toulouse and has a research position at ONERA Toulouse in the computer science department. He received the Ph.D. degree from University of Toulouse in 1997. He has been working on the modelling and verification of embedded and real-time systems since 1989. His activities and research interests include modeling languages for real-time and distributed systems, formal methods and computer-aided verification applied to avionics systems. He has published 50 peer-reviewed articles in this area.

Luca Santinelli is researcher at ONERA Toulouse in the computer science department. He received the Ph.D. degree from Scuola Superiore Sant'Anna (Pisa, Italy) in 2010. His main research interests include real-

time operating systems, scheduling algorithms, energy aware scheduling and probabilistic real-time analysis. He is author of more than 40 peer-reviewed publications on these topics.

RealTime-at-Work

Loïc Fejoz is a research engineer at RTaW. He has a background in formal methods and software development tools and processes. He received the PhD. degree from University of Nancy 1 in 2009, for his thesis funded by a Microsoft Research Phd Scolarship. He has been working since then at RTaW on projects involving MDE, formal verification and performance evaluation, and he is the main designer of the CPAL language.

Lionel Havet is a research engineer at RTaW. He graduated from ENSICA aviation and aerospace engineer school in 1996. He worked in the area of embedded systems at Sagem, Giat Industries, Alspace, Philips. He has lead during 3 years the development of the embedded software in automatic gearboxes at General Motors. Lionel Havet has an extensive practical experience in the design of control systems and the development of embedded software for critical systems. He joined RTaW in 2013 and is responsible for the development of the CPAL toolset.

Experience and coordination The consortium is a balanced mix of senior and young researchers, and industrial experts, all of them working in the field of critical real-time systems. The members of the project are experts on all topics covered by the project: scheduling (G. Lipari [But+02; LB05; Lun+15], L. Santinelli [Cuc+12; Luc+14]), programming languages (J. Forget [Pag+11], F. Boniol [For+10]), WCET analysis (C. Ballabriga [Cho+13; Bou+08]), design and development of critical systems (L. Pejoz [ANF15; SNF15; Cia+16] and L. Havet [Nav+16; Sun+16]).

Many of the members of the project have experience in the development of software programs for real-time systems (CPAL, a language and model-driven environment for real-time systems; SCHED_DEADLINE, a scheduler for Linux; ERIKA, a RTOS for automotive systems; Prelude, a real-time language with its compiler; OTAWA, a tool for WCET computation) that are now used in the industry and in the open source domain.

Some members have participated to EU projects and French-funded projects that are related to the topics of this proposal, all in the domain of real-time systems: FP7 PROARTIS, ITEA2 TIMMO-2-USE, FP7 Dreams and FUI Waruna. Finally, the three partners have already worked together in the past and published papers together. The consortium will exploit such synergies to achieve the project objectives.

2.3 Scientific program

2.3.1 Dynamic forecast of extreme execution times

A widely-used approach for statically estimating the WCET of a task consists in providing an abstract model of the program and the hardware features (such as cache), and then computing the WCET based on this model. The result is guaranteed safe, but usually the process involves making conservative (pessimistic) hypotheses about software and hardware characteristics that cannot be determined statically: the most notable example is the content of the cache, which depends on the previous execution of other tasks or on task preemptions. As a consequence, WCET computation by static analysis usually suffers from a large overestimation.

Our approach aims at significantly reducing this overestimation. First, a *parametric WCET* is computed statically, before task execution. It consists of a WCET formula that depends on parameters such as cache state, task input values, loop bounds, etc. This formula is then instantiated dynamically at run-time at task release, at which point parameter values become available, producing a WCET estimate that is tighter than a purely statically estimated WCET.

We will focus in particular on the WCET variability caused by cache states. The techniques we will develop will also apply to the computation of the WCET in terms of other parameters, in particular software parameters such as function or input arguments, which can effect loop bounds and feasible paths. WCET variability caused by the cache state is an important topic because the cache behaviour produces global timing effects (cache block eviction at some program location can produce effects on some other remote location) and because the WCET variation due to cache will occur in most programs.

To evaluate the impact of the cache on the WCET, we will need to follow the evolution of its content. First, we need to know the cache content when a task starts. Second, we need to know the impact of preemptions on the cache. To limit the impact of preemptions, we will use the technique of *preemption points* [Ber+11], which limits the execution points where a task instance can be preempted. Third, in a multicore platform the

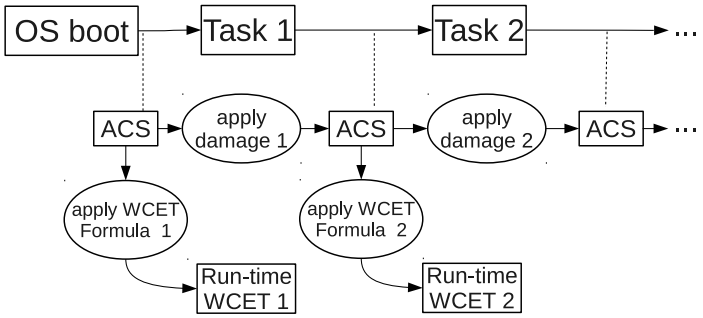


Figure 1: Run-time cache state tracking and WCET formula instantiation.

shared cache may be accessed by tasks executing on other cores. In this project we focus on **single processor** hardware platforms. The extension of our work to multicore systems is discussed in Section 2.3.5.

Tracking cache states Most hardware processors do not enable direct access to cache states. Thus, we plan to use the OS in order to estimate the cache state despite this hardware limitation. Instead of tracking the exact cache state, we can use a safe cache abstraction, such as the ACS (Abstract Cache State) described in [Fer+99]. An ACS value is an abstraction for a set of concrete cache states. The exact definition of the mapping between ACS and concrete cache states is implementation dependent. A widely-used implementation is the May/Must ACS, which tracks the minimum and maximum possible Least Recently Used (LRU) ages of each cache blocks, for all possible execution paths. We will use the OS to maintain this ACS at run-time so that it remains a valid abstraction of the real cache state throughout the execution.

In a multi-task system, maintaining the ACS requires to track the impact of task execution on the cache. For this purpose, we plan to use the *cache damage* technique [BCS08] as a starting point. For each task we maintain an ACS and a cache damage function. When a task is about to execute, we compute its ACS by applying the cache damage of all tasks that have been executed since the last execution of this task, and we represent this ACS as a parameter in the WCET formula.

Many improvements can be made over the method presented in [BCS08]. For instance, the various abstractions (ACS and damage function) could be adapted to be less costly to compute (at the cost of precision), if the experiments show a need to do so. The reverse (sacrificing speed for precision) is also possible if needed. For example, instead of computing a fixed damage for each task, it should be possible to make it scenario-sensitive, and have a different damage for different task execution scenario (based on paths or input values).

Producing and evaluating the WCET formula To produce the WCET formula, we plan to reuse our work from [BFL15], which presents a method for computing the parametric-WCET of a program. The method is generic as parameters can represent various factors, such as a formula input values, function parameters, etc. In this project, we need to adapt the work in [BFL15] to introduce cache-related parameters (such as the ACS). There are two WCET computation phases: an off-line phase (for computing the WCET formula) and an on-line phase (for evaluating the formula).

In order to produce the formula during the off-line phase, we need to determine which parts of the task are dependent on the cache state. We plan to do this by statically analyzing the task and detecting the worst-case and best-case output cache state depending on the cache state at the beginning of the task.

We also have to provide some adaptation layer, to be used at run-time, to translate the abstract cache states into parameters suitable for the WCET formula evaluation. A preliminary step could be performed statically, by computing the different cache damages across the different possible execution paths of a task. This information can then be used at run-time to predict the impact of a task execution on the cache, given the abstract cache state at the task entry point.

To summarize, for each task the cache damage and the WCET formula are computed statically and the WCET formula is then instantiated at run-time. Figure 1 illustrates the run-time workflow and details ACS tracking and WCET formula instantiation are related.

2.3.2 Deadline-miss prevention

The second objective of our work is to propose language constructs that enable the specification of an alternative system behaviour when an extreme value of the execution time is predicted. In this way, it will be possible to dynamically adapt the behaviour of the system to *prevent* WCET overruns and deadline-misses, instead of *correcting* their consequences.

We could program the adaptive behaviour of the system by using low-level languages such as C. When a task instance starts executing, it first instantiates the parametric WCET using parameters such as input data and cache state; if the WCET is below a certain limit, the task executes normally, otherwise it executes a simpler backup behaviour, which is guaranteed to have execution time inferior to a certain. However, it might be difficult to design and to statically analyze large programs with complex adaptive behaviours and still guarantee correctness. In this project we will investigate higher level language constructs to support adaptive programs that are **correct-by-construction**.

Required language constructs First, we need to provide a construct that instantiates a parametric WCET formula. The formula will be instantiated repeatedly at run-time, thus its computation must be fast. Since the formula will be part of the program code, its size must remain small. Finally, we do not need to compute the exact WCET from the formula, instead we only need to check that it remains lower than a given threshold, which simplifies the parametric-WCET computation.

Second, we need constructs to specify how the system reacts when a WCET is higher than its acceptable threshold. This will lead to a program which consists of several *operating modes*: a main steady mode, and several recovery modes, when some WCET exceed their threshold. In the following, an *extreme instance* refers to an instance of a task whose execution time is deemed to reach an extreme value. The task this instance belongs to is called an *extreme task*. Figure 2 illustrates several different possible ways to react to an extreme instance. Extreme instances are marked by an X and system adaptations (recovery modes) are depicted in gray:

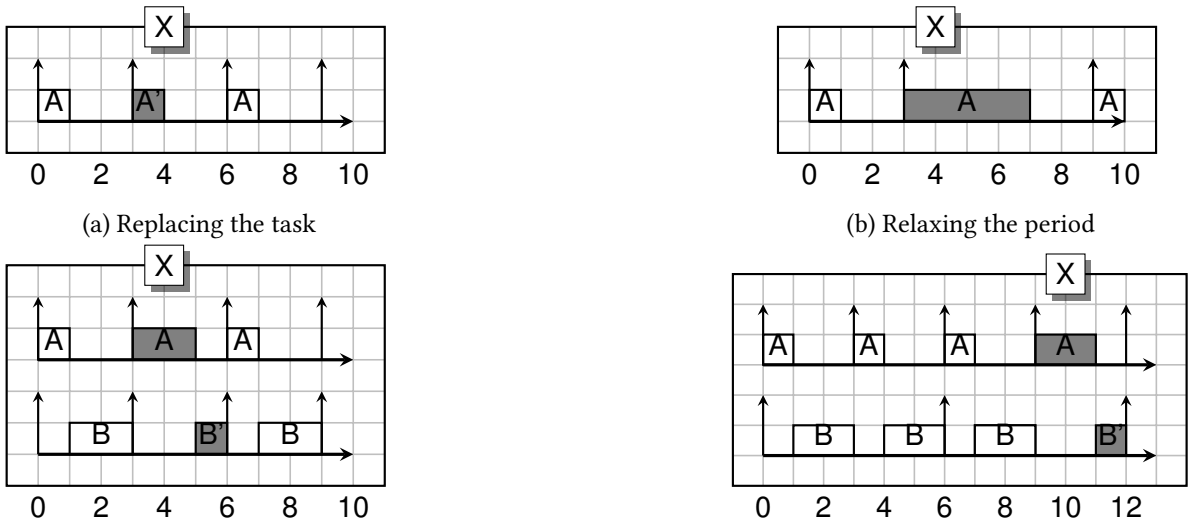
- The simplest adaptation consists in replacing the extreme task by a different one. For instance, in Figure 2a, A' replaces A;
- In Figure 2c, though A is the extreme task we decide to replace B by B'. An additional difficulty arises in case we want to alter the behaviour of a task whose rate is different from the extreme task. For instance, in Figure 2d, B has already started its execution when the extreme instance of A occurs;
- The last possibility is to change the timing parameters of some tasks (either the extreme task or others). For instance, in Figure 2b, we increase the period of A when an extreme instance occurs.

Synchronous language extensions We plan to introduce the language mechanisms we just described as high-level dedicated constructs in Synchronous data-flow languages [Ben+03]. During the project, we will study the semantics of these language mechanisms in a general synchronous data-flow setting and experiment them in the Prelude language [For+10; Pag+11]. First, as a reminder, let us consider a simple Prelude program:

```
imported node swap(i, j: int) returns (o, p: int) wcet 100;
imported node id(i: int) returns (o: int) wcet 150;
sensor i wcet 50; actuator o wcet 50;

node sampling(i: int rate (500, 0)) returns (o: int rate (500,0))
  var vf, vs;
let
  (o, vf)=swap(i, (5 fby vs)*^3);
  vs=id(vf/^3);
tel
```

We first declare all the nodes that will be used and their WCETs. Then, we declare that `sampling` inputs and outputs all have a period of 500. `vf/^3` produces a flow that only keeps the first out of three successive values of `vf`, thus producing a flow three times slower than `vf`. Flow `f*^3` does the opposite: it repeats each



(a) Replacing the task

(b) Relaxing the period

(c) Altering another task

(d) Altering a slower task

Figure 2: Reacting to an execution time extreme value

value of `f` three times, producing a flow three times faster. Node execution rates are deduced from flow rates: `swap` has period 500 while `id` has rate 1500. The *clock* of a flow defines its rate, i.e. when it produces values. The *clock calculus*, checks the consistency of the clocks of a program. For instance, it checks that the inputs of imported nodes all have the same clock, that is to say that they are *synchronous*.

Let us now consider how we could program the different recovery modes presented in Figure 2. For each case, we propose a synchronous data-flow implementation and we emphasize the language extensions required (they appear in red in the program listings). The simplest recovery mode, which relies on a backup task (Figure 2a), could be programmed as shown below.

```
node main(i,j:int) returns (o:int)
var c;
let
  c=is_wcet_lt(A,p,8);
  o=merge(c,A(i when c,j when c),A'(i whennot c, j whennot c));
tel
```

Operator `is_wcet_lt` will test whether the execution time of `A` is lower than 8, given some arbitrary parameter `p` (which could for instance correspond to an ACS or to some input value). `i when c` is a flow with the same values as `i`, except that it bears a value only when `c` is true (no value otherwise). The `merge` says that if the execution time is below 8, `o` is computed by `A`, otherwise it is computed by `A'`. The only language construct that does not currently exist for this example is `is_wcet_lt`. However, we also need to devise compilation mechanisms that keep track of the fact that `A` will not execute for longer than 8, to provide this information to the schedulability analysis.

The recovery mode described in Figure 2c can be programmed in a similar way. The recovery mode described in Figure 2d could be programmed as shown below:

```
imported node B(i:int) returns (o:int) wcet 10;
imported node B'(i:int) returns (o:int) wcet 6;
imported node A(i:int) returns (o:int) wcet X;
sensor i wcet 1; sensor j wcet 1; actuator o wcet 1; actuator p wcet 1;

node main(i: int rate (30,0); j:int rate(10,0)) returns (o,p:int)
var c2;
let
  c2=is_wcet_lt(A,p,3);
  o=B(i) abort(c2) B'(i);
  p=A(j);
tel
```

We need to introduce a new construct, the `abort` operator, that will be capable of interrupting a task after its execution has started. In this example, `B` is aborted to execute `B'` instead. This is a completely new concept in the field of synchronous languages, which will raise interesting and challenging semantics questions.

Finally, we propose an implementation of the recovery mode described in Figure 2b:

```
imported node A(i:int) returns (o:int) wcet X;
sensor i wcet 1; actuator o wcet 1;

node main(i:int rate(10,0)) returns (o,p:int)
var c;
let
  c=is_wcet_lt(A,p,9);
  o=A(c ?^ i :^ i/^2);
tel
```

`c ?^ i :^i/^2` has value `i` if `c` is true, `i/^2` otherwise (two times slower). This cannot be programmed using operator `merge` because `merge` requires operands with complementary clocks. The difficulty here resides in introducing the possibility to specify a flow whose clock rate changes dynamically (`o` in this example), while keeping the system analyzable by a schedulability analysis.

In the previous examples, we only considered recovery modes that concern a single task. In the more general case, we may need to alter several tasks simultaneously (e.g. replace a set of tasks by a different set of tasks). We plan to rely on mode automata [CPP05] to handle such mode changes.

Design methodology Our design methodology is summarized in Figure 3. It relies on several separate tools. First, the programmer will design the system with Prelude, to specify precisely and formally real-time characteristics and data-communications. Second, the Prelude compiler will generate a CPAL model[1]. CPAL is an acronym for the *Cyber-Physical Action Language*. It is meant to model, simulate, verify and program Cyber-Physical Systems (CPS), such as those considered in the project. While Prelude focuses on the high-level software architecture of a system, CPAL is more oriented towards the description of the functional aspects of the system. In particular, it supports states machines, which we will use to implement recovery modes. Third, the CPAL compiler will translate the program into C code and a C compiler will produce the embedded binary code.

Once the first version of the binary code is available, we will analyze its execution time and produce a parametric WCET using OTAWA[2]. This information will be used in conjunction with the CPAL code to perform a timing analysis of the system. This step can be performed either by simulation, using the CPAL simulator, or with dedicated analyses relying on Time4Sys[3]. Time4Sys provides meta-models, transformation rules, and authoring tools required to perform analysis or simulation of the timing aspects in the design of a real-time system. The timing analysis will provide information useful to modify the Prelude in case some real-time constraints are violated. Then, we iterate this design process until constraints are met (see next Section for more details).

2.3.3 Timing analysis

To complete the design methodology, we need to analyze the system and guarantee that all tasks will always complete before their deadline constraints. Classical schedulability analysis takes as input the task parameters (periods, deadlines, priorities and non-parametric WCETs) and gives a yes/no response. The large WCET estimates produced in this phase (due to the intrinsic variability of the execution time) may lead to an unschedulable system. We will instead rely on parametric WCETs to reduce this pessimism.

We must compute bounds to WCET values under which the task set is schedulable (i.e. all tasks meet their deadlines). To do this, we propose to use *sensitivity analysis* [BDB08; Sun+14] that computes the *intervals* of admissible values of the execution times that make the system schedulable.

Once these bounds have been computed, the designer codes alternative system behaviours, using the language extensions proposed in Section 2.3.2, for cases where the parametric formula returns a value that

[1] https://www.designcps.com/
[2] http://www.otawa.fr/
[3] https://www.polarsys.org/proposals/polarsys-time4sys

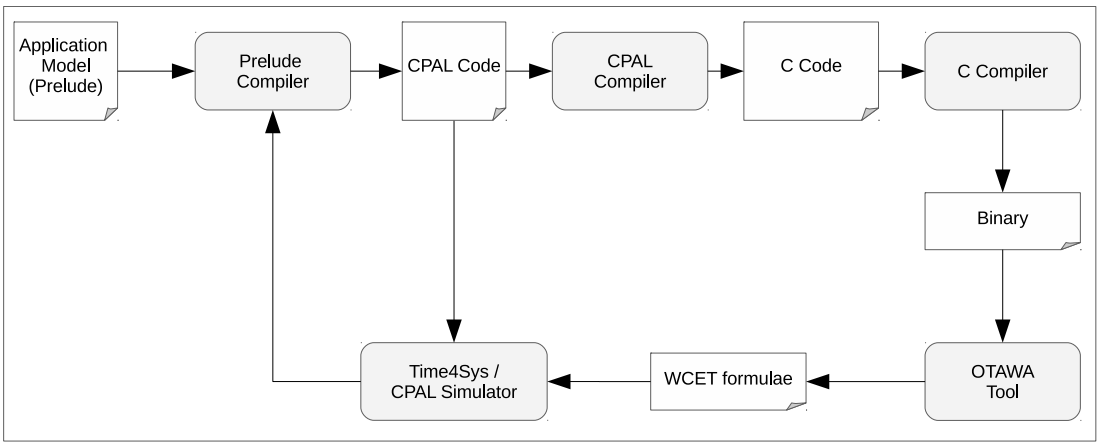


Figure 3: Design methodology overview

exceeds the bound computed in the previous step. This will lead to a system that has several operating modes that depend on execution time variations. Then, a new schedulability analysis is performed to verify the correctness of the modified system. If the system is still unschedulable, the designer can iterate the process.

The main challenge here is to guide the developer in exploring the space of schedulable solutions. There is a vast literature on the optimal selection of preemption points; operating modes have been investigated in the real-time literature [RC04] with the goal of guaranteeing that the system remains schedulable during a mode-change, and specific scheduling protocols have been proposed. However, sensitivity analysis for systems with mode changes and preemption points has never been investigated so far. Thus, a problem that we need to address in this project is to extend the theory of sensitivity analysis to the case of preemption point selection and multiple operating modes.

A second challenge in putting all this research together is that the number of parameters of the scheduling problem is large: it includes classical task parameters such as periods, priorities, deadlines, and WCET bounds and, in addition, preemption points and operating modes. It is not realistic to let the designer explore the space of solutions manually. On the other hand a classical optimization algorithm may hide important details, and it may become impossible for the designer to understand how a certain design choice impacts the final system configuration. We will investigate different strategies for the exploration of the solution space by using a semi-automatic procedure. The tool will propose a set of "optimal" solutions based on hints provided by the designer. One possibility is to first explore variability in blocks with larger variation of execution time, i.e. those on which the impact of WCET parameters is larger. The designer could help the tool by tagging more important blocks, or by just addressing one single subsystem at a time. In addition, we will use run-time measurement of execution time to compute how likely a task is to execute for more than its computed execution bound in a real setting.

2.3.4 Case study

The case study is the landing gear control system of an aircraft [BW14]. It is in charge of manoeuvring landing gears and associated doors. The system is made up of three gears: front, left and right. Each gear has a landing gear uplock box and a door with two latching boxes. Gears and doors are manoeuvred by hydraulic jacks. Hydraulic power is provided by a command unit that consists of a set of sensors and actuators. The **actuators** are: solenoid valves to isolate the emergency hydraulic system; electrovalves to open and close doors, to bring down and to retract gears. The **sensors** provide the current state of different parts of the system: position of landing gear actuating cylinders, of lock actuators, state of landing gear shock absorbers, state of the hydraulic system, etc. To command the movement of gears, the pilot has a set of buttons with two positions (up and down) and a set of lights showing the current position of gears and doors.

The control software, which is in charge of controlling the three gears and associated doors, is part of a feedback loop with the physical system, and produces commands for the distribution elements of the hydraulic system based on the sensor values and on the pilot orders. The **inputs** of the software are: command button values (up or down); position of the six doors locks; position of the three actuating cylinders; position of the three landing gears locks; state of the drag struts; state of shock absorbers; an oil pressure switch giving

the pressure of the hydraulic system after the isolating solenoid valve. The **outputs** of the software are: commands to start and stop stimulation of the electrovalves and solenoid valves; a set of warnings for the pilot in case of malfunction or of absence of response from mechanical components.

The control software consists of a set of specialised functions: 1) Monitoring functions for gears and doors, which signal any fault int the landing system; 2) A command function that implements the sequence gear movements (up/down). This function directly controls the mechanical components. It is made up of a function that computes stimulation commands for each component and of functions to manage the commands emission; 3) Monitoring functions to verify that the system reacts correctly and to control the functional and temporal coherence of the orders then to actuators.

In nominal mode, the system exhibits low variability in terms of control and data flows. Thus, it does not exhibit large variations of execution times. However, in faulty modes (when faults occur somewhere in the physical part of the system), the control and data flows change, in order to detect and to report inconsistent situations, which leads to higher execution times.

The functions of the control system are implemented by about 20 periodic and sequential processes executed at different periods. Two versions of the software are readily available: one programmed in Prelude and Lustre by Onera, and one programmed in CPAL by RTaW. Each version contains about three thousand lines of code.

2.3.5 Extension to multicore systems

In this project we focus on single processor hardware platforms. This is due to the technical difficulties already present in our approach and described in the previous sections. Focusing on single processor systems should not substantially reduce the applicability of the results since safety-critical systems are built on partitioned multiprocessor scheduling solutions, which, up to a certain point, amounts to having several single processor systems running in parallel. Nevertheless, we envision possible extensions to multicore systems after the project results have been validated.

In particular, one popular approach for executing hard real-time safety critical applications on multicore systems is to partition the shared cache among the cores using one of the techniques available in the literature, and to use a resource reservation technique to reserve the bus bandwidth for access to the main memory. One notable example of this technique is MemGuard [Yun+13]. By using this kind of techniques, our methodology can be easily applied to multicore systems by considering each core as a single processor with dedicated bus and bounded memory access delay. Naturally, the introduction of a multicore system adds several dimension to the problem: 1) how to allocate tasks on cores in a optimal way; 2) how to optimally partition the cache; 3) how to assigns bus bandwidth to cores; etc.

3 Impact and benefits

3.1 Thematics of the call

The proposal addresses Challenge 7 (défi: “Société de l’information et de la communication”) of the call. Our proposal addresses the thematics of Axe 3 “Computer sciences and technologies” (“Sciences et technologies logicielles”), such as real-time operating systems, programming languages and tools for embedded systems, program analysis, and verification timing properties of programs.

The problem of the variability of execution times is strongly tied to the evolution of the architectures of present single core micro-controllers and multi-core hardware platforms. Addressing such a problem is of fundamental importance for the development of safety critical embedded systems. The lack of correct-by-construction methodology for supporting execution time variability in safety-critical real-time systems is remarkable despite some recent efforts (see Section 1.3.1) in scheduling analysis and in probabilistic analysis. However, very few research works try to integrate design and analysis taking into account both functional and timing correctness.

Concerning ERC classification, our proposal clearly falls into *PE6_2 – Computer systems, parallel/ distributed systems, sensor networks, embedded systems, cyber-physical systems*. Since one part of the project is concerned with formal semantics of programming languages, we are also in *PE6_3 – Software engineering, operating systems, computer languages*. Finally, we apply formal methods to parametric WCET analysis and to the

schedulability analysis, hence we are also in *PE6_4 - Theoretical computer science, formal methods, and quantum computing*.

3.2 Impact

Scientific impact: The objective of CORTEVA is to contribute to the design and development of next generation safety critical embedded real-time software systems. Safety critical software can be found in many transportation systems on which we rely for our everyday lives (airplanes, cars, trains). Hence, it is of paramount importance to ensure the correctness of the software, else some human life could be in danger. At the same time, it is necessary to contain the cost of developing and producing such systems. The methodology proposed in our project will eliminate some common errors by using automatic code generation and compilation from provably correct models. CORTEVA will:

- *enhance safety and predictability* of commercial off-the-shelf multi-core processors granting their application for real-time safety critical systems: more resources accessible at a lower cost;
- eliminate the temptation to ignore extreme values of execution times enforcing the correct-by-construction design of real-time safety critical: *correct-by-design predictability in multiple operational modes*;
- *assure a better utilization of the hardware resources* with the reduction of both development and production costs. The timing analysis and the schedulability analysis will better cope with the different working condition of the system for a reduced pessimism.

Socio-Economic impact: Since CORTEVA aims at enhanced resource utilization with correct-by-construction methodologies for supporting the execution time variability, it will have an important effect on the system cost reduction.

The methodologies and the solutions proposed by CORTEVA apply to the avionic, aerospace and public transportation sectors, uplifting the innovation capacity and competitiveness of France's embedded systems' sector.

Contributions from ONERA will demonstrate the applicability of the proposed methodology to industrial real-time safety critical application by applying a research oriented and formal approach. In this way, we will at the same time support SME activities which are vital for the social and economic fabric of France computer science activities and give them an advantage over competitors.