KU LEUVEN

FACULTY OF ENGINEERING SCIENCE

Exploring schedulability on TEEs with availability guarantees

Providing real-time guarantees with acceptance tests, dynamic scheduling and the Aion architecture

Emeric Goossens

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Gedistribueerde systemen

Promotoren:
Prof. dr. ir. F. Piessens
Dr. J.T. Mühlberg

Assessoren:
F. Alder
Dr. ir. L. Sion

Begeleiders:
Dr. J.T. Mühlberg
F. Alder

Academiejaar 2021 – 2022

Preface

First, I would like to thank my promotor for his helpful feedback on the intermediate presentations. The hint to provide a real-life example, ensured that this thesis became not too abstract. Secondly, special thanks go to my daily supervisors. They helped me to focus on the important aspects of my research. Their enthusiasm and interest, combined with some jokes in the meetings, kept me motivated. Thirdly, I would like to thank my friends and family for their support at each moment. Especially my family had to endure the stressful moments before the deadline. Finally, I'm grateful for all the other people that I met this year, and the sometimes unexpected, but interesting conversations.

I would like to conclude with a quote from *The Lord of the Rings*, the famous story by Tolkien.

Not all those who wander are lost.

This year, for my thesis, I've sometimes had the impression of wandering around without road map. This can be scary at times. Luckily, there are these compasses along the way, to confirm that you're heading in the right direction. I'm grateful for that.

Emeric Goossens

Contents

Abstract

In real-time systems with different mutually distrusting stakeholders, applications should be certain to execute timely, even in the presence of an attacker. An earlier approach introduced an embedded Trusted Execution architecture with availability extensions. This could enforce an upper bound on the activation latency for the applications on the system. While providing real-time guarantees by means of the fixed upper bound, the system could not take into account differences in application requirements. This had two consequences. First, for a given upper bound, no guarantees could be given to applications with more stringent latency requirements. Secondly, the system's utilization would be sub-optimal if not all applications required the provided level of responsiveness.

In this thesis, an approach using schedulability tests is proposed, extending the type of availability guarantees that can be given and allowing the applications to specify real-time requirements. These tests are dependent on the assumptions made about the system and its applications. In the approach, a trusted scheduler is responsible of enforcing these premises.

The schedulability test will be used to assess whether a schedule exists where all real-time requirements are met for a given set of applications. To develop the tests, we propose a methodology based on logical modeling to guide the design process and to prove their correctness. This is more efficient and error-proof than a manual approach. We use this method to derive tests for different sets of system assumptions. Finally, we illustrate the real-life applicability of the approach in the context of a smart metering system. It shows that strong availability can be achieved in the presence of an attacker. Furthermore, the approach allows applications with very diverse real-time requirements, improving the utilization of the system.

Samenvatting

In open real-time systemen kunnen programma's van verschillende partijen uitgevoerd worden op dezelfde processor, waarbij die partijen elkaar in de meeste gevallen niet vertrouwen. In dat scenario moet de tijdige uitvoering van de toepassingen verzekerd kunnen worden, zelfs in de aanwezigheid van een aanvaller. Een eerder werk introduceerde een embedded Trusted Execution-architectuur met uitbreidingen om availability te garanderen. Deze architectuur beperkte de activatie latency van de toepassingen op het systeem tot een bovengrens. Hoewel het systeem real-time garanties kon bieden door middel van deze bovengrens, hield het geen rekening met verschillen in vereisten tussen toepassingen. Dit had twee gevolgen. Ten eerste konden geen garanties worden gegeven aan toepassingen met strengere latency vereisten dan de vastgelegde bovengrens. Ten tweede werd het processorgebruik suboptimaal als niet alle toepassingen de geboden garanties nodig hadden.

In deze thesis wordt een aanpak uitgewerkt, die gebruik maakt van schedulability tests. Deze benadering breidt het soort vereisten dat kan verzekerd worden uit en laat de toepassingen toe om zulke real-time vereisten te specifiëren. De tests zijn afhankelijk van de aannames die zijn gemaakt over het systeem en de aanwezige toepassingen. Een beveiligde scheduler is verantwoordelijk voor het verzekeren van de geldigheid van deze assumpties.

De schedulability test heeft als doel om, voor een verzameling toepassingen, in te schatten of er een planning bestaat waarin alle real-time vereisten voldaan zijn. Om de tests te ontwikkelen, stellen we een werkwijze voor die gebruikt maakt van logische modellen. Deze modellen laten toe het ontwerpproces computergestuurd te begeleiden en de correctheid van de test te bewijzen. Dit is eenvoudiger en minder foutgevoelig dan een handmatige aanpak. We gebruiken deze werkwijze om tests af te leiden voor verschillende aannames. Ten slotte illustreren we de praktische toepasbaarheid van de availability aanpak in de context van een smart metering systeem. Dit illustreert dat een sterke mate van betrouwbaarheid kan worden bereikt ondanks de aanwezigheid van een aanvaller. De methode laat bovendien toepassingen toe met zeer uiteenlopende real-time vereisten, waardoor de processor efficiënter gebruikt wordt.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations

DoS	Denial of Service.
EDF	Earliest Deadline First.
FOL	First-Order Logic.
IaaS	Infrastructure as a Service.
IoT	Internet of Things.
IP	Infrastructure Provider.
lcm	Least Common Multiple.
MCU	Microcontroller Unit.
OS	Operating System.
OW	Observation Window.
PL	Propositional Logic.
SMT	Satisfiability Modulo Theories.
SP	Software Provider.
TCB	Trusted Computing Base.
TEE	Trusted Execution Environment.
WCET	Worst Case Execution Time.

Chapter 1

Introduction

The amount of digital devices world-wide is continually increasing. More and more applications are equipped with digital support, also in domains with high safety requirements. By interconnecting these devices, intelligent and dynamic systems are created. This is labeled the Internet of Things (IoT).
A lot of these applications have time-related requirements. Systems where timeliness is a functional requirement are called ‘real-time’ systems. Correct functioning of these time-critical systems does not only depend on the logical correctness of the result of a computation, but also on its release in time. [6] Guaranteeing that the system can meet the timing requirements is thus of high importance.
There is a trend to allow on the IoT devices the deployment of applications from different stakeholders, achieving a type of Infrastructure as a Service (IaaS). Furthermore, devices are becoming increasingly dynamic, allowing them to be updated after deployment. Besides adapting present functionality, new applications could be deployed at run-time on the system. Alder et al.[1] use the term ’open system’ to denote these dynamic systems with different stakeholders, who do not necessarily trust each other.
In the automotive industry for example, cars are increasingly becoming autonomous, embedding applications with stringent real-time requirements. For instance, the emergency brake subsystem should react within very strict time limits to a dangerous situation. These safety-critical tasks need to be carried out with high reliability. On the other hand, the car provides an increasing set of functionality to the driver, potentially from different software providers. A car could for example be enhanced with an application that shows parking lots and available places in a given city, using the city’s parking monitoring system as discussed in [13]. This application could be provided by the city itself and could be added to the cars functionality by remote update. However, this new functionality should not be able to nullify the earlier real-time properties of the system (e.g., the brake system).

1.1 Security requirements

Combining the extensible approach of ’open systems’ with real-time requirements is thus a challenge. In real-time systems that are ’open’ to be updated, the system should ensure that malicious applications cannot violate the security requirements. Therefore it needs to be prevented that one application spies, alters or stops the execution of another application.
In computer security there are three very common requirement types, often called the CIA-triad. First of all, most applications require a form of **confidentiality**: sensitive data should not be read by unauthorized parties (could be applications or human actors). Secondly, **integrity** is an important concern: Only authorized parties should be able to change data of an application. On the other hand, application code is also an asset that should be protected from tampering by unauthorized parties. Finally, **availability** covers operational up-time, but requirements related to timeliness as well.

Achieving these security goals on resource-constrained devices is not straightforward, especially when ‘opening’ them to the external world. A lot of approaches are designed for systems without these resource constraints and rely on a full-fledged Operating System (OS) to protect the system. In contrast, special IoT OS were designed (e.g., RIOT [2]), that account for the resource constraints, but assume that all applications can be trusted. However, in the case of open systems, applications could be under control of an attacker. Therefore, the developer of one application should not be forced to trust all the other applications on the system.

1.2 TEE and Aion

Trusted Execution Environment (TEE) architectures try to solve this problem, by reducing the Trusted Computing Base (TCB) of an application. In system security, the term TCB is used to refer to the components of the system that have to be trusted by an application. The TCB thus reflects the assumptions made about the hardware and software of the device. In TEEs the TCB is reduced by isolating applications from each other and by trusting on the hardware for confidentiality and integrity (Sancus is an example [21]). A Software Provider (SP) has thus only to trust the hardware, device drivers (if external devices are used) and his application code to be certain of authentic execution of his application.
However, TEEs cannot guarantee availability. Nevertheless, some systems (like the car) have safety-critical timing requirements. The Aion architecture, a hardware-software co-design proposed by Alder et al. [1], addresses the availability problem. It is used on top of a TEE to provide availability, next to confidentiality and integrity. By adding a trusted software scheduler to the TCB of the TEE architecture, it reaches two important achievements in the domain of availability. First of all, Aion can ensure progress of an application. It introduces bounded atomicity, allowing atomic executions of code for a fixed maximum of processor cycles. Next to that,

Aion can ensure an upper bound on the latency for an application. This can thus provide timeliness guarantees, given that the number of applications on the system is bounded.

1.3 The Problem

Even in the presence of a trusted scheduler, no absolute guarantees can be given. An attacker could still submit a multitude of applications and thus overload the system, resulting in high latencies. To prevent this, a mechanism has to regulate the applications that can be accepted on the device. This is commonly called the acceptance problem. In Aion, this is solved by setting a higher bound on the number of applications on the system. However, this approach is very inflexible, giving the same guarantees to all applications independent of their characteristics and needs. No distinction is made between processor-intensive and light applications. In the car example, this would mean that the same guarantees would be given to the dashboard monitor as to the brake engine.
It could be beneficial to replace the fixed bound on the number of applications, by a test that also take into account the characteristics and requirements of that application. This mechanism will be able to decide whether a new application (with known characteristics) can be accepted on the device without overloading the system. These ‘acceptance’ tests take the run-time characteristics and real-time requirements of the applications as input and produce a boolean output (yes/no). This approach makes it possible for SP to have different requirements on the system.

This thesis presents an approach for providing availability guarantees to applications. The different applications on the system and the system’s OS are explicitly not trusted for availability. The goal of the thesis is to guarantee timeliness to an application on the system in the assumption that all other applications on the system are attacker-controlled. Even if n malicious applications are present on the system, the benign application should be able to behave normally. This contrasts with threat models that assume cooperativeness to ensure availability.
The approach uses schedulability analysis in the context of Aion. Scheduling in the Aion architecture has to account for bounded atomicity, being thus nor fully preemptive nor fully non-preemptive.

1.4 Contributions

In the context of this thesis, we make the following contributions:

- We propose an approach to provide real-time guarantees in the context of open systems. This approach takes into account the specific requirements of an application to provide availability while enabling efficient processor utilization.
- A simulator in Python is implemented (see Chapter 4), that simulates the EDF policy in the context of Aion (adapted with additional enforcement mechanisms).

Furthermore, a graphical representation of the schedule is generated. This makes it possible to visually check if a given task set is schedulable.

- A method to design acceptance tests is developed. It makes use of logic modeling to guide the user in the design process and to check formally the sufficiency of the obtained tests. Multiple logic models have been made corresponding to different sets of assumptions on the system.
- Acceptance tests are derived for different configurations.
- As a case study, the smart metering example (as discussed in [20]) is further extended. The designed approach makes it possible to provide availability guarantees to the applications, even in the presence of attackers, while reducing the number of processors needed for security.

1.5 Outline

The thesis is structured as follows:

- Chapter 2 contains background knowledge, that is used in the other chapters.
- Chapter 3 describes the execution environment and defines the proposed approach for providing availability guarantees.
- Chapter 4 goes more in depth on scheduling and provides insight in how a schedule can be attacked. The used scheduling policy is shortly discussed, followed by the implementation of the simulator. Finally, using the simulator, different problems that could occur on the system are described.
- Chapter 5 to 7 document the used approach for designing acceptance tests. In Chapter 5 a manual derivation method is shown. This approach is improved in Chapter 6 by using a Satisfiability Modulo Theories (SMT) solver to guide the design process and prove sufficiency. Finally, in Chapter 7 different acceptance tests for more complicate scenarios are designed with the improved method.
- In Chapter 8 a case study is examined. The real life example shows the relevance of the research question by discussing how the approach can be used in a smart metering context.
- To conclude, Chapter 9 provides a reflection on the obtained results. Next to insight into the advantages and disadvantages of the used availability approach, multiple starting points for further work are proposed. The chapter ends by discussing the research questions.

Chapter 2

Background

The thesis relies on knowledge in the fields of real-time systems, scheduling, schedulability analysis and theorem provers. This chapter provides an overview of the needed background information.

2.1 Real-time systems

Systems where the applications have constraints on the response time, are called real-time systems. The response time of an application is defined to be the time needed for the system to react (output) on some associated stimuli (input). The correctness in a real-time system requires timeliness, the response should be generated within a well determined period. As Burns and Welling [6] state:

> *Consequently, the correctness of a real-time system depends not only on the logical result of the computation but also on the time at which the results are produced.*

Real-time systems often show up in safety-critical areas, such as the automotive industry [26], where it is of high importance that the results are delivered timely. It is also very important that such systems do not fail, that they are more generally available. In other words the availability of the system, and more specifically of the application, must be ensured. This is not straightforward, as both expected difficulties in the application and faulty software design have to be considered. In this thesis, additionally, a third component comes into play, namely an active attacker trying to harm the availability.[6]

Hard real-time applications can be distinguished from soft real-time applications:

> *Service within this span of time must be guaranteed, categorizing the environment as "hard-real-time" in contrast to "soft-real-time" where a statistical distribution of response times is acceptable.* [17]

There are systems that support both types at the same time, even adding support for applications without deadlines at all. In this thesis however, only hard real-time applications will be considered.

Meeting the response time requirements is very important in those systems. Therefore, often more processing power is foreseen than needed on average, to ensure that the hard deadlines can be met in the case of the worst-case behavior. Furthermore, in most cases, the duration of a context switch and of interrupt handling is not negligible compared to the real-time requirements. This has thus to be carefully taken into account. [6]

2.2 Scheduling Scheme

Burns[6] defines scheduling as a way to restrict non-determinism. A scheduling scheme embeds two features. On the one hand it provides some ordering on the use of resources, namely the scheduling policy. On the other hand, it contains a schedulability test. It makes it possible to predict the worst-case situation and predict if real-time requirements can be met. [6]

In this thesis, we focus on EDF as the scheduling policy.

2.2.1 Scheduling terminology

In the scheduling domain, the term ‘task’ is commonly used to designate a thread, process or application that implements a part of the functionality of the system. The scheduling policy defines an order on the execution of tasks.
A task can be periodic with an associated period p_i. This task will have to execute once each period p_i. Moreover, tasks will sometimes place time requirements on their execution. In the case of periodic tasks, for each periodic execution a deadline has to be met. Finally, tasks that are already present on the system can be in three possible states. First, it can be ‘running’, which means that it is currently executing. Secondly, if a task is ready to execute, but it has to wait, it is said to be ‘ready’. Finally, a ‘sleeping’ task is one that currently does not need to be executed.

Scheduling policies can be static or dynamic. In a static policy, each task gets a specific (static) priority beforehand and these priorities determine the order of execution. In a dynamic policy, the order is dependent on time-dependent characteristics of the tasks, e.g. the deadline.
Furthermore, policies are either preemptive or non-preemptive. Preemption means that the system will switch tasks if another task with a higher priority (static policies) or a nearer deadline (e.g., EDF) is ready to run. [6] To switch between tasks, the task context is saved and a context switch occurs, after which the more critical task is able to run. In contrast, non-preemptive policies will allow the running task to complete. A task with a higher priority will thus have to wait till the currently running task finishes.

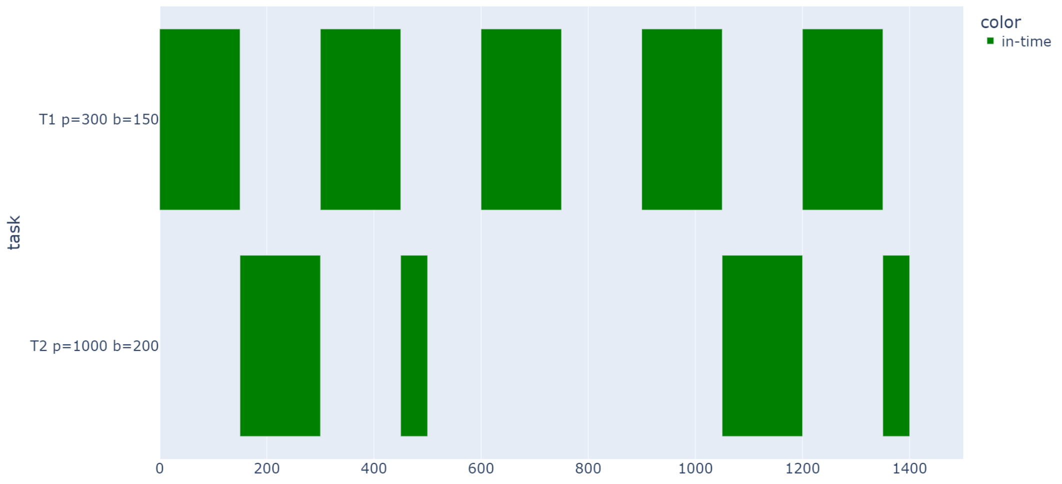


FIGURE 2.1: Example schedule by the preemptive EDF algorithm with two tasks on the system. The period of a task is indicated by p and the execution time by b. At time-point 300, an example of preemption occurs, where task `T2` is preempted to give `T1` the ability to run. On the x-axis the time in discrete time-points is shown.

2.2.2 Earliest Deadline First

EDF is a deadline-driven scheduling algorithm where full utilization of the processor can be achieved. [17] Hence, this is an optimal algorithm when the utilization of the processor is considered, as Liu and Layland [17] write:

> *... if a set of tasks can be scheduled by any algorithm, it can be scheduled by the deadline driven scheduling algorithm.*

The execution order in EDF is based on the deadline of a task, making it thus a dynamic scheduling scheme. A task will increase in importance as it is approaching its deadline.
EDF can be both preemptive and non-preemptive. In the preemptive case, it manages to keep one invariant: at each moment the running task will be the 'ready' task with the nearest deadline. So each time when a task becomes ready, a schedule decision will be made. If the task has a nearer deadline, the task that is running will be preempted by the scheduler and a context switch will occur. Once the more urgent task has finished, the previous task can proceed (unless another more urgent task is ready at that moment).

A simple example of a schedule with preemptive EDF can be seen in Figure 2.1. Two periodic tasks are running on the system, as shown on the y-axis. Task `T1` has a period of 300 cycles and an execution time of 150 cycles. Task `T2` runs every 1000 cycles and has an execution time of 200. The deadlines for the periodic executions are defined to be at the end of the periods. On the x-axis the time is expressed in

time-points. The time-point 0 can be considered the start of the system, at that moment both tasks `T1` and `T2` are ready to run. At each time-point , the EDF policy will prioritize the task with the nearest deadline. `T1` will thus start running at time-point 0.
`T2` is at time-point 150 the only ready task on the system and will thus be able to run. At time-point 300, task `T1` becomes ready (new period) with a deadline at time-point 600. This is nearer than the deadline of task `T2` (time-point 1000), so it will be preempted. At every moment, the running task will be the task that has the nearest deadline. If in EDF no task is running, this means that no task is ready. This is the case between time-point 500 and 600.

It is important to mark that the EDF policy is unpredictable when the system is overloaded. Once a deadline is missed, this can generate a cascading effect of tasks missing their deadlines. [6] A last note is that in literature, in the context of EDF, sometimes the term 'priorities' is used. However, this does not refer to static priorities based on business value or other degree of importance, but to a degree of urgency based on the deadline of the task. The scheduling policy will 'prioritize' a task with a near deadline over a task with a far deadline.

2.3 Schedulability analysis

If it can be guaranteed that, on a system for a given task set, all tasks executions will finish before their deadlines, then the task set is said to be schedulable. [28] Schedulability analysis is the domain that tries to understand and predict if a given task set is schedulable. It analyzes whether the system will satisfy its real-time requirements when deployed. It is both important to run the tasks at the correct rate and to finish the executions timely. [6]

To be able to do this analysis, the system has to be clearly specified, as Burns and Wellings [6] note:

> *It is difficult to undertake this analysis unless the details of the proposed execution environment are known.*

In practice, as done in [17][6], a set of characteristics is identified and documented. First of all it is important to specify which scheduling policy is used. Then, assumptions are made about the type of tasks. An analysis could for instance assume that all tasks are independent and periodic with their deadline at the end of the period. Lastly, the type of information that is needed from a task should be determined. This data could for example consist of a periodicity and a Worst Case Execution Time (WCET). The WCET of a task is an estimated upper bound on the execution time of that task. [27]

2.3.1 Acceptance test

To assess if a system is schedulable, a schedulability test can be employed. For example, Liu and Layland [17] derive a condition for a task set to be feasible with the preemptive EDF approach. If there are m tasks on the system, a valid schedule will exist if

$$\frac{b_1}{p_1} + \frac{b_2}{p_2} + ... + \frac{b_m}{p_m} \leq 1 \tag{2.1}$$

with b_i the WCET and p_i the period of the task i. [17]

A schedulability test can be used to decide whether to accept an additional task on the system. In the case of the test by Liu and Layland, the equation 2.1 would be evaluated for a system with $m + 1$ tasks. If the result is still schedulable, the new task can be accepted. When the test is used to decide acceptance, it is called an acceptance test.

2.3.2 Sufficiency, necessity and sustainability

In the area of schedulability tests, the following important terms are often used. A first important concept is sufficiency. If the acceptance test is sufficient, no task set exists that will be accepted while being unschedulable.
Related to this is the concept of necessity. A test is necessary if schedulability of a task set implies acceptance of that set. So, no sets exist that are schedulable, but not accepted by the test. If a test is both necessary and sufficient, it is exact.
Finally, a test can be sustainable. Burns [6] defines sustainability as follows:

> *A scheduling test is said to be sustainable if it correctly predicts that a schedulable system will remain schedulable when its operational parameters 'improve'.*

With 'improve' is meant that a task could for example have a bigger period length, leading to less executions of that task. It is however not generally true that the system will remain schedulable, although intuitively the requirements on the system seem to be relaxed.

2.4 Open systems

Real-time systems can be opened to different stakeholders (called software providers) to deploy their applications. Nevertheless, these stakeholders do not necessarily trust each other, thus some degree of security and isolation should be foreseen. TEE architectures, like Sancus, achieve integrity and confidentiality for applications, without relying on additional software (e.g., an OS). Aion builds on top of TEEs and adds a trusted software component to provide availability guarantees to applications.

2.4.1 Sancus

Sancus is a security architecture that tries to achieve integrity and confidentiality by properly isolating different applications. It reduces the TCB to be only the hardware. [21] This means that a deployed module only needs to trust the hardware, device drivers (if used) and its own code for security. Moreover, it can be attested to the SP that a certain module runs uncompromised:

> *Sancus can remotely attest to a software provider that a specific software module is running uncompromised, and can provide a secure communication channel between software modules and software providers. Software modules can securely maintain local state, and can securely interact with other software modules that they choose to trust.* [21]

To obtain these security guarantees, a module can be deployed as an enclave. Hence, authentic execution of that application is ensured.

2.4.2 Aion

However, in the presence of a misbehaving enclave, Sancus and other TEE architectures cannot ensure availability. Alder et al.[1] describe the Aion architecture as a way to ensure availability for these 'open systems'. Aion is a hardware-software co-architecture that combines the security offered by TEEs with availability and real-time guarantees. As in Sancus, the TCB is kept as small as possible, by only requiring an application to trust its own code, the device drivers (if used), the hardware and a trusted scheduler. Aion can be implemented on any TEE architecture that provides memory isolation, that does not reset or block on a violation and where the hardware is cleared of any cryptographic state before an interrupt is handle. [1]

Aion provides progress and availability by combining two concepts: bounded atomicity and preemptive multitasking. On the one hand, a '`clix`' instruction is introduced, giving a program the ability to disable interrupts for exactly x cycles. Hence, an application can make progress even if a lot of interrupts are received. To prevent attackers from disabling the interrupts for too long, an upper-bound on the number x is chosen. Additionally, it is not allowed to do nested `clix`. If an attacker tries to breach the rules, an atomicity violation will be triggered.

On the other hand, the trusted scheduler will be responsible of providing fair scheduling. Therefore, an attacker must not be able to influence the scheduling process. The scheduler thus runs inside an enclave and can be attested to ensure that it is uncompromised. Furthermore, it will run on every interrupt and violation. This is necessary for the scheduler to keep control and schedule a new application if needed. To prevent the scheduler from being interrupted itself, it can disable interrupts directly, without issuing a `clix`. [1]

With the aforementioned building blocks, Aion can provide integrity, confidentiality as well as availability. The system can ensure a deterministic activation latency to applications. This makes real-time guarantees possible, as long as the number of applications on the system is bounded. The Infrastructure Provider (IP) is responsible for setting this bound and enforcing it, as noted by Alder et al.[1]:

> *Any platform owner that accepts a new application to be deployed to the open system must check that the requirements of the new application do not exceed the capabilities of the available shared resources.*

2.5 Logic modeling

Expressing a system in logic, makes it possible to formally proof properties about the system. In this thesis, logic modeling is used to perform schedulability analysis. More specifically, a SMT solver is employed to reason about the constraint model of the system.

2.5.1 Logic

De Moura and Bjørner [10] define logic as follows:

> *Logic is the art and science of effective reasoning.*

With logic, conclusions and consequences can be drawn from a given set of facts. Following de Moura and Bjørner, each logic has a syntax, semantic and proof system.

Different branches of logic exists. A first example is propositional logic, that uses the logical symbols `or`, `and`, `not`, `implies`... in combination with propositional variables (e.g., p, q, r...). Logical symbols have a fixed interpretation, which is defined by the semantic of the logic. The propositional variables need to be given an interpretation. In propositional logic, the possible values are *true* and *false*. [10]

A formula is said to be satisfiable if for some instantiation of the variables, the formula becomes true. This instantiation is a so-called 'model' for the formula. If no model exist, then the formula is unsatisfiable. An example of a formula is the following one:

$$(p \wedge q) \vee (p \wedge \neg q) \tag{2.2}$$

For formula 2.2, there exists a model, namely $p = true$ and $q = true$. Consequently, this formula is satisfiable. There exist another model, namely $p = true$ and $q = false$. If for all possible instantiations of the variables, the formula is true, then it is said to be valid. An example of a valid formula is :

$$p \vee \neg p$$

Whatever value is assigned to p (true or false) the formula will always be true. The negation of such a valid formula will also always be unsatisfiable, and vice-versa. [10]

2.5.2 SMT

For most branches of logic, strategies exist to check for satisfiability of a certain formula, or to find models for that formula. For Propositional Logic (PL) this are commonly called SAT solvers. Whereas SAT solvers could only check satisfiability for PL formulas, SMT generalizes the SAT-solving approach to a broader range of logic. SMT solvers are able to check satisfiability for formulas containing predicate symbols, function symbols, quantifiers, logical connectives and variables. What is allowed depends greatly on the capabilities of the SMT-solver. This is achieved by combining multiple solvers for these different logics together.

> *Satisfiability modulo theories (SMT) generalizes boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories.* [9]

2.5.3 Z3

In this thesis, the Z3-solver is used. It was developed at Microsoft Research[1] and first released in 2007. The Z3-solver makes use of the SMT-LIB2 language. This standard defines the syntax and semantics of the language that is used to interact with the SMT tools [3]. Nevertheless, Z3 also provides an API for integration with general-purpose languages, like Python.

Z3 allows encoding of First-Order Logic (FOL) formulas. It combines the capabilities of different solvers together by using an approach that is called Model-based Theory Combination. If the formula F is expressed using different theories T_i, then this approach will start by searching for a model for one of the theories T_j. Next, the combination component will check if all other theories agree with this model. If a contradiction is found, it will backtrack and search a different model for T_j. [8]

2.5.4 Decidability

As stated by de Moura and Bjørner [10], Z3 can decide whether quantifier-free formulas are satisfiable. However, from the moment on that quantifiers are used, it is not necessarily decidable whether a formula is satisfiable. If the formulas expressing the system constraints, are in the undecidable part of logic, then the prover will not be able to reason about satisfiability. For example, combining linear arithmetic with quantifiers will in general be undecidable. Luckily, there are subcategories of logic that are known to be decidable [15]. The way in which the system constraints are encoded has also a large impact on decidability:

> *It turns out that the exact encoding of a given problem can have a very significant impact on the ability to solve it...*[15]

Note that the choice of the theorem prover itself has no impact on whether the problem is decidable or not. Decidability is dependent on how the constraints are expressed in logic, and which combination of theories is used to do so.

[1]https://www.microsoft.com/en-us/research/project/z3-3/

Chapter 3

Problem Statement

This chapter starts with discussing some related work. Then the execution environment will be described, along with the proposed approach for providing real-time guarantees to applications. The capabilities of the attacker will be outlined, followed by the goal of this thesis. The chapter ends by reporting the used research methodology.

3.1 Related work

In the area of real-time systems, not all security architectures do consider availability in scope [12][14]. However, there are different approaches that do achieve availability and provide real-time guarantees.
First, mixed-criticality systems are an established way to ensure availability for applications with different security requirements. Each application has a criticality level assigned to it, that determines the level of safety assurance that is needed. [5] However, no guarantees are possible in the presence of a strong adversary that can add applications to the system.
Masti et al. [19] discuss trusted scheduling, where a hardware scheduler component is used to ensure availability of the applications. They assume the set of applications to be predefined. In contrast, in this thesis, the set of applications is allowed to change during the life-time of the system.
Finally, in [4], Brasser et al. discuss TyTAN, a security architecture that allows real-time guarantees for mutually distrusting stakeholders. However, they do not limit the number of applications on the system. Furthermore, tasks cannot perform atomic sections, thus no availability can be ensured in the case of an adversary that can generate interrupts.

3.2 System model

The system, one computing node, hosts different tasks with real-time requirements. This will be referred to as the ‘task set’ on the system. The EDF policy defines an order on the execution of these tasks according to their deadline requirements.

The execution environment is based on Aion. As mentioned in Section 2.4.2, Aion integrates a trusted scheduler to guarantee availability. In this thesis, the scheduler and hardware of Aion are slightly adapted to make more precise guarantees possible (see Appendix A).

3.2.1 Tasks

Different SP can upload their software[1] as enclaves on the node. Then, they can submit task instances that will execute the software. Tasks use the Microcontroller Unit (MCU) and can handle interrupts. Each task will have a contract associated with it. This contract is a set of run-time requirements on the system. In this thesis, tasks are assumed to be periodic, to have a clear WCET and to be independent.

Contract

When submitted by the SP a task will have some meta-data with it (called its contract). It contains information about the requirements of that task and can be interpreted as an agreement between the SP and the system. The meta-data contains a number of measurable metrics, namely periodicity and run-time properties (WCET), and is used by the system to schedule the tasks. A task will always need to have the possibility to use at least all the resources it has specified, although it should be prevented from consuming more. Hence, if a task goes out of bounds, the system will need to act to prevent the task from harming the availability of the other tasks in the system.

Task configuration

To be able to analyze the schedulability of task sets on the system, some assumptions are made about tasks and their execution. These assumptions are bundled in 'task configurations'. A task configuration defines the system features that a task could use and how they can be used. Furthermore it defines what the contract should at least embed. It is assumed that the system can enforce the assumptions, preventing tasks from breaking them. To illustrate that the assumptions are realistic, Appendix A lists possible enforcement mechanisms for the configurations considered in this thesis.
In Chapter 6 the strict and very restrictive 'base' task configuration will be investigated. Some of the restrictions of this 'base' configuration will be released in Chapter 7.

All task configurations in this thesis have the following assumptions in common (can be released in future work: Chapter 9):

[1]The software of benign tasks is assumed to be thoroughly tested and thus bug-free. No guarantee can be given to a task with for example a memory management vulnerability. An attacker could just take over control of the application by exploiting this vulnerability.

- Tasks are periodic: Each task i has a **single** period p_i. A periodic run of a task will be called its 'periodic execution'. A task i should have one periodic execution each p_i time-points . If a period of a task i starts at t_0, then the following period will start at $t_1 = t_0 + p_i$.
- Hard deadlines: Each deadline will be assumed to be of vital importance and must not be missed.
- No (static) priorities: No priorities are assigned before the start of the execution of the system.
- Tasks are independent: Independent data and functionality is assumed. Tasks will have no dependencies on other tasks, for example common data or an execution order that has to be respected.
- Tasks cannot start new tasks.
- The contract can be enforced. Every entry in the contract can be monitored and a task will be unable to act differently than is stated in the contract. This means that a `clix` will run the declared number of time-points . If the contract does not contain all the required information, then the task is rejected by default.
- The deadline matches the period end: A tasks periodic execution can start at the beginning of its period and has to end before the end of that period.

3.2.2 Scheduler and acceptance test module

In the approach proposed in this thesis, adding a new task to the system happens in three steps (as shown in Figure 3.1). First, a SP wants to submit a new task and sends it with its contract to the system. It is assumed that at that moment, the needed software is already uploaded securely on the system (using the TEE architecture). The acceptance test component will decide if the task can be accepted, based on the contract (WCET, period, ...). Secondly, if the task passes the test, it will be enqueued and be scheduled together with the other tasks using the scheduling policy. Third, in a final step the application can attest to its SP that it is running with the provided requirements.

Note that there is a difference between *submitted* and *accepted*. A task is *submitted* if it has been sent from a SP to the system, but has not yet been accepted. A task is *accepted* if it was submitted and then passed successfully the acceptance test.
Both the acceptance test and the scheduler are special tasks. The scheduler task will have the characteristics as defined in Aion. It will evaluate the scheduling policy and perform context switches, therefore the scheduler will have a non-negligible run-time. It is assumed that there is an upper bound (WCET) on the run-time of the scheduler. The acceptance test will be a task without real-time requirements and will be scheduled by the scheduler in the slack time. This prevents submission of tasks from impacting the system's schedule.

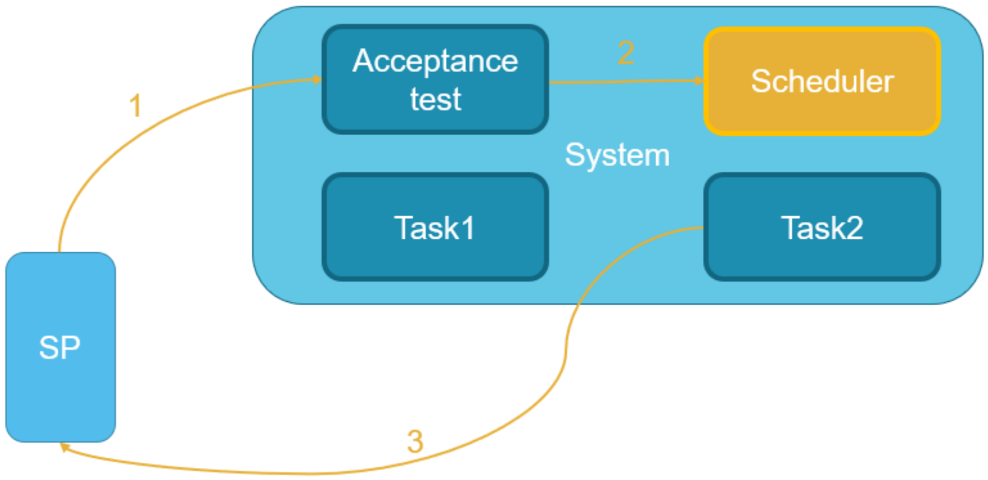


Figure 3.1: System with acceptance test. The SP submits new task (1) to the acceptance test. If it is accepted, it will be enqueued and scheduled by the scheduler on (2). Finally, a task can attest to its SP that it is running with the required guarantees (3).

3.3 Attacker model

We will consider an attacker that wants to break the real-time guarantees of a specific task on the system. Integrity and confidentiality can already be ensured by using a TEE as in Aion [1]. She can submit tasks with various characteristics, to reach her goal. She knows the system and will make use of all the possible features attempting to break availability guarantees.
First of all, it is assumed that there is no limit on the number of tasks an attacker can submit. Therefore, it is possible that in the worst case, only one benign task[2] is present on the system, and n attacker-tasks.
Secondly, the attacker is assumed to be in complete control of all software outside of the TCB of the benign task. She has also control over the communication channel to the device (as described in [18]). However, she cannot perform a Denial of Service (DoS) attack on the communication channel.
Thirdly, she can try to invalidate her contract, by performing `clix` of a longer length than declared, or trying to nest `clix`.
Finally, she can also generate interrupts.

[2]If there are only attacker tasks, then there is no need for availability.

3.4 Goal

The goal is to be able to give a SP the possibility to formulate real-time requirements for tasks and to ensure that the system can meet the requirements. The task should still function as intended, even in the presence of n attacker-tasks. Availability in this case means that the system offers to the tasks the possibility to use all the resources mentioned in their contract and will guarantee that these resources are available. To achieve this, the system has to ensure two things:

1. The task will be able to use the resources that it was promised timely (before some given deadline). The system has to schedule the tasks in a manner that makes this possible.

2. Tasks have to be prevented from using more than was described in their contract. Otherwise the system is not predictable anymore.

Enforcement mechanisms will ensure the second property. The scheduling policy (EDF in this case) will make it possible to achieve these first property, but only if the system is not overloaded. To ensure availability it should thus be prevented that no valid schedule exists, in other words the task set should be schedulable.
To achieve schedulability, the acceptance test task will be used. As explained in Chapter 2, an acceptance test will check if the worst-case behavior of the system is still acceptable (i.e. all deadlines are met). [6] If the system remains schedulable, the task is accepted on the system, otherwise it is rejected. An acceptance test can be for example a mathematical formula (as in [17]), but other forms are possible too.

The thesis will handle the following three topics:

- How can schedulability analysis be employed to provide availability in the presence of an attacker? How does the execution environment impact this analysis?

- How do acceptance tests need to be designed to guarantee schedulability of tasks? How can be ensured that the test is sufficient?

- How can we adapt the acceptance tests to allow more complex scenarios?

3.5 Methodology

To investigate schedulability in the context of Aion, the following methodology was used. First of all, a literature study on Aion and on schedulability analysis was conducted. Secondly, a more in depth understanding of scheduling policies in the context of Aion was desired. Therefore we implemented a simulator that would make it possible to simulate the scheduling of task sets on the Aion architecture. Thirdly, research on a first acceptance test was conducted, by using a manual approach and proving the sufficiency of the test by hand. Finally, this approach was refined, by using an SMT-solver to formally prove sufficiency of an acceptance test. The

SMT-solver makes it possible to guide the iterative refinement to find an acceptance test.

Chapter 4

Scheduling

This chapter will go more in depth on scheduling and the difficulties that arise when bounded atomicity comes into play. The insight into this ‘tricky’ situations will make better understanding of the acceptance test design process possible.
The chapter begins with describing some modifications made to the EDF algorithm in the context of Aion. Then an introduction is given to a visualization tool that is developed in this thesis. The timelines it produces will be discussed, to familiarize the reader with the type of figures that are used throughout the rest of the text. Finally, a selection of peculiarities of the scheduling policy is discussed, by considering a system with an attacker. The attacker will try to breach the system and corrupt availability. To illustrate the different possibilities, a lot of figures are used.

4.1 EDF revisited

As discussed in Chapter 2, when a scheduling decision has to be made, EDF will choose the task with the nearest deadline. If a preemptive version of EDF is used, the running task can be ‘interrupted’ by a task with a nearer deadline. In the context of Aion however, bounded atomicity implies that some parts of executions will be atomic. Surprisingly enough, the addition of bounded atomicity does not require any changes on the EDF algorithm itself. Scheduling decisions will just be restricted to happen outside of `clix` sections. Moreover, bounded atomicity has no impact on the deadline of a task. Note that in general the system won’t remain schedulable if `clix`-sections are used (see Section 4.3).
Scheduling decisions will occur at three occasions. First of all, when a task finishes its (periodic) execution, another task can be scheduled. Secondly, a scheduling decision should be made right after the end of the `clix` section of the currently running task. This handles the case where a more urgent task became ready or an interrupt occurred while performing the `clix`. Finally, if a task becomes ready then a scheduling decision should be made (if no task is issuing a `clix` at that moment).

4.1.1 Discussion

In this thesis, EDF seemed the most appropriate scheduling policy for the following reasons.
First of all, the scheduling policy should be dynamic. In an open system, tasks come in at run-time and a pure static policy would thus be inappropriate.
Secondly, to have systems that are reliable without sacrificing too much efficiency, a high processor utilization is desired. The EDF algorithm is one that can achieve this.
Third, most of the tasks requirements can be modeled as deadlines. If a deadline can be attributed to a task, then the EDF algorithm will naturally work on this task.
Fourth, a lot of research has been done on scheduling policies and on EDF in particular (see Chapter 2). So challenges of this policy are well-understood.
Finally, it is assumed that all the deadlines are hard. This means that it is critical to not miss any. As long as no deadline is missed, the EDF policy will work fine (no cascading effect).

4.2 Simulating policy

It is difficult to fully grasp the behavior of a scheduling policy without examples. To ease the understanding of the implications and peculiarities, this thesis presents a simulator for scheduling policies in the context as stated in the Problem Statement (see Chapter 3). It is implemented in Python and simulates the policy as well as the hardware and scheduler specifics needed to evaluate this policy. Scheduling decisions will only be made at the moment the scheduler runs. The scheduler is completely event-driven and will thus only run if some task ends, comes in, or an interrupt occurs. This means that the system should ensure that the scheduler runs when needed (as stated in the previous section), by scheduling additional interrupts. In this thesis, it is assumed that the system has this capability.

The simulator takes as input a task set for the system and outputs a timeline representing a part of the schedule. To generate the schedule, it takes the relevant details of Aion [1] into account and simulates the processor utilization during a specified time-span. The task set is a specification of the task flow on the system. This means that a number of tasks is given, together with their contract.

The code for the simulator is made publicly available, and can be found in Appendix B.1.

4.2.1 Design choices

The choice has been made to keep the simulator as simple as possible. It is mainly meant to be a tool for a first insight into the system. So the possible features of tasks may seem overly restrictive. However, this simplicity is also an advantage, because it makes the timelines easy to understand and interpret.

The simulator only allows task sets containing periodic tasks. The tasks can specify their contract consisting of a period and a WCET (also called budget). Next, some run-time characteristics can be provided, that represent a simplistic view on program execution. This executions can consist of two types of statements. On the one side there are `clix` instructions, that take one cycle and start an atomicity section. An argument should be provided, namely the number of cycles to `clix`. On the other side *calculation* sections are present, that can take an arbitrary number of cycles. The scheduler is assumed to have some fixed run-time, its WCET. It will schedule interrupts after each `clix` section to regain control, and will terminate tasks if they try to go out-of-bounds. Finally, in the simulator tasks are allowed to end their `clix` section before the requested length, because this makes the simulator more flexible. This seems a logical choice, but it should be handled with care (see Section 4.3.2). In real system implementations it is better in that case to make the scheduler fill up the remaining `clix` time.

4.2.2 Example schedule

The timeline is a visualization of the processor utilization for a given scheduling policy, in this case the EDF policy. On the timelines a lot of information is displayed, so an example is helpful to clarify the different aspects of the drawing (see Figure 4.1). The information found on the timeline is threefold. First, there is information about the tasks, namely their contract (period and WCET). Secondly, the schedule is shown with the tasks on the vertical axis and the time-points on the horizontal axis. Finally, the task executions are shown in green, blue or red. This color indicates if they respectively ended in-time, missed their deadline or performed some violation and thus are terminated.

The different tasks and features relevant to scheduling are placed on the vertical axis. For the timeline 4.1 there are four components: two tasks `T1` and `T2`, the scheduler task and the interrupts. On the horizontal axis the time is set out, measured in processor cycles. Time-point 200, for example, indicates that by then 200 cycles have passed. The time a task is running on the processor is indicated by the colored boxes. For task `T1` for example, it is running from timepoint 0 to 200. At the end of a (partial) run of a task, the scheduler task will run to choose which task to run next. After `T1` finishes, the scheduler thus runs for 20 cycles, and chooses `T2` to run next. `T2` runs from 220 to 420, followed by a scheduler run and so on. A note that can be made about this chapter and the rest of the thesis, is that the order of magnitude of MCU cycles is in no way representative for real-life cases. This is done for the purpose of clearness, readability and understandibility. This is however not a problem, because scaling up the number of cycles does not change the difficulties associated with the schedulability.

Because the interrupts also impact when scheduling decisions are made, they are displayed on the visualizations. On this graph, there are two types. The green color is used for the sleep interrupts, that indicate the beginning of a tasks period

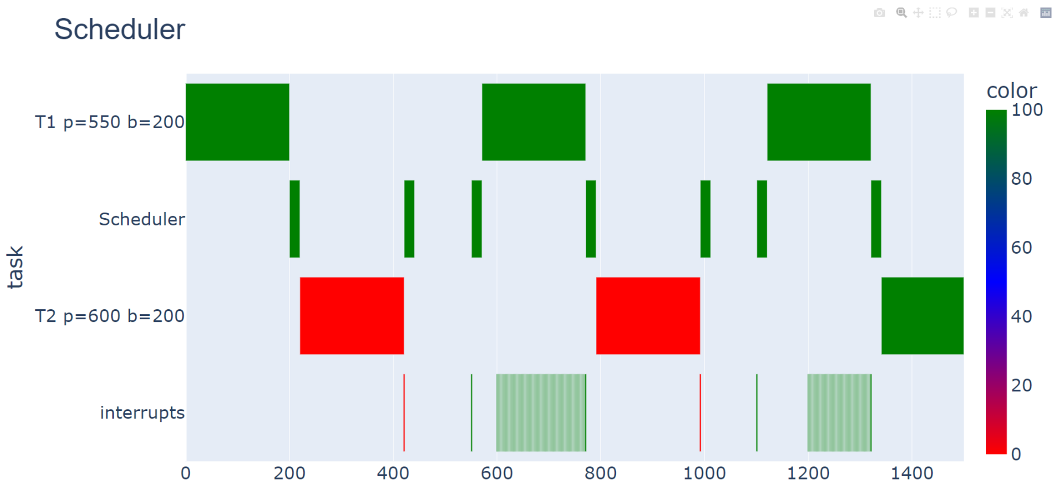


Figure 4.1: Schedule of two tasks to illustrate interpretation of timeline.

(becomes ready). The budget interrupts (red) are the other type, they occur when a task tries to violate its contract.

4.2.3 Example table

The schedule shown in Figure 4.1 however doesn't provide some necessary details about the tasks. The contract is given, but the tasks internal characteristics are completely hidden. The contract that is shown on the visualization is typically what the system (and thus the scheduler) knows about the task. This suffices for the scheduler to do its job. Nevertheless, to be able to interpret the visualizations better and deeper, a description format for the underlying task behavior would be interesting. In the following sections this behavior will be set out in a table. An example is shown in table 4.1 and can be interpreted as follows. Task `T1` first issues a `clix` to start an atomicity-section of 200 cycles. Then it will do some calculations in the 199 cycles that last from its budget (while the `clix` is done). An important note is that the `clix` instruction itself is taken as part of the bounded atomicity region... Task `T2` on the other hand, just does some calculations that would take 300 cycles to complete. But the contract of `T2` (see Figure 4.1) only mentions 200 cycles of budget. So that explains why the task is interrupted before finishing and colors red on the schedule visualization. As such, the tables will be a tool to understand where the scheduler or hardware are acting to prevent tasks from going out-of-bounds.

4.3 Attacking the schedule

Now that the visualization tool has been introduced, it can be used to illustrate different scenarios where an attacker disrupts availability. It turns out that a lot of

Task T1		
Instructions		
Instruction	nr_of_cycles	param
Clix	1	200
Calculation	199	/

Task T2		
Instructions		
Instruction	nr_of_cycles	param
Calculation	300	/

TABLE 4.1: Table for the example task configuration in Figure 4.1.

corner cases and unexpected problems can arise when scheduling. An attacker will try to find vulnerable situations and exploit them by influencing the schedule. In this section, task T1 will be a benign task with real-time guarantees. All other tasks can be under attacker-control.

4.3.1 Scheduling

Tasks can make use of bounded atomicity to ensure their progress, by issuing `clix` instructions. In this subsection, some difficulties arising from the integration of `clix` into EDF are highlighted by looking at how an attacker could take advantage of this non-preemptive sections. Next to that, the scheduler needs to determine the scheduling order and perform a context switch if needed. The scheduler will therefore have some latency and the attacker can also try to use this fact. Finally, the cascading deadlines effect in EDF will be illustrated.

Preemption

The benign task T1 with period 200 is running on the system, making use of a `clix` section of 100 cycles. To make a comparison possible, the attacker does not use atomicity sections in this case. In Figure 4.2 the resulting schedule is shown. All executions of T1 have finished in-time. In the schedule can be seen that task T2 is interrupted a few times while executing, and is re-scheduled at a later moment in time. All the partial runs form one periodic execution of T2 (the highlighted region in the schedule).

Bounded atomicity

Now, the same tasks as in the previous paragraph are considered, but the attacker can also use bounded atomicity. T2 will issue a `clix` equal to its run-time (see Table 4.2). When the EDF schedule is generated for this adapted task set (Figure 4.3), T1 misses deadlines. The sleep-interrupt that comes at time-point 200, is not

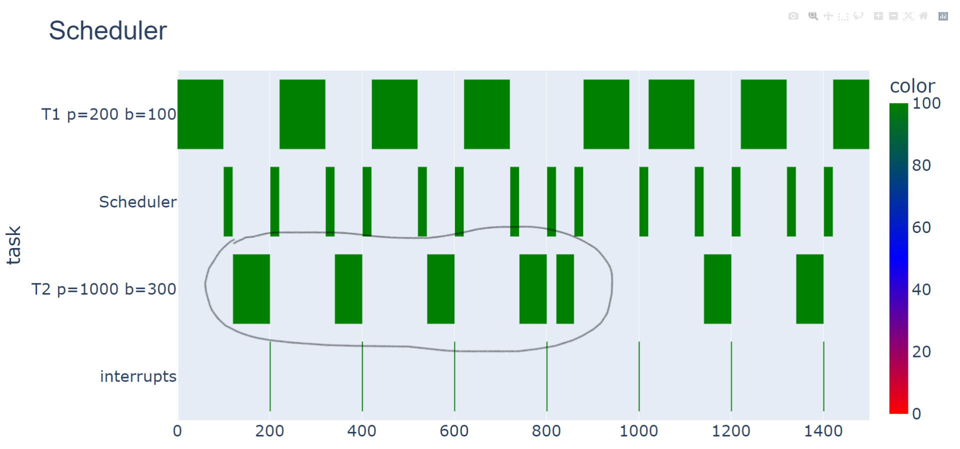


Figure 4.2: Schedule with tasks T1 and T2 that do not have atomicity sections. The highlighted parts are one periodic execution of task T2.

Task T1		
Instructions		
Instruction	nr_of_cycles	param
Clix	1	100
Calc	99	

Task T2		
Instructions		
Instruction	nr_of_cycles	param
Clix	1	300
Calc	299	

Table 4.2: Both tasks perform a `clix` section where calculations are done.

handled (because of atomicity) before time-point 400. Hence, the periodic execution for task T1 starts too late (already after its deadline in this case). This shows that a lot of care has to be taken with bounded atomicity. If a system is preemptively schedulable, this will not mean it is still schedulable if `clix`are allowed. It is important to note that task T2 does nothing illegal, it does not violate its contract. It is the combination of the tasks with their periods and budgets that create a problem.

Scheduler latency

Scheduling new tasks, handling interrupts and switching between tasks take time on the MCU. In the previous scheduling visualizations this has been illustrated by the

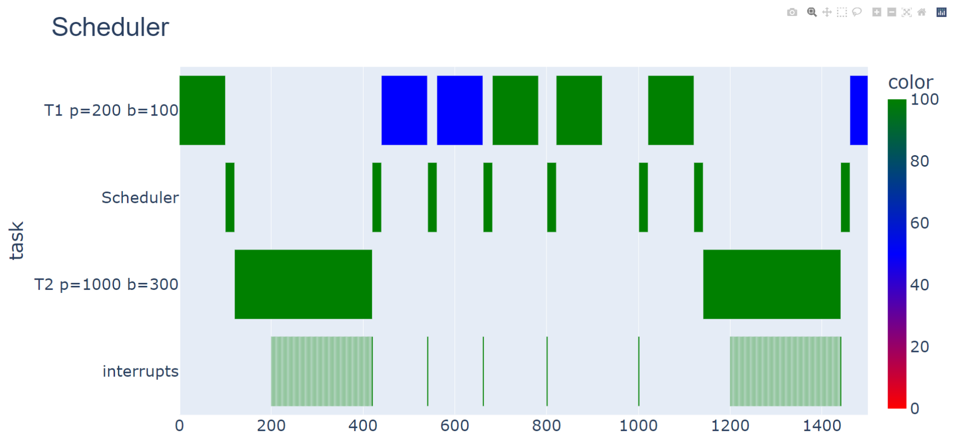


FIGURE 4.3: Schedule with bounded atomicity. T1 misses some deadlines, because of the `clix` section of T2.

scheduler run-time on the system. In this paragraph the effect of this latency will be discussed. It is assumed that there is some upper bound (WCET) on the scheduler latency and that the latency will always be exactly that upper bound.
The fact that there is some scheduler latency, gives a powerful tool in the hands of the attacker. By creating situations where the scheduler is triggered very often, she could subvert availability. To achieve this, the attacker submits a task to the system with a very short period. In Figure 4.4 the schedule of a task set is shown where T1 does not use `clix` sections. Task T2 has a very short period and thus scheduling decisions will have to be made very often. Task T1 and task T2 both start to miss their deadlines, resulting in a complete system degradation. This is because the scheduler itself already consumes a good part of the available processor time, creating an overload on the system.

Overloading the system

Finally, an attacker can also just overload the system by submitting a task. In Figure 4.5 a schedule is shown for a task set containing two tasks T1 and T2. The tasks meet both all their deadlines.
However, by just submitting a small task T3, the attacker succeeds in overloading the system, see Figure 4.6. If the utilization of the MCU is too high, then obviously some deadlines will be missed. This example also illustrates the cascading effect of EDF. When task T3 finishes after its deadline, this produces a cascading effect that ends by having even task T1 missing its deadline (for the execution at time-point 1400). So by having one deadline missed, all three tasks miss deadlines. This is an illustration of the fragility of EDF. An extremely good processor utilization can be achieved, but there is a cascading effect when a deadline is missed.

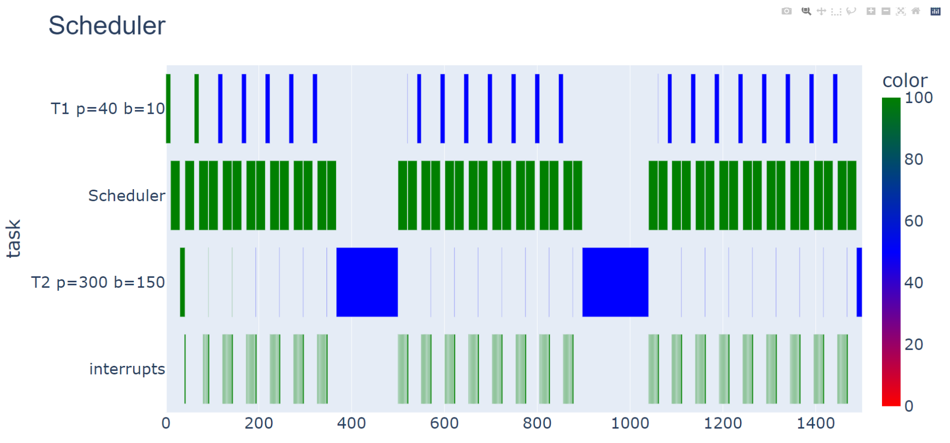


Figure 4.4: Schedule with a scheduler latency of 20 cycles.

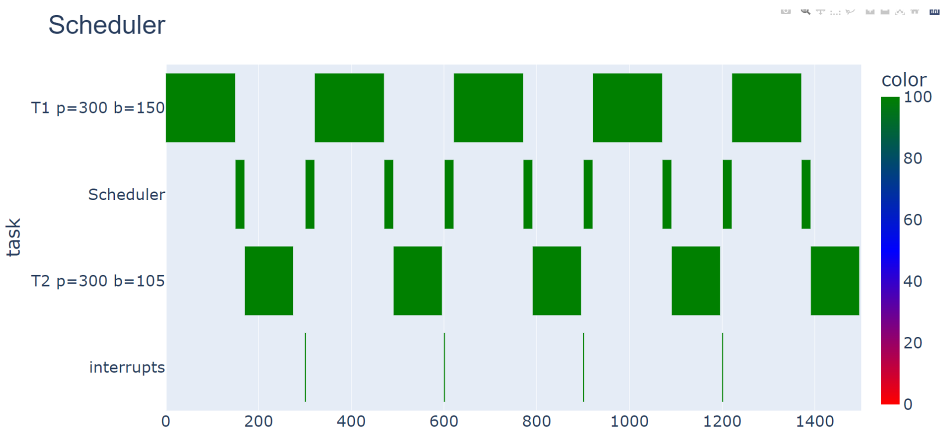


Figure 4.5: A system with two tasks T1 and T2, no deadline is missed.

4.3.2 Other vulnerabilities

As shown in the previous examples, providing availability guarantees is not self-evident. An attacker has a lot of possibilities to make the system go wrong, and from the point on one deadline is missed, the real-time guarantees are breached. This section discusses further possible ways to to abuse the features that the system offers. This problems definitely have to be taken into account when making statements about the schedulability of a system.

Clix ending earlier than expected

Atomicity sections definitely impact severely the schedule, making scheduling with bounded atomicity far from trivial. It turns out for example that often task sets with bounded atomicity are not sustainable. If the requirements of one task are weakened,

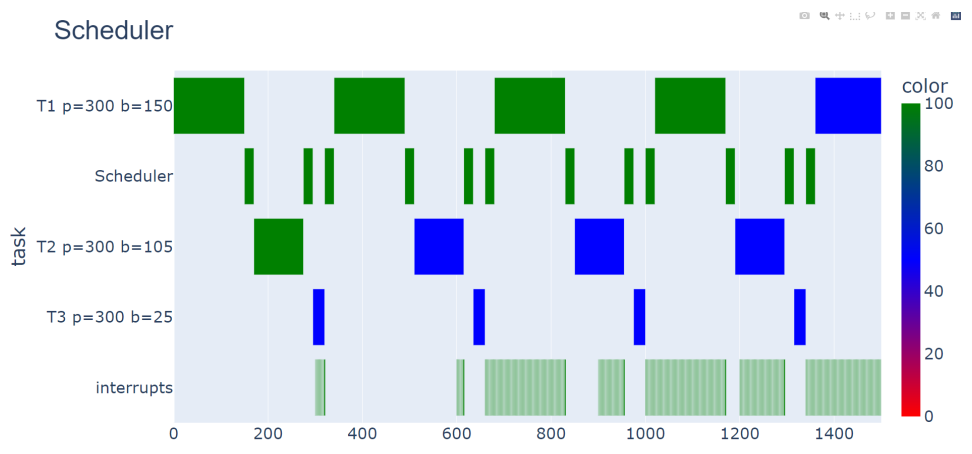


FIGURE 4.6: The same system as Figure 4.5 with an additional task T3. It is lightly overloaded, resulting in all three tasks missing deadlines (cascade effect).

then it is not necessarily true that a valid schedule will exist.
This situation occurs for example if `clix` are allowed to end earlier than expected. One should expect that if one task ends its `clix`prematurely, the following task could already run and this would benefit the whole system. However, this is not the generally the case. To illustrate this, Figure 4.7 shows a schedule for the task set where all `clix` are executed to their end. All the tasks meet their deadlines. If the attacker-task T2 in contrast would end its `clix`earlier, then T1 suddenly misses a deadline. This situation is shown in Figure 4.8. The reason for that is that normally, task T1 would already be able to run if T2 finishes as expected. By finishing beforehand, task T2 release the MCU before task T1 is ready again. Following the EDF policy task T3 will be scheduled and will start to run. Since T3 runs a `clix`-section, T1 will not be able to start immediately at the beginning of its period. This results in T1 starting too late and thus going over deadline.

Influencing the scheduler latency

As illustrated in the previous paragraph, the precise timing of the start of one `clix` can have impact on finding a valid schedule. An attacker can break availability by ending its `clix` section prematurely, but also by manipulating the scheduler latency. In Section 4.3.1, the scheduler latency was assumed to be held constant. In this paragraph, it will shortly be illustrated what happens if this can be influenced by choices of the SP. If the scheduler latency is not artificially kept constant, then it is dependent on the tasks being switched, the number of interrupts present on the system and the number of tasks on the system. In that case the same situation as above could be obtained by making the context switch shorter than its WCET.

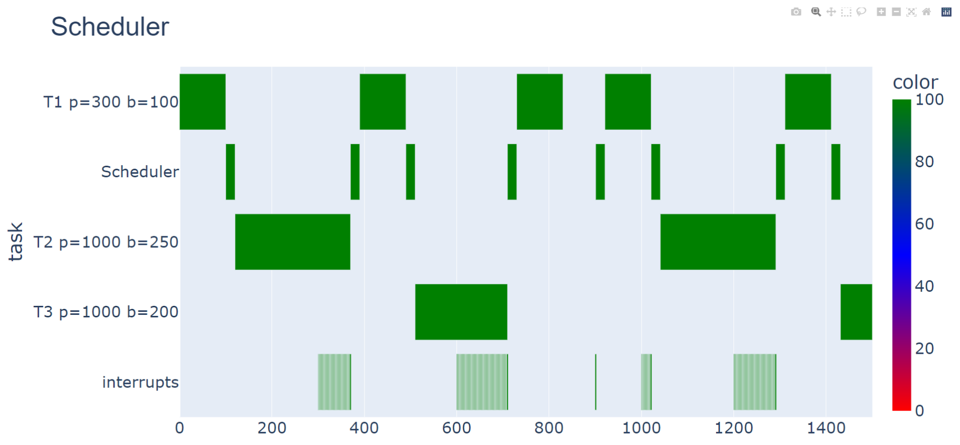


FIGURE 4.7: All the tasks run their `clix` to the end. This system is schedulable.

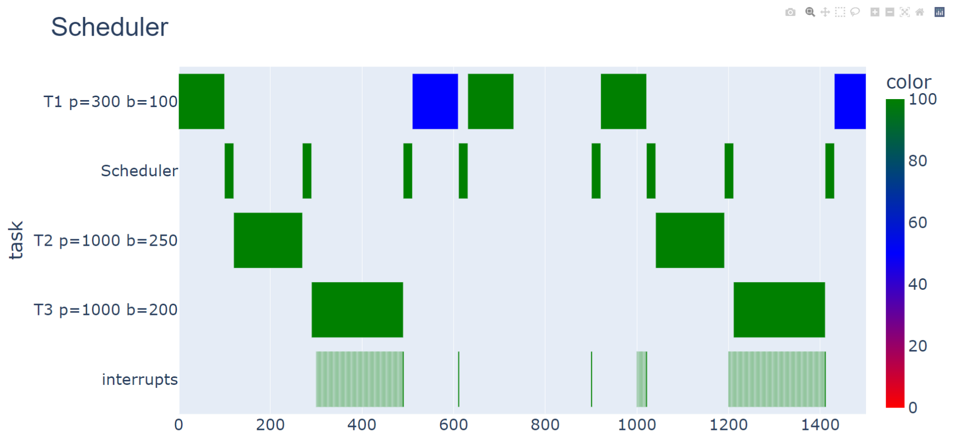


FIGURE 4.8: Same configuration as in Figure 4.7, but now task T2 ends its `clix` after 150 cycles instead of 250 cycles.

Clix longer than target period

A last example of an attack, could be by issuing a `clix` that is longer than the period of a task. The system could for example contain a critical task that has to execute often, and thus will have a short period. In Figure 4.9 a scenario is shown where the attacker has submitted its task T2 with a `clix` length of 300 cycles, while the period of T1 is only 200 cycles. For its second execution, task T1 is only scheduled after the run of task T2, when it's deadline is already missed.

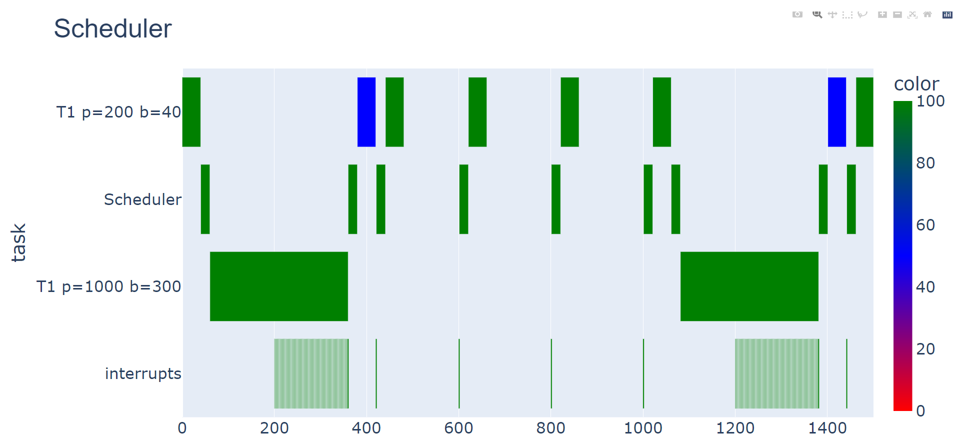


FIGURE 4.9: Schedule where Task T2 has a clix-section that is longer than the period of task T1. Task T1 misses deadlines.

Chapter 5

Manual schedulability analysis

Chapter 4 illustrated that an attacker has a multitude of ways to break availability guarantees for benign tasks on the system. She submitted carefully crafted tasks to the system, that would make it unschedulable with EDF. To prevent the attacker from harming availability, these tasks should be rejected. This chapter will perform schedulability analysis to find an acceptance test, that can be used in the architecture as described in Chapter 3. The goal of the chapter is to illustrate the reasoning behind the design steps, by building on the insight of Chapter 4
Before performing the analysis, the assumptions on the system will be made explicit. Next, iterative refinement will be used to find a sufficient acceptance test. Finally, the found acceptance test will manually be proven to be sufficient. Despite being intuitive and insightful, the approach in this chapter is quite long and error-prone. In more complicate scenarios it will therefore be unusable. Chapter 6 will improve the approach by using a tool to guide the design and to prove sufficiency.

5.1 The task configuration

To make the design of acceptance test possible, a precise definition of the considered task configuration is needed. As such, the assumptions on the tasks and the attacker will be detailed.

5.1.1 Task

Some restrictions (additional to those in Chapter 3) are imposed on tasks. First of all, each task on the system is only allowed to issue one `clix` per periodic execution. The task is assumed to not do any computation outside this atomicity section. The task execution will also be assumed to last for the whole `clix`. Furthermore, the task will be allowed to specify its required `clix` length and its period, by using its contract. For now, only two tasks are allowed at the same time on the system.

5.1.2 Attacker

Since the enforcement mechanisms control the run-time behavior of a task, the attacker cannot break the assumptions stated above. Furthermore, performing violations is not beneficial because this will only make the scheduler run. An attacker can submit only one task (the benign task is assumed to be on the system already). She can choose the `clix` length and period freely.

5.2 A condition for schedulability

To be able to guarantee schedulability, the system has to assess whether a newly submitted task can be accepted or not. The first step consists of identifying an intuitive condition for a schedulable system. This is the starting acceptance test that will be iteratively refined. First, a task set is searched that satisfies the current test, but is unschedulable. Such a task set would be an attack vector for breaking availability. Then, the test is adapted, to reject this task set. The refinement step is repeated, until the resulting acceptance test only accepts schedulable task sets and thus is a sufficient condition for schedulability.

5.2.1 Processor Utilization

The processor utilization is defined as being the fraction of time that the processor is utilized. For a task with period p_i and `clix` length c_i, the fraction of time it spends on the MCU is $\frac{c_i}{p_i}$. The sum of all time fractions is the processor utilization and should thus be less than or equal to 1:

Lemma 1. *Utilization bound:* $\sum\limits_i \frac{c_i}{p_i} \leq 1$

with c_i the `clix`-length and p_i the period of task i.

This acceptance test is analog to the one by Liu and Layland [17], as shown in Chapter 2. However, it is not sufficient when bounded atomicity is introduced. If the acceptance test is inspected, it becomes clear that for a task T1 with a small period, the attack shown in Figure 4.9 can be performed. The attacker could submit a task T2 with a `clix`-length that is larger than p_1 with deadline misses of T1 as result.

5.2.2 Bound on clix length

To handle the problem, the relation between budget and period of the two tasks has to be taken into account. T2 should not be accepted if its `clix` length is longer than the period of T1. The condition should even be somewhat stronger, requiring the sum of the `clix` lengths to fit in the period of T1 ($c_1 + c_2 \leq p_1$). Nevertheless, the opposite should not be true either (in the general case both tasks distrust each other). So T1 should also have a bound on its `clix` length. The additional constraints are thus as follows:

Lemma 2. *Bounds on `clix` length:*

$$c_2 \leq p_1 - c_1$$
$$c_1 \leq p_2 - c_2$$

Under the stated assumptions the combination of 1 and 2 will guarantee that an attacker cannot break the guarantees of task T1 while the two tasks run on the system. However, the attacker can still make the system unusable, right after deployment. It can be that in the beginning no tasks are present on the system. If the attacker submits her task at that moment (before T1 is submitted), she could choose $p_2 = c_2$. With the equations above, this would mean that the system is already fully utilized, so no additional task can be accepted.

5.2.3 Underbound period, upperbound `clix`

This attack can be easily countered by adding an absolute under bound on the period (`MIN_PERIOD`) and an absolute upper bound on the `clix` length (`MAX_CLIX`). It is of course system-dependent what bounds will make sense. The `clix` upper bound should be at least smaller than the under bound for the period, to prevent an attacker from requiring $p_2 = c_2$. Lemma 3 is thus introduced:

Lemma 3. *Absolute bounds, where it is assumed that* $\mathtt{MAX\_CLIX} < \mathtt{MIN\_PERIOD}$ *:*

$$c_1 \leq \mathtt{MAX\_CLIX} \qquad p_1 \geq \mathtt{MIN\_PERIOD}$$
$$c_2 \leq \mathtt{MAX\_CLIX} \qquad p_2 \geq \mathtt{MIN\_PERIOD}$$

At this point, it seems that a good acceptance test has been found, in the sense that it will be sufficient. It is the conjunction of Lemmas 1, 2 and 3. If a task set following the assumptions in 5.1 satisfies this acceptance test, a valid schedule should thus be obtained.

5.3 Proving sufficiency

However to be certain that sufficiency holds, the statement "Acceptance implies schedulability" should be formally proven. This section will walk through a possible proof for this acceptance test.

There are two tasks T1 and T2 with `clix`-lengths c_1 and c_2, and periods p_1 and p_2. Before running on the system, the tasks have to pass the acceptance test. It should be proven that for all situations with two or less tasks running on the system, a valid EDF schedule will exist.

If no tasks are present on the system, it is trivially schedulable and thus the statement holds. If only one task T1 is present on the system, then from Lemma 3, $c_1 < p_1$ holds. It is clear that T1 should thus meet its deadline. If task T2 is the only task on the system, the reasoning is analogous.

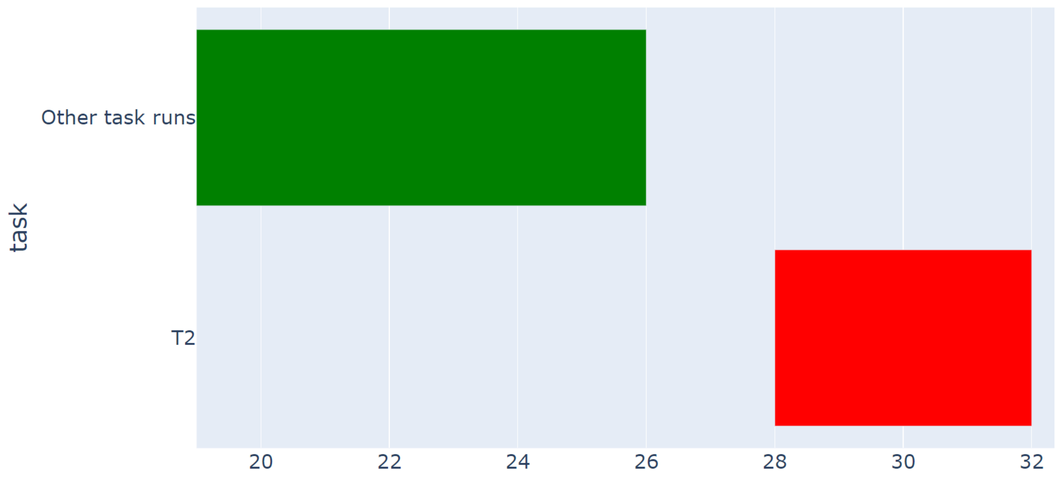


Figure 5.1: There is some gap in the schedule before T1 starts running.

Now remains the case with two tasks on the system. This case is proven by contradiction: It is assumed that the two tasks have satisfied the acceptance test, but a task on the system misses a deadline. It will be proven that this situation isn't possible.

Let's start from the following assumption:

Premise. *At some time t_s an execution of T1 starts and misses its deadline t_d. It is the first faulting execution on the system.*

The end of the execution will be at $t_e = t_s + c_1$ with $t_e > t_d$. The start of the current task period is $t_p = t_d - p_1$, since the deadline is assumed to be at the periods end. First of all, given the EDF scheduling policy, a task execution ended right before task T1 started its faulting run. From Lemma 3, $c_1 < p_1$ holds, and thus it follows that

$$
\begin{aligned}
&t_d - p_1 < t_d - c_1 < t_e - c_1 \\
&t_p < t_s
\end{aligned}
\tag{5.1}
$$

This means that task T1 started some time after the beginning of its period, because another task execution prevented T1 from running earlier. A gap in the schedule, as shown in Figure 5.1, is not possible when using EDF.

The situation must therefore be as in Figure 5.2. There are two possibilities. The task execution before the faulting run can be either another execution of T1 or an execution of T2. Now it will be shown that both possibilities end up contradicting the premises of the proof.

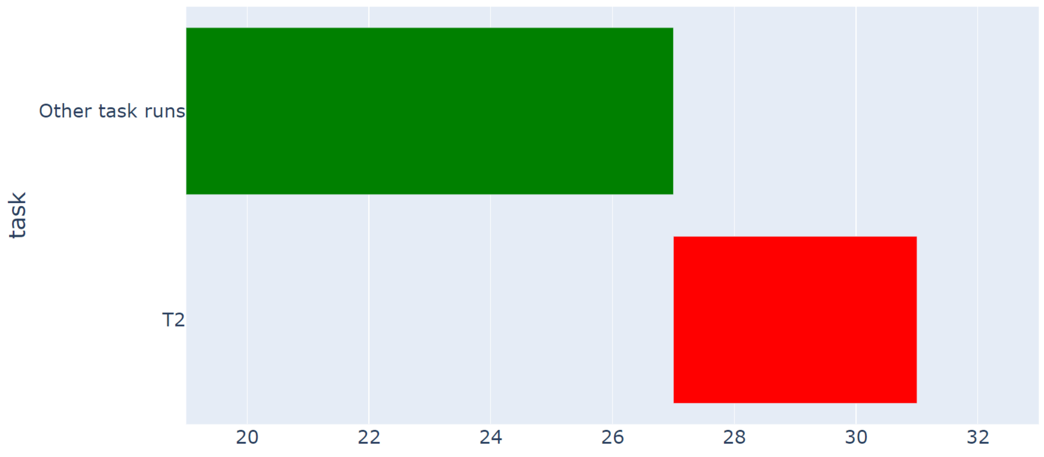


Figure 5.2: T1 is missing a deadline

If it is an earlier execution of T1, then this must have finished in time, according to the premise. Its deadline is the end of its period, or equivalently the start t_p of the current period. So this means that the current execution of T1 would have started at t_p and thus $t_s = t_p$, contradicting equation 5.1.

On the other hand, it can be an execution of T2 that corresponds to a period started at $t_{p'}$. In that case, $t_{p'} \leq t_p$ because otherwise T1 would have started running before T2. This means that T2 will have run a number of cycles $q \geq 0$ before the start of T1's new period.
Additionally, because there is no gap between the runs, T2 has thus ended at time t_s. Using equation 5.1, it is clear that T2 will have consumed a number of cycles $c_2 - q$ during the period of the faulting execution of T1. Next to that, T1 will have run itself a number of cycles $c_1 - (t_e - t_d)$ before the end of its period. This means that:

$$\begin{aligned} t_d - t_p &= c_1 - (t_e - t_d) + c_2 - q \\ &\leq c_1 + c_2 - (t_e - t_d) \\ &< c_1 + c_2 && (t_e > t_d) \\ p_1 &< c_1 + c_2 \end{aligned} \tag{5.2}$$

Equation 5.2 is in contradiction with Lemma 2, and thus contradicts the fact that both tasks must have passed the acceptance test to run on the system. This means that neither an execution of T1, nor T2 could have preceded the faulting execution of T1. T1 could thus not have missed a deadline. An analogous reasoning can be done if T2 would be assumed to miss a deadline. It can thus be concluded that if the tasks pass the acceptance test, then the system will meet its deadlines. The acceptance test is thus sufficient.

5.4 Conclusion

The method for finding and proving acceptance tests, as discussed above, is quite long, error-prone and work intensive. It puts a high responsibility on the designer. He has to carry out the refinement steps successfully and prove the acceptance test to be sufficient afterwards. The identification of the 'problematic' task sets, the task sets that are accepted but not schedulable, requires good insight into the system. Moreover, overlooking one problematic case is easily done. Finally, making a mistake in the proof could lead to false conclusions about the sufficiency of a test.
In this chapter a system containing only two tasks and a very limited number of features was considered. For more complex systems, the approach is not appropriate. As already mentioned, Chapter 6 will suggest a better approach.

Chapter 6

Guided schedulability analysis

In Chapter 5, an iterative approach was used to carry out the schedulability analysis. Each step consisted of identifying schedulability issues, by using the insight obtained in Chapter 4. As pointed out, the technique is long and error-prone, requiring of the designer to identify all the 'problematic' task sets. The task sets that are accepted by the acceptance test but unschedulable, are called problematic. This chapter introduces a more structured approach for carrying out the refinement steps, based on formal modeling.

6.1 The general idea

The approach uses formal modeling to guide the design of acceptance tests. For this goal, the system is modeled in FOL and the resulting constraint model is used with an SMT-solver (Z3). The solver makes it possible to identify the task sets that are currently faultily accepted, i.e. accepted while being unschedulable. This will be used to support the designer in the refinement steps. Furthermore, it will be more easily to prove that an acceptance test is sufficient, by checking that no problematic task sets can be found for that test.

6.1.1 The task configuration

To illustrate the new modeling approach, this chapter will discuss a simple task configuration. This configuration will be restrictive and easy to understand, to make it possible to carefully state the different concepts associated with the design approach. Note that this task configuration will consist of the same assumptions as these made in the Aion[1] architecture. In Chapter 7 more complex task configurations will be discussed.

Task

In this very simple task configuration, tasks are assumed to have the same contract. First of all, each task on the system is assumed to issue one `clix` of a fixed length (`CLIX_BOUND`). This bound is set by the system administrator. All computation is assumed to happen inside the atomicity section. The task execution will be assumed to last for the whole `clix`. Secondly, the period of the task is fixed too. Some default system value `DEFAULT_PERIOD` is used. Compared to Chapter 5 this task configuration is a simplification where all the tasks on the system have the same period and `clix` length.

Attacker

Since the enforcement mechanisms control the run-time behavior of a task, an attacker cannot break the assumptions stated above. In this configuration, the attacker will not be able to use a custom period and `clix` length, as these are only allowed to have the fixed default values. However, she still can submit multiple tasks.

6.1.2 The constraint system

The constraint model of the system consists of a set of constraints that precisely reflect the possible ways a system can evolve. In this case three types of constraints are needed to express the system.
Firstly, there are constraints that are related to the physical boundaries of the system. On the one hand, a task cannot run before the MCU is started (before time-point 0). On the other hand, only one task can use the processor at a time. Secondly, the assumptions of the given task configuration have to be expressed. Lastly, the scheduling policy also puts some constraints on the system and the schedules that can be generated by that system. The EDF-policy embeds two base rules. First, at each moment in time, it should be that either a task is running or no task is ready at that moment (the MCU is idle). Second, if a task starts running, it will be by definition the task with the earliest deadline.

6.1.3 Checking sufficiency

The obtained constraint model can be used to guide the acceptance test design and prove sufficiency of the resulting test. This is done by checking if acceptance implies schedulability for all possible task sets in this task configuration. The following formula can thus be checked on satisfiability in the constraint model:

$$\texttt{accepted}(set) \Rightarrow \texttt{schedulable}(set) \tag{6.1}$$

with *set* referring to the task set. If the implication holds, the acceptance test is sufficient, because any task set that is accepted, will be a schedulable task set.
If the implication does not hold, then the acceptance test has to be refined by using the constraint model to find problematic task sets. These task sets are accepted, but are not schedulable:

$$\texttt{accepted}(set) \land \neg\texttt{schedulable}(set) \tag{6.2}$$

The solver can be used to check equation 6.2 on satisfiability. It can be used as well to generate problematic task sets, by searching for satisfying models for this formula.

6.2 Modeling in Z3

In this thesis, the Z3 SMT-solver is used. This section will shortly discuss how the constraints are encoded to be decidable. The difficulty in encoding the system, is to express time. It is infinite and it will need to be quantified over. A first approach is to encode time as an integer and use functions to map tasks to their characteristics. However, this is an encoding that needs quantifiers over the time (an unbounded integer), and is undecidable.
To end up with a decidable encoding, the time variable should be bounded. This can be done by observing that, in this task configuration, all periods will be synchronized. If a schedule is found for one period length (`DEFAULT_PERIOD`), then by replicating this schedule, a valid EDF schedule for unbounded time is obtained. This means that proving schedulability in the time window implies schedulability of the system. The time frame considered in the encoding, can thus be restricted to one `DEFAULT_PERIOD`. A similar time frame will be called an Observation Window (OW) in what follows.

6.2.1 Modeling using matrices

The decidable encoding makes extensively use of matrices and models time implicitly. The schedule is modeled as a matrix. The rows of that matrix are the tasks in the system. The columns in the matrix are the considered time points. The OW determines how much time points are considered, and makes it possible to have a finite matrix. Each cell in the matrix registers (by a boolean value) whether the task runs at that time point. By having a finite matrix representation of the problem, the modeling is part of the decidable part of logic. This constraint modeling can thus be used for further investigating acceptance tests.

The code for this modeling technique could be found in Appendix B.2.1.

6.2.2 Results

The design of acceptance test starts (as in Chapter 5) by formulating a first intuitive condition for a task set to be schedulable. The utilization bound can again be used (see Lemma 1 in Chapter 5) as a first condition. In this case however, the equation can be simplified, because the `clix` length and period are equal for all tasks. If n is the number of tasks on the system, then the acceptance test is the following:

$$n * \texttt{CLIX\_BOUND} \leq \texttt{DEFAULT\_PERIOD} \tag{6.3}$$

Note that the only variable in this equation is the number of tasks.
Now that a first condition is phrased, sufficiency can be checked. When checking the formula 6.1 with the equation 6.3 as acceptance test, the Z3 solver returns `sat`. This means that the implication is satisfiable and it can thus be concluded that the acceptance test implies schedulability.
Note that the correctness of the constraint model should be thoroughly checked.

Otherwise sufficiency of an acceptance test could be faultily concluded.

6.3 Conclusion

In this Chapter, an SMT-solver is used to guide the design process of acceptance tests for new task configurations. The solver could furthermore be used to prove the sufficiency of an acceptance test. As was already pointed out, the considered task configuration corresponds to the Aion[1] architecture. It is thus no surprise that the equation 6.3 corresponds to the upper bound on tasks in Aion. This upper bound is set by the system administrator to guarantee a maximum response time. In this configuration, each task is guaranteed to have a maximum response time of `DEFAULT_PERIOD`.

Chapter 7

Analyzing extended configurations

In Chapter 6, the guided design approach is discussed for a simple configuration. In that task configuration, the period and `clix`-length of tasks is some fixed, default value. In addition, a task only consists of one single atomic section.
In this chapter, the approach using Z3 will be further used to search acceptance test for more elaborate task configurations. These configurations will each relax assumptions about tasks, that were made in the previous chapter. By allowing more complex task sets on the system, the following sections will thus illustrate the wider applicability of the approach.

7.1 Clix of different length

In the previous chapter, it was assumed that all tasks would issue `clix` of a given fixed length. Now, all tasks will still have the same, fixed period `DEFAULT_PERIOD`. The OW will therefore stay the same. Nevertheless, the tasks are no more assumed to have all the same `clix` length. Each task will be allowed to perform `clix` with a duration ranging from 1 to a maximum value. In the following, this maximum value is fixed to be the `DEFAULT_PERIOD`. This means that in this case, each task will specify in its contract the `clix` length that it needs to fulfill its functionality. This adaptable `clix` length goes beyond the enforcement capabilities of Aion, but can be enforced by using an adaptable `clix` hardware upper bound (see Appendix A).
An attacker can submit multiple tasks, as before. She can also specify the `clix` length for each task.

7.1.1 Changes in model

To adapt the previous model to the new configuration, the following adaptations are made. A new list is introduced, containing variables that represent the `clix` length for each task. These variables do not have an a priori value. Every formula containing the `DEFAULT_CLIX` is rewritten to use these custom `clix` lengths.

Next, an additional constraint is added to the system, to reflect the allowed `clix` lengths (as defined in the previous paragraph). A `clix` should be bigger than 0 and smaller than the `DEFAULT_PERIOD`. This is the only new constraint. The new model can be found in Appendix B.2.2.

7.1.2 Acceptance test

To design the acceptance test, the utilization condition (Lemma 1) can again be used as a starting point. Note that all the tasks have the same period `DEFAULT_PERIOD` and thus the equation can be simplified:

$$\sum_i c_i \leq \texttt{DEFAULT\_PERIOD} \qquad \text{with } i \in \text{Tasks} \tag{7.1}$$

This acceptance test expresses that the sum of the `clix` lengths should fit inside the default period. This acceptance test can be proven sufficient, by using the formula 6.1. In that case, the SMT-solver gives `sat`, indicating that all accepted systems will have a valid schedule. Therefore, no additional refinement step is needed.

Note that, in the present case, the acceptance test is not only sufficient, but also necessary. To be a necessary condition for a task set to be schedulable, schedulability of task set should imply that the task set is accepted.

$$\texttt{schedulable}(set) \Rightarrow \texttt{accepted}(set) \tag{7.2}$$

In other words, there should be no schedulable task sets in this configuration that are rejected by the acceptance test. When the solver is used, it gives `sat`, meaning that the implication indeed holds. The acceptance test 7.1 is both sufficient and necessary, thus exact. An exact test will always preferred, because it will not rule out schedulable task sets.

7.1.3 Remarks

The present scenario realizes a more efficient system than the one in Chapter 6, because a SP can more precisely specify the needs of its task (more elaborate contract). The acceptance test in Chapter 6 will only allow a fixed amount of tasks on the system, given a default period and `clix` length established by the system administrator. There, it does not matter if a task is really short and does not need the full default `clix` length. This is clearly inefficient. The new system with the acceptance test 7.1, will in general be able to allow more tasks, because a task can specify in its contract the needed `clix` length. A task with a small WCET, can specify a short `clix` length and will therefore have a high probability of being accepted, even if the system is already heavy loaded.

7.2 Different periodicity

This task configuration will again extend the configuration of Chapter 6, by releasing the assumption that all tasks have the same period. For simplicity, the `clix` lengths will again be assumed to be equal and fixed to a constant `CLIX_BOUND`, in contrast to previous section. Section 7.3 will combine both extensions.
Each task will be allowed to specify its periodicity requirement. The fixed period `DEFAULT_PERIOD` will thus be replaced by task-specific periods. These period lengths should be at least 1. This makes different timeliness requirements possible, compared to Aion where one latency bound is enforced for all tasks. Here, a task with low latency requirements can require a short period, where tasks with less stringent requirements can be attributed higher period lengths, making the system more flexible and efficient.
An attacker is able to submit multiple tasks and can specify the periodicity for each submitted task.

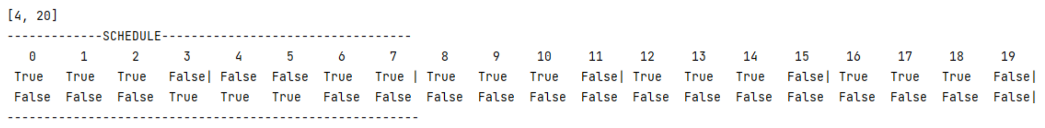
7.2.1 Changes in model

The most important change is related to the OW. In this task configuration, the periods are different, so the reasoning in Chapter 6 to find the OW is not suitable. An appropriate value for the OW can be found by observing that all the periods should coincide with the end of that time-frame. If a schedule is found for a time-frame with that property, it can be replicated to form a valid schedule for unbounded time. The smallest value for which this property holds, is the Least Common Multiple (lcm) of the task's periodicities.
The Z3-encoding does require the OW to be specified as a constant, because it is used in the list comprehensions. In the encoding, the OW is thus fixed beforehand, and the tasks periods are restricted to be divisors of the OW. This means that sufficiency is proven by the model, for one specific OW value. To be certain the acceptance test is sufficient in general, the model should be evaluated for every possible OW.
What remains is to adapt the previous constraints to this new configuration. The resulting code is attached in Appendix B.2.3. To make the list comprehension operations work, an additional constant `MAX_RUNS` was introduced. This constant has no impact on the model or the validity of the proofs, but ensures that the list comprehensions have a deterministic length.

7.2.2 Acceptance test

As in the previous cases, Lemma 1 is used to start the design process. In this configuration, the `clix` lengths are equal and fixed (`CLIX_BOUND`), and the equation reads:

$$\sum_i \frac{\texttt{CLIX\_BOUND}}{p_i} \leq 1 \qquad \text{with } i \in \text{Tasks} \tag{7.3}$$



```
[4, 20]
-------------SCHEDULE---------------------------------
  0      1      2      3      4      5      6      7      8      9      10     11     12     13     14     15     16     17     18     19
 True   True   True   False| False  False  True   True | True   True   True   False| True   True   True   False| True   True   True   False|
 False  False  False  True   True   True   False  False  False  False  False  False  False  False  False  False  False  False  False  False|
-------------------------------------------------------
```

Figure 7.1: An example of an unschedulable task set that passed acceptance test 7.3 for a default `clix` length of 3 cycles. The vertical bars are used to indicate the periodic deadlines. Task 1 misses the second periodic deadline, only running for two cycles before the end of the period.

When this acceptance test is checked in Z3 by using formula 6.1, `unsat` is returned. This means that the acceptance test 7.3 is not sufficient. A refinement step will thus be needed. Formula 6.2 is used to generate a problematic task set, an unschedulable set that passed the acceptance test.

The solver was run for task sets with a `CLIX_BOUND` of three. In the Figure 7.1, two tasks are shown with respective period lengths 4 and 20. These tasks indeed validate equation 7.3, but the task set is unschedulable. It can be seen that in its second period, task 1 only runs for two cycles, instead of three.

This example shows the added value of the modeling technique as a way to guide the acceptance test designer. The solver makes it possible to highlight the shortcomings in the test. Here, the problem is related to having one task with a short period, hardly bigger than the default `clix` length. This is similar to the issue in Paragraph 5.2.2 where an additional constraint on the relation between the `clix` lengths and periodicities was introduced. This approach can be used here too, by generalizing Lemma 2 to task sets with n task, a sufficient acceptance test can be found for this configuration. Because the `clix` lengths are fixed and equal to `CLIX_BOUND`, the equation can be simplified:

$$n * \texttt{CLIX\_BOUND} \leq p_i \qquad \forall\, i \in \text{Tasks} \tag{7.4}$$

The solver is used with Formula 6.1 and gives `sat`. It can thus be concluded that equation 7.4 is a sufficient acceptance test.

7.2.3 Remarks

The acceptance test 7.4 is sufficient, but is not necessary. This means that some task configurations that are schedulable, will nevertheless be rejected. The approach used for the previous section, can be used here to search for an acceptance test that accepts more task sets, but is still sufficient.
The OW makes this case very interesting, because there is a relation with the different periods used in the system. The acceptance test 7.4 is only proven to be sufficient for a given OW. Probably, it will be safe for all OW sizes, because it doesn't depend on it, but that's not completely certain. To ensure that there is no exotic, problematic case, all possible integer values should be checked, but that's not feasible. Two possible

scenarios can occur. Either the system administrator chooses a fixed OW, and the SPs have to submit tasks with periods that are divisors of the OW. Or the OW is adapted dependent on the current task set by taking the lcm of the periods on the system. This gives attackers an attack vector to attack the system, by searching for periods that would make the OW very large. However, an attacker can only attain the availability of the acceptance service, and not tasks that are already running on the system.

7.3 Different periods and clix of different lengths

Finally, this section will investigate a combination of releasing the period constraint and allowing for `clix` of different lengths. The model for Section 7.2 is taken as starting point and adapted to embed different `clix` lengths. A task will thus specify, in its contract, the required period and `clix` length. This configuration is equivalent to the one used in Chapter 5, except that now task sets with n tasks are considered. An attacker can submit multiple tasks as before. Additionally, she can specify for each task a different period and `clix` length.

7.3.1 Acceptance test

We start the design approach with an utilization based acceptance test (Lemma 1):

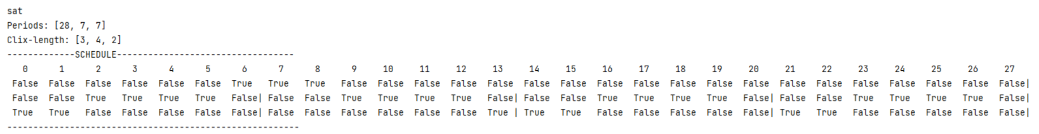
$$\sum_i \frac{c_i}{p_i} \leq 1 \qquad \text{with } i \in \text{Tasks} \tag{7.5}$$

This is however not sufficient, because this does not rule out task sets where the `clix` length for task i is longer than the period of task j. This situation was discussed in Section 4.3.2. There are thus combinations of periods and `clix` lengths that would give unschedulable situations.

To rule out these problems, the following equations should be added to the acceptance test:

$$c_i \leq p_j \qquad \forall\, i, j \in \text{Tasks} \quad \text{and} \quad i \neq j \tag{7.6}$$

Alternatively, a higher bound `MAX_CLIX_BOUND` on the allowed `clix` length and a lower bound `MIN_PERIOD` on the period can be set. If the minimum period is higher than the upper bound on `clix` length, then this will imply the equations 7.6. In this case, the acceptance test will be the conjunction of equation 7.5 and 7.6.
Nevertheless, when this acceptance test is checked on sufficiency, by checking the satisfiability of Formula 6.1, then `unsat` is returned. By using the solver on Formula 6.2, a task set is found that is unschedulable, as shown in Figure 7.2. At time-point 6, Task 1 starts running, followed by Task 2 and too few time is left for Task 3 to finish its execution. This is again something that is not easily found by hand, but that Z3 can point out.



```
sat
Periods: [28, 7, 7]
Clix-length: [3, 4, 2]
-------------SCHEDULE----------------------------------
   0      1      2      3      4      5      6      7      8      9     10     11     12     13     14     15     16     17     18     19     20     21     22     23     24     25     26     27
 False  False  False  False  False  False  True   True   True   False  False  False  False  False  False  False  False  False  False  False  False  False  False  False  False  False  False  False|
 False  False  True   True   True   True   False| False  False  True   True   True   True   False| False  False  True   True   True   True   False| False  False  True   True   True   True   False|
 True   True   False  False  False  False  False| False  False  False  False  False  False  True | True   True   False  False  False  False  False| True   True   False  False  False  False  False|
-------------------------------------------------------
```

Figure 7.2: Unschedulable task set for the acceptance test composed of equation 7.5 and 7.6. Task 3 does not finish its periodic execution before the deadline at time-point 14.

This is similar to the situation discussed in Paragraph 5.2.2. Here, the problem occurs (see Figure 7.2) because the processor cycles in a period of an intermediate task i are consumed by the slowest task, followed by tasks that are more urgent than i. Too few cycles are left for the intermediate task to perform its execution timely. To account for this, Lemma 2 could be generalized. This new equation is added to the previous equations to form the final acceptance test:

$$\begin{aligned}
\sum_i \frac{c_i}{p_i} &\leq 1 && \forall\, i \in \text{Tasks} \\
p_i &\geq \texttt{MIN\_PERIOD} \\
c_i &\leq \texttt{MAX\_CLIX} \\
\sum_{j,\, p_j \leq p_i} c_j + \texttt{MAX\_CLIX} - 1 &\leq p_i && (7.7)
\end{aligned}$$

Note that for the equation 7.6, the alternative approach has been used. Equation 7.7 is the newly added equation. Finally, the solver can again be used to check for sufficiency. In this case `sat` is returned, proving sufficiency.

7.4 Other scenarios by reduction

The previous sections focused on modeling the extensions explicitly and designing an acceptance test specific for that task configuration. Much effort was put in extending the model for the new task configurations. Acceptance tests have thus been established for these 'fundamental' task configurations. However, it is possible to generalize the results to a broader set of configurations, by reducing new configurations to one of the fundamental configurations. A reduction from configuration A to configuration B is defined to be a transformation from a task set in configuration A to a task set in B by setting assumptions on the execution of this task set by the scheduler.
If the assumptions can be enforced, then the acceptance test of configuration B can be used on the transformed task set. This approach will be illustrated for three extensions and can be used to find acceptance tests for other extensions too. The benefit of this approach is that no new constraint model has to be built to find a sufficient acceptance test.

7.4.1 Scheduler latency

A first simple extension could be to take into account a scheduler run-time. It will be assumed that the scheduler executes within some higher bound (b_{sched}, its WCET). The task configuration also allows different `clix` lengths c_i and different periodicities p_i. It will be called `SchedLat`.
In the assumption that the scheduler will run before each `clix`, a reduction is possible from the `SchedLat` configuration to the configuration in Section 7.3 (called `DiffPerClix` for clarity). A task set s in the `SchedLat` configuration can be transformed to an equivalent task set s' in the `DiffPerClix` configuration. Each task in s' has a `clix` length that is equal to $c_i + b_{sched}$, where c_i is equal to `clix` length of the corresponding task in s. The acceptance test for `DiffPerClix` can thus be used on the transformed task set s'. This results in a sufficient acceptance test for the `SchedLat` configuration:

$$\sum_i \frac{c_i + b_{sched}}{p_i} \leq 1 \qquad \forall\, i \in \text{Tasks}$$

$$p_i \geq \texttt{MIN\_PERIOD}$$

$$c_i + b_{sched} \leq \texttt{MAX\_CLIX}$$

$$\sum_{j,\, p_j \leq p_i} (c_j + b_{sched}) + \texttt{MAX\_CLIX} - 1 \leq p_i$$

7.4.2 Multiple atomicity sections

For the moment each task could only perform one `clix` per period. This can however be relaxed to r_i `clix`-sections of length c_i per period p_i. The parameters r_i, c_i and p_i should be part of the contract of the task. This allows much more complicate and longer tasks, that previously couldn't fit into one `clix`. However, this case is also more challenging, because a task could be interrupted between `clix` sections, and execution switched to another task. An attacker could try to use this fact to break real-time guarantees of a task, by submitting tasks with well chosen characteristics.
Also for this configuration a reduction exists to the `DiffPerClix` configuration. Each task with multiple `clix` is transformed to an equivalent task i^* with

$$p_{i^*} = \frac{p_i}{r_i}$$

and

$$c_{i^*} = c_i$$

The resulting task i^* has one `clix` per period, and thus the acceptance test 7.7, could be used:

$$\begin{aligned} \sum_i \frac{r_i * c_i}{p_i} &\leq 1 & \forall i \in \text{Tasks} \\ \frac{p_i}{r_i} &\geq \texttt{MIN\_PERIOD} \\ c_i &\leq \texttt{MAX\_CLIX} \\ \sum_{j,\ \frac{p_j}{r_j} \leq \frac{p_i}{r_i}} c_j + \texttt{MAX\_CLIX} - 1 &\leq \frac{p_i}{r_i} \end{aligned} \tag{7.8}$$

A task i will thus be assumed to execute in r_i parts, by running one `clix` per $\frac{p_i}{r_i}$ cycles. The system is assumed to enforce this property, otherwise the transformed acceptance test will not be sufficient.

Splitting the task up into n tasks could on the first sight seem to be a good approach too. However, this is not the case. Different tasks are assumed to be independent, and so the chronology of the task could be broken.

7.4.3 Interrupts

Often, it is interesting to have some form of external interrupt handling. In most cases, some latency requirement is given on how fast an interrupt-driven task should have handled the interrupt. A task should handle for example network interrupts within a latency of 1ms. Interrupt handling is difficult to model, because of the unpredictability of when an interrupt will occur.
Nevertheless, it is possible to reduce this configuration to a fundamental configuration, by modeling it as a task with p_{i^*} equal to half of the admitted latency. By taking this periodicity, it is ensured that two task executions are separated by at most the required latency. If an interrupt occurs between two task executions, it will thus be serviced timely. It is assumed that the functionality of the interrupt-driven task will fit into one `clix`. In that case the acceptance test 7.7 could again be used.

Chapter 8

Real Life Example: Smart Metering

In the previous chapters, acceptance tests have been discussed as part of the approach (see Chapter 3) to provide availability guarantees. Now, it can be interesting to look at a real-life example, where availability guarantees are needed. The considered application is smart metering. The European Parliament [11] defines a smart meter as follows:

> *'smart metering system' or 'intelligent metering system' means an electronic system that can measure energy consumption, providing more information than a conventional meter, and can transmit and receive data using a form of electronic communication;*

The deployment of smart meters is a quite recent trend, as a way to deal with the use of renewable sources and of more energy-efficient homes. As described in [24]:

> *Climate change, awareness of energy efficiency, new trends in electricity markets, and the gradual conversion of consumers towards more active agents are promoting not only the use of Renewable Energy Resources (RES), but also the Distributed Generation (DG) and Distributed Storage (DS), which urge a dramatic evolution of the actual electricity model.*

These intelligent, electronic devices make it possible to manage the distributed resources and to have a detailed view on the energy consumption (in a house for example). [16] Smart metering systems can achieve the following goals. First of all, it can make consumers more aware of their electricity consumption and of the ways to change their behavior. Secondly, fraud is a lot more difficult with these systems. Thirdly, meter readings can be done remotely, reducing considerably the cost of taking these measurements. Lastly, grid operators will have a better insight by enhanced monitoring capabilities. [25]

In what follows, a deeper dive into the security of smart meters is made from the point of view of availability. The requirements and system model for the smart metering problem are based on the paper on implementing a secure smart meter by

Mühlberg et al. [20]. They use a simplified model of the smart meter that enables a focus on the security aspects.

8.1 Security in Smart metering

Security is critical in the design of smart meters. First of all, because of privacy reasons. A lot of sensitive data is handled, that could be misused. So, integrity and confidentiality of the applications on the meter should be ensured (see [20]). Next, some services on the smart meter should be highly available. Mühlberg et al.[20] identify the load switch as such a critical service.

8.1.1 Considered use cases

The following usages of the smart meter are discussed in [20]. First of all, it carries out the billing process, by measuring the consumption and providing this data to the energy provider. Secondly, in most smart meter systems there is a load switch that can be used to uncouple the household from the power grid. Either because the consumer has exhausted it's credit (in prepaid mode), or because the load on the grid is too high and some consumers are preventively disconnected from the grid to avoid a black-out. This second use is the critical part, where fast response is primordial. Thirdly, the meter can communicate data to smart appliances by using the HAN interface. [7] This can be appliances of any kind that the customer can use to monitor its electricity usage and to react on it.

8.1.2 Confidentiality and integrity

In the case of smart metering, confidentiality and integrity can be guaranteed by using a TEE, as done in [20]. This is important to guarantee authentic execution of the software, even in the presence of an attacker. However, guaranteeing a degree of availability is not yet possible in their architecture. An application may thus be prevented running by an attacker monopolizing the MCU.

8.1.3 Availability goals

For smart metering systems, some additional degree of availability should be ensured. Mühlberg et al. [20] distinguish three important real-time properties. First, there must be information updates via the HAN gateway at least every 10s. Secondly, the device should monitor the credit balance, if prepaid mode is used. If some threshold has been exceeded, then the power supply should be switched off. Finally, any unauthorized physical tampering should result in the power supply being disabled and the system entering a locked state. This last constraint is critical, because it may harm the system badly if an attacker could circumvent this:

> *That is, severe system damage (e.g., damage to the grid or critical infrastructure, large-scale fraud) may occur when an adversary succeeds*

in physically accessing the smart meter's internals without the locked state being established. [20]

In this chapter therefore, the emphasis will be on this last requirement. It is the most critical and is safety-related. The other requirements can be analyzed analogously.
To model the system, in section 8.3, the different requirements will be embedded into different tasks. These tasks will run independently on the system. The load switch task will be the one that will have the most stringent availability guarantees. It will be shown that, in the presence of attackers, the system can ensure the task's availability requirements.

8.1.4 Threat model

The attacker is assumed to be some determined entity, that wants to undermine the load switch availability. She wants to prevent the load switch from reacting timely on the corresponding interrupts. This can be for various reasons. The attacker could be the customer, that wants to physically tamper with the system to make fraud possible. An attacker could also be some remote entity, that wants to prevent the load switch from functioning when the load on the overall grid is too high.
The attacker is able to update or change the behavior of all modules except the load switching module. The attacker can also update and change the corresponding tasks, that are running on the system. This means that the attacker can use the tasks to undermine availability of the load switch task. It is assumed that the load switching module does not contain any vulnerabilities (including memory management vulnerabilities). Other modules however can be breached and thus made to act differently than they were supposed to.

8.2 System model

8.2.1 Hardware

It is assumed that the main processor is a MSP430, that has been adapted to the Sancus architecture [21]. It is assumed that the necessary adaptations are done to support the Aion architecture [1] on top of it. The other parts of hardware and connections are assumed to be the same as in the paper of Mühlberg et al. [20].
To make the case understandable, the processor is assumed to run at a constant speed of 1MHz. This means that the device runs 1000 cycles in 1ms. The maximum upper bound on `clix` is assumed to be 50.000 cycles.

8.2.2 Remote update

Updates of the software are possible, by using the remote update interface. This happens in a number of steps. First, needed modules are submitted to the untrusted loader, and can be attested. After successful attestation, the SP can decide to submit a task to the acceptance test. The acceptance test will eventually accept or reject the task, depending if it satisfies the requirements of the test. If the updated task is accepted it will be queued for and the old task will be removed. From then on the newly requested guarantees will be ensured and the task can attest to its submitter that the required guarantees will be met. If the task is rejected, the received modules can eventually be removed.

8.3 Task model

To be able to run the applications on the system, a translation from application to tasks will be needed. The system can be implemented by using three tasks with distinct requirements: the load switch task, the credit balance monitor task and the information update task. Next to the three tasks that embed the business logic, there are two important tasks (scheduler and acceptance test) that coordinate everything. The task configuration discussed in Section 7.3 can be used to specify the tasks. Very important is the fact that all tasks will need to do their calculations inside `clix` sections. This means that the application has to be restructured to fit in the format required for tasks. Such transformation can have a non-negligible effect on the WCET of the application. In the general case, it will become larger (as shown in the next paragraphs).

8.3.1 Load switching

First, is the load switch task. Although it is specified that it should be very responsive, no precise specifications are given concerning the responsiveness [20]. Therefore in the following, it will be assumed that 200ms is the maximal allowable latency for the load switch, to react on physical tampering. This is fast enough to prevent an attacker from being able to perform meaningful adaptations to the smart meter's hardware before the locked state is entered. The load switch task is interrupt-driven, this can however be inserted as a normal task by taking as period half of the admitted latency for handling the interrupt (see Paragraph 7.4.3). This gives a period of 100ms. The WCET of this task depends on the complexity of the program, and will be assumed to be 5ms (what corresponds to 5.000 cycles). Since 5.000 cycles is smaller than the maximum `clix` length of 50.000 cycles, the task only needs one `clix` with a `clix` length of 5.000 cycles.

Behavior

In Listing 8.1, the pseudo-code for the main load switch functionality is given. The load switch will issue a `clix` of 5000 cycles, and then check in this bounded atomicity region if there has been a physical breach (should be an interrupt that is reported by the scheduler). If there has been some breach, then the load should be disabled.

```
def CheckPhysicalBreach:
        CLIX 5000
```

Contracts		
Task	Field	Value (in 1000 cycles)
Load Switch Task		
	Period	100
	Clix length	5
	Nr clix	1
Credit balance monitor Task		
	Period	5.000
	Clix length	50
	Nr clix	5
Information update Task		
	Period	10.000
	Clix length	50
	Nr clix	13

TABLE 8.1: The contracts for the different tasks on the smart meter.

```
# Check if some interrupt has occured
if is_physical_breach_or_overload():
        disable_load()
```

LISTING 8.1: Entry Point into Load Switch Module

8.3.2 Monitoring credit balance

Next is the credit balance monitor task. For this use case too, no quantitative latency requirements are specified. Here, it is less critical that the system shuts down immediately. In this case, we assume that a hard deadline of 5s is good enough. The WCET of this task is assumed to be 0.2s (200,000 cycles). The WCET of this task is longer than the `clix` length, what means that in general, the task will not be able to run in only one `clix`. It is assumed that rephrasing the task would make it fit into 5 `clix` sections of maximal length. As noted earlier, the task's WCET has become larger, because of the rephrasing that was needed. In most cases, this will involve some yielding or idling within a `clix` section, thus wasting a little efficiency.

8.3.3 Information updates

Finally, the customer should get regular updates from the smart metering, about consumption statistics etc. This has to happen each 10s at least once. This can be done by using a task with period 10s. This task will probably be more complex, and is assumed to last for 0.5s (500,000 cycles). As with the credit balance monitor task, this task also has a WCET far above the maximum `clix` bound. This means that the task will have to be partitioned into different sections. In this case, it will be assumed that this results into 13 sections of each 50.000 cycles (maximal `clix` bound). Again, the resulting task is less efficient than the original one, but availability can be guaranteed.

8.3.4 Scheduling and acceptance

Next to the three tasks that embed the main functionality, two additional special tasks are needed to make the system work. First, a scheduler task who determines which task can run next. This task will also be responsible of handling interrupts and enforcing the other task's contract. By default the scheduler task has no deadline, but will be invoked by the hardware each time an interrupt/violation occurs or a task ends its current run. The WCET of the scheduler is assumed to be 10ms (10.000 cycles). This is based on the upper bound in the Aion architecture (6.000 cycles [1]), but accounting for the additional bookkeeping that the scheduler has to do.
Second, the acceptance test is a task as well. This task has no deadline and will be run by the scheduler if no other task is ready. The acceptance test module is part of the TCB for each task, because it needs to be trusted for availability. It is an enclave and thus can be attested. The acceptance test task should not contain any `clix` instruction, to avoid it breaking guarantees for other tasks. Its WCET is not easy to determine, because it depends on the used acceptance test and also on the number of tasks on the system. The task has no deadline, to avoid it impacting the schedulability of the system. It will run in the slack time and can be preempted by the scheduler at any time.

8.4 Guaranteeing availability

There is a safety critical requirement, that the load switch should respond quickly when someone tries to physically access the device.
In the next sections, only the availability of the Load Switch task will be discussed, because of its criticality. The other modules are potentially under attacker control. The system has to ensure safety, under the stated attacker assumptions.
Mühlberg et al. [20] foresee following threats[1]:

> *First, an attacker may overwrite the system-wide Interrupt Vector Table (IVT) that records interrupt handler addresses. This can be prevented by mapping the IVT memory addresses into the immutable text section of a dedicated PM. Second, an adversary may hold on to the CPU by disabling interrupts for arbitrary long times. To prevent such a scenario, and to establish a deterministic interrupt latency, running applications should not be allowed to unconditionally disable interrupts.*

The Aion architecture already deals with these two problems as discussed in Chapter 2. On the one hand, the scheduler is in charge of handling interrupts and furthermore runs inside an enclave (its data, e.g. an IVT, cannot be overwritten).

[1]PM (Protected Module) is a synonym for enclave

On the other hand, in Aion some higher bound is set on the `clix` length, preventing tasks from disabling interrupts for long times.
Nevertheless, when dealing with hard timeliness constraints, three additional things have to be ensured by the system administrator. First of all, he has to establish that the originally deployed system is schedulable. In the benign case, all tasks have to be able to finish in time, especially the load switch task. Secondly, the scheduler and acceptance test should be checked on integrity by attesting them. Thirdly, all new tasks submissions/updates to existing code, will have to pass via the acceptance test task. By ensuring the first property, the system will start in a schedulable state. The next two properties ensure that changes in the task set on the system will preserve the schedulability. The system will thus always be schedulable, all tasks will meet their deadlines and safety is ensured.

8.4.1 Acceptance test

The acceptance test is very dependent on the assumptions on the system and mostly on the tasks it should handle. In this case, tasks can have different `clix` length, multiple `clix` sections and different periods. In Paragraph 7.4.2 an acceptance test is derived for this task configuration. However, the latency of the scheduler has to be taken into account, and thus equation 7.8 needs to be adapted analogously to Paragraph 7.4.1. This gives the following acceptance test:

$$\begin{aligned}
\sum_i \frac{r_i * (c_i + b_{sched})}{p_i} &\leq 1 && \forall\, i \in \text{Tasks}\\
\frac{p_i}{r_i} &\geq \texttt{MIN\_PERIOD}\\
(c_i + b_{sched}) &\leq \texttt{MAX\_CLIX}\\
\sum_{j,\ \frac{p_j}{r_j} \leq \frac{p_i}{r_i}} (c_j + b_{sched}) + \texttt{MAX\_CLIX} - 1 &\leq \frac{p_i}{r_i}\\
\text{with} \qquad r_i =&\ \text{the number of clix sections for task i}\\
c_i =&\ \text{the length of clix sections for task i}\\
p_i =&\ \text{the period of task i}\\
b_{sched} =&\ \text{the WCET of the scheduler}
\end{aligned}$$

8.5 Evaluation

To evaluate the effectiveness of the followed approach, it will be compared with the original approach that does not ensure availability (see [20]) and with the prototypical Aion implementation (as in [1]). It will be assumed that at some point in time, a physical breach of the system occurs and thus an interrupt is generated. This interrupt should then be timely handled by the Load Switch Task and the system should enter the blocked state.
If all tasks are benign and the system is properly deployed, then the system should

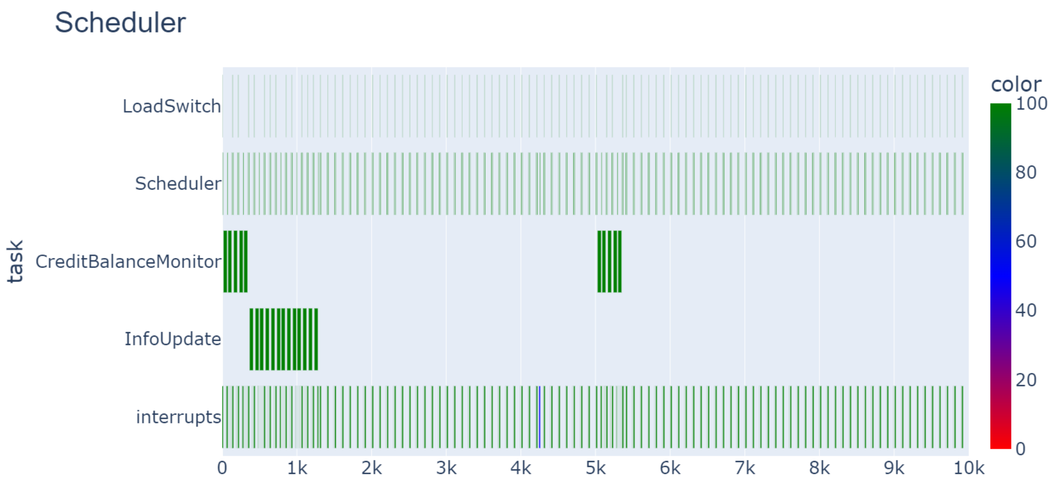


Figure 8.1: A valid schedule for the smart metering problem, with all tasks being benign tasks.

be schedulable. A valid schedule for EDF policy when all the tasks are benign, can be seen in Figure 8.1. *In this figure and the ones that follow, the values on the x-axis have been divided by 1000, to increase readability. This means that 5k on the x-axis, corresponds to 5,000,000 cycles.*

8.5.1 Without availability guarantees

The original design in [20] could ensure integrity and confidentiality. But, as stated in the paper, availability could not be ensured solely based on Sancus. A first type of attack would consist of generating interrupts, by scheduling a task with a very short period. This would make the scheduler run very often. This will lead to most tasks being unable to progress, if they are not able to atomically execute parts of their code. In Figure 8.2, this situation is shown. Only the load switch task is plotted on the schedule to have better readability. The task can only progress by one cycle at a time and will thus be unable to timely service the external interrupt (shown in blue). The red lines in the figure are the interrupts created by the attacker's task (not shown in the figure).
Thus, some section is needed where interrupts are temporarily disabled. This is a good way to guarantee that the task can make progress, as can be seen in Figure 8.3. In that atomic section, the load switch task will not be preempted and can finish timely its process of handling the external interrupt.
However, in that case nothing prevents an attacker from disabling interrupts for extended periods of time. This can be seen in Figure 8.4, where an attacker performs a `clix` of 1,000 ms (or 1,000,000 cycles). The Load Switch task is in this case unable to run for an extended amount of time. If the external interrupt (in blue) comes at a moment the attacker is blocking the system, then the load switch task will not be

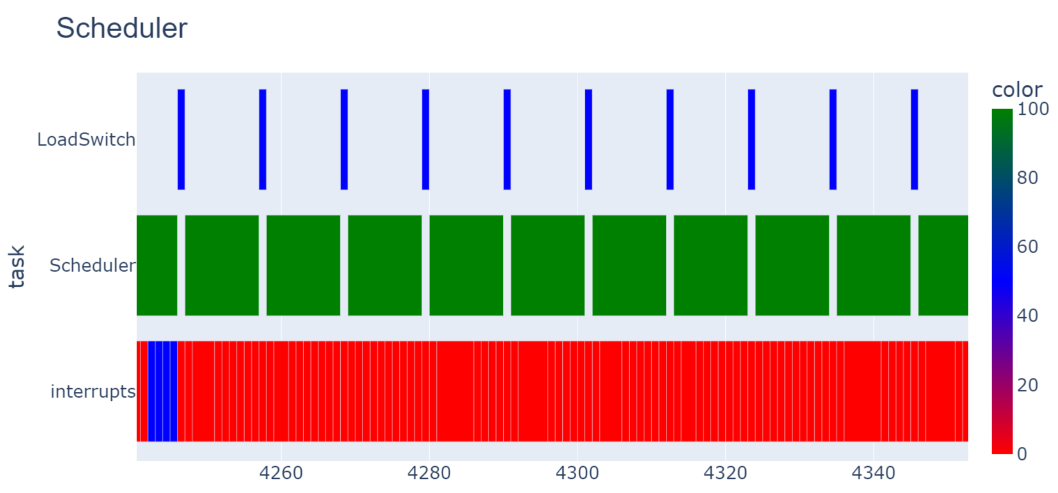


FIGURE 8.2: Schedule for the original smart metering system and with an attacker generating interrupts. The load switch task cannot make progress, because the scheduler always preempts it to handle the attackers interrupts. The task therefore fails to react timely on the external interrupt (in blue).

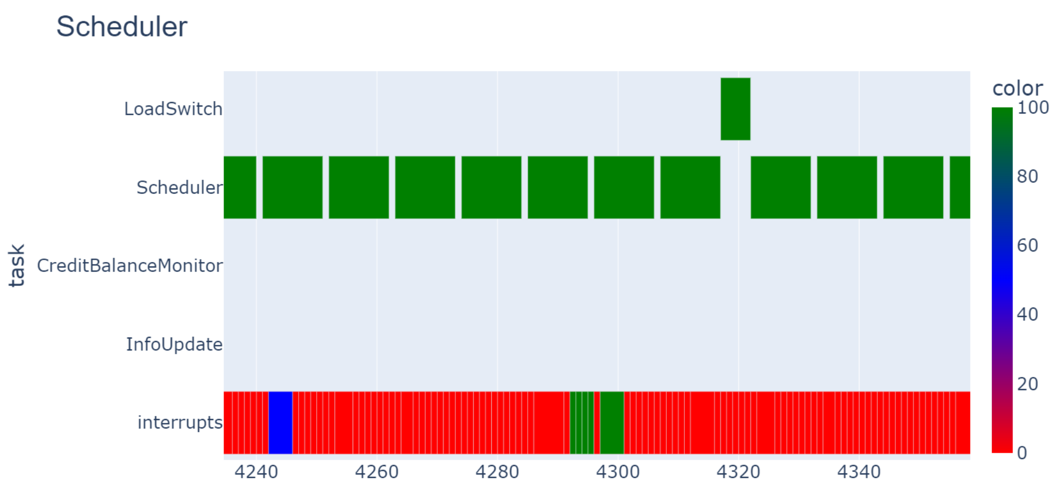


FIGURE 8.3: Schedule for the smart metering system with atomicity sections and with an attacker generating interrupts. In spite of the attacker, the load switch will eventually start running and will be able to run atomically. It will react timely on the external interrupt (in blue).

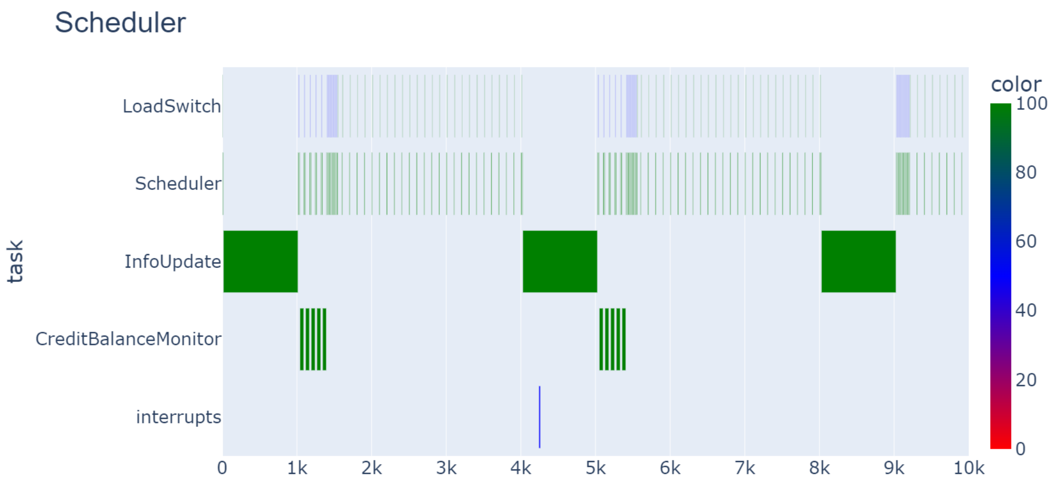


Figure 8.4: Schedule for the smart metering system with atomicity sections and with an attacker performing a long `clix`. The load switch task cannot run while the attacker is performing its `clix`. The external interrupt at time point 4261 will therefore not be timely handled. For clarity, in this figure, the sleep interrupts (in green, to awake the tasks) are left out.

able to react on time on that interrupt. Some upper bound on this atomic sections should therefore be ensured. This is what Aion provides.

8.5.2 Aion

Aion addresses the problem by adding the notion of bounded atomicity [1]. This ensures progress of an application, but also guarantees that all atomic sections will be shorter than a fixed upper bound. Aion can be used with the round-robin policy, as done in [1]. This is an over-restrictive solution, because then the current system with its requirements is already unschedulable (see Figure 8.5) and one of the tasks therefore should be dropped. However, with the EDF policy, the benign case is schedulable (see Figure 8.1).
Unfortunately, Aion without adaptations cannot be used with the EDF policy. Aion puts a constraint on the number of tasks on the system and not on the type of tasks. An attacker could thus update a task to have a smaller period p_a than the load switch task and a `clix` c_a length as close as possible to p_a. This task would be given priority before the load switch task, by having in most cases a nearer deadline. It will consume most of the MCU cycles, leaving not enough for the load switch task. Using the EDF scheduling approach will thus require a way to take into account the tasks characteristics before accepting it on the system.

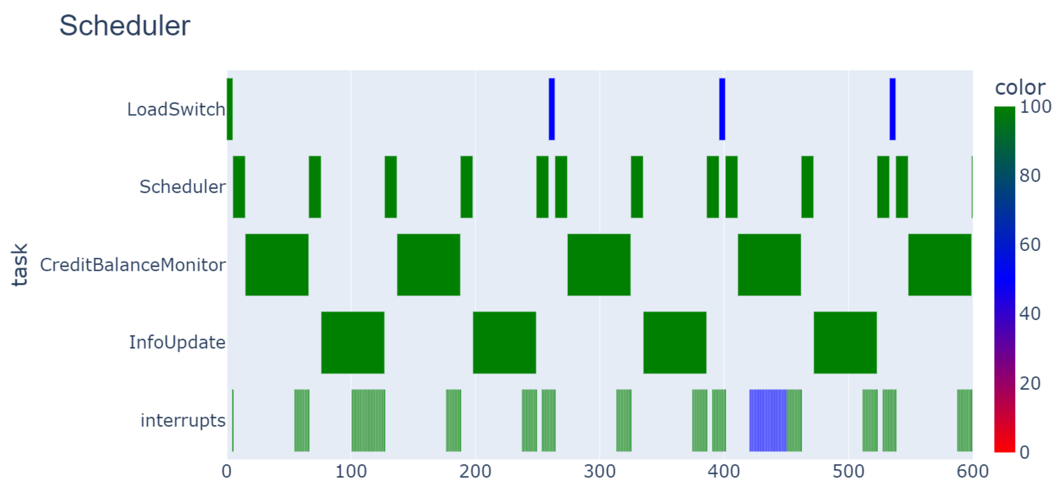


FIGURE 8.5: A small snippet out of the Aion-schedule with the round-robin policy. The system is not schedulable in this case, because all the tasks are treated equally. This gives problems, because the load switch tasks has far more stringent requirements. The load switch task cannot react timely (within 100ms) on the external interrupt.

8.5.3 System with acceptance and enforcement

Stronger constraints and better processor utilization can be achieved by using the Aion architecture in conjuncture with acceptance tests. Acceptance tests however have to be accompanied by enforcement mechanisms. If no enforcement is provided, then the attacker can just lie in her contract and breach the system by going out-of-bounds.
The system in Figure 8.1 is schedulable from the start. If some task violates its contract, this will not change the schedule. The scheduler will just terminate that run of the task, and simulate it till the next period of the task.
An attacker could only break the system by submitting a contract that passes the acceptance test and still makes the system unschedulable. By having proven the sufficiency of the acceptance test (see Section 7.3), this is not possible. The only thing an attacker can do, is submit tasks that will fill the schedule, but the system will remain schedulable.

8.5.4 Remarks

By using this acceptance test method, there is no more need for a second processor as used in [20], from the point of view of availability. The load switch task can be guaranteed to meet its requirements, when running with the other tasks on the same processor. This lowers cost and energy consumption.

The followed approach will treat the different tasks equally. This means that

as long as a task does not violate its contract, the scheduler will guarantee that it will meet its requirements. All deadlines are considered as hard. This can be highly inefficient if some tasks do not require such hard guarantees. The acceptance test could be made much looser in that case, will be discussed in Chapter 9. As long as a task doesn't contain a vulnerability or tries to go over budget (for malicious reasons or because of a bug), the task will be guaranteed to finish in time.
The provided approach has some drawbacks as well. First of all, a WCET has to be provided for each task, and this is non-trivial [27]. Furthermore, the applications should be rephrased to fit within the structure of `clix` sections. For a more elaborate discussion of drawbacks, see Chapter 9.

8.6 Other use cases

Aside from the smart metering application, there are many other possible scenarios where guaranteeing availability in the presence of attackers is of high importance. Modern cars, for example, contain more and more sensors, combined with a lot of computing behind it. They are becoming increasingly autonomous. This however means that it is of vital importance that the processes inside such a car (e.g., the brake) work timely, meeting hard real-time constraints. [26]
Another sector where availability is vital, is the health sector. E-health and IoT devices increase our monitoring and life-saving capabilities. They can be used to follow up patients in their daily life. Nevertheless, some of these devices, such as pacemakers, cannot be easily accessed. This means that the system should support the remote sending of software updates or new applications. [23] This, however, represents a certain risk, because it could be that an attacker is able to submit a new application or corrupt an existing one. The attacker could attempt to monopolize the device. This is a life-threatening situation in the case of pacemakers, where the pacemaker is supposed to react timely to a heart attack, at each moment.

8.7 Conclusion

In this chapter, the smart metering system discussed in [20] was adapted to support the acceptance test approach. The original prototype relies on TEEs and cannot provide availability on top of confidentiality and integrity. Our approach is able to provide real-time guarantees, while providing flexibility to tasks to specify their requirements. It is shown in this chapter that Aion with acceptance tests, can provide a strong notion of availability while reducing the number of processing elements needed to safely implement the scenario.

Chapter 9

Discussion and Conclusion

This chapter reflects on the results obtained in this thesis. It starts with listing some important notes. Then the limitations of the followed approach are highlighted, and some alternative design choices are pointed out. Next, possible directions for future work are formulated. Finally, the different achievements of this thesis are mentioned as an answer on the research questions.

9.1 Notes

No guarantees when bugs

It is imperative to keep in mind that availability of a task assumes that a program is free of vulnerabilities. If an attacker can breach the program itself (e.g., by exploiting a memory management vulnerability), then no guarantees can be given.

Deployment of the acceptance test

The acceptance test can in principle be deployed at different locations. This could be on a separate, dedicated processor for example. However, putting the acceptance test as a special, deadline-free task on the system (see Chapter 8) has some advantages. First, it can be restricted to run in the slack time, and will consequently have no impact on the schedulability of the system. This prevents the acceptance test from becoming a new attack vector. Secondly, this deployment scenario doesn't require any additional hardware than the one previously needed for Aion. Thirdly, the acceptance test module can be deployed inside an enclave, ensuring integrity of the code.
As the applications have to trust the module for availability, including it in their TCB, the integrity of the acceptance test module should be assured. If an attacker can influence the acceptance process by corrupting the module, no availability can be guaranteed. Deploying it inside an enclave makes it possible to attest the acceptance test module for integrity.

Attacks on the acceptance task

An attacker cannot harm availability by violating its contract, because this will be prevented by the enforcement mechanisms. The attacker is also not able to overload the system, because the acceptance test ensure schedulability. An attacker will thus be unable to harm the timeliness of a task that has already been accepted onto the system. The design goal is thus achieved. However, this doesn't prevent attacks on the acceptance test itself. An attacker could issue a DoS attack on the acceptance test, by either submitting a lot of tasks, or submitting contracts that would result in complex calculations. This could make the acceptance process unavailable for benign tasks that are waiting to be accepted. On the other hand, an attacker could also try to fill the processor with tasks, before other SP submitted their task. This would again not harm availability of tasks that were already present, but can still be disturbing for the usability of the system. Both attacks are out of scope.

9.2 Limitations and alternatives

There are inherently some limitations connected to the followed approach. This section will shortly discuss the most important limitations and will also point out some alternatives for design choices that were made.

WCET

It is assumed that the task's parameter, such as WCET, can be estimated. Estimating this upper bound is difficult, but well established in research. [27] Nonetheless, the followed approach to guarantee availability is very restrictive, because the WCET of a task can be quite high, while its average run-time is orders of magnitude lower.

Enforcement

The acceptance test approach relies heavily on assumptions made about task's run-time characteristics. The guarantees that can be given are highly dependent on which properties can be enforced. However, as already mentioned, acceptance and contract enforcement have to be combined. If either one is left out, no availability can be guaranteed. If only enforcement is provided, then it is clear that nothing prevents the attacker from submitting tasks till the system is overloaded. If only acceptance is provided, the attacker can give a contract, but use more resources than promised.

Proof of sufficiency

In the models that were used to prove sufficiency, depending on the task configuration different system parameters must be fixed in the encoding. For example a value had to be attributed to the `OBS_WINDOW` and sufficiency was proven for that value. Then the assumption was made that the acceptance test would also be sufficient for other values of `OBS_WINDOW`. However, this is not generally true, and in theory for each possible `OBS_WINDOW` sufficiency should be proven. This represents a high computational cost, but it may be that some general proof methodology could be found to solve this problem.

SMT solver alternatives

Outside the SMT solver spectrum, different other technologies can be used to model the system and reason about it. There are programs like Ivy[1], that an assist the user in expressing the constraints to obtain a decidable encoding. Ivy is built on top of Z3. Besides, there are logic programming languages such as Prolog. Important is to take into account the specificities of the language, e.g. ‘Negation as a Failure’ in Prolog.

9.3 Future work

Finally, the work presented in this thesis is only a little drop in a vast ocean of questions and research. This section describes some directions for future work.

Prototype implementation

A prototype of the acceptance test approach could be implemented in the future. This would make it possible to test it in a physical scenario. To implement this prototype, further research in concretely implementing this enforcement mechanisms is needed. It should be studied how these mechanisms can be implemented on a concrete system to enforce the assumptions made in Chapter 3.

Resources and dependencies

In this thesis, resource interactions that last longer than one `clix` are not representable in the system. Further research could focus on how to adapt the EDF policy and the acceptance test to handle longer (asynchronous) interactions and to take care of difficulties such as priority inversion.
Another limitation is that tasks are explicitly assumed to be independent, so no (blocking) interaction occurs between tasks. Future work could search for acceptance tests that allow dependencies. The scheduling policy should also be adapted to take these dependencies into account.

Sporadic and aperiodic

In the future, task configurations with sporadic and aperiodic tasks could be considered. These tasks only have to execute irregularly, but could have hard deadlines. A server approach is often used in that case (such as in [22]). It should be investigated how acceptance tests can be designed to take this into account.

[1]http://microsoft.github.io/ivy/

Deadline different from period end

For all task configurations, it was assumed that the deadline of a task is equal to the end of the period of that task. However, there are scenarios that could be better expressed if the deadline is allowed to be smaller than the period.

Soft deadlines

In this thesis, it was assumed that all task have hard deadlines. In the future, it can be interesting to support tasks with soft deadlines, allowing occasional deadline misses. For these tasks the acceptance test could probably be more relaxed and this would make higher utilization of the processor possible.

Other scheduling approaches

An interesting direction for further research is schedulability analysis in open systems when choosing other scheduling approaches. The EDF approach is the one with the highest processor utilization (see 2.2.2). It is however not certain that it will reach the highest possible utilization when used with acceptance tests. In future work, it could be investigated if other scheduling approaches can be found that are more efficient, because the acceptance tests can be more relaxed.

Acceptance tests

In this thesis, an iterative approach was used to refine the acceptance tests until sufficiency could be proven. Z3 also allows to derive consequences out of a system of equations. It could be investigated in future work, whether the acceptance test design could not be improved by using this inference.
In the future, the design approach could also be adapted to search for sufficient acceptance tests with specific properties. It could be desired to find exact acceptance tests (sufficient and necessary), or tests that are computational efficient.

Attestation

Finally, an attestation policy for availability could be developed, similar to that for confidentiality and integrity (as in Sancus [21]). So, a task would be able to attest to its SP that it has been accepted and is running on the system with the required guarantees.

9.4 Achievements

This thesis formulates a possible solution to the availability problem, by answering the research questions in Section 3.4.

Schedulability analysis

As shown in this thesis, schedulability analysis can be used to provide availability guarantees to applications. The applications on the system are assumed to be controlled by an attacker. The approach consists of three aspects: a scheduling policy, enforcement mechanisms and an acceptance test.
First of all, a scheduling policy ensures an ordering of the tasks. In this thesis, the EDF policy was used to order the tasks based on deadlines and to achieve maximal processor utilization. Using this policy in combination with bounded atomicity is not straightforward, as the examples in Section 4.3 show.
Secondly, each task has a contract, which contains a description of its characteristics and requirements. The most elaborate contract discussed in this thesis consisted of a period, a `clix` length and the number of `clix` sections needed. However, next to satisfy the requirements of the task, the system should also enforce the contract.
Thirdly, acceptance tests were introduced as a way to ensure schedulability of the system. These tests perform schedulability analysis using the specification of the task and the current load on the system. It is then possible to decide whether a new task can be accepted or not. The acceptance test solves the problem of guaranteeing availability by guaranteeing that no unschedulable task set can be accepted.

Acceptance test

In this thesis, a methodology is developed to design acceptance tests consisting of mathematical formulas. The design method consisted of an iterative refinement of the acceptance test. A manual approach can be used, but that is inconvenient. By modeling the system, an SMT-solver could be used to guide the refinement process and speed up the design. Additionally, the solver was used to prove the sufficiency of the test. This approach has a variety of advantages. First of all, it is easier to reconstruct and verify the results obtained by this methodology. The constraint model is easily understandable. Secondly, the constraint model can be extended to embed more complexity or different constraints (like a different scheduling policy). This makes it possible to check an acceptance test again if some small changes to the system are made.

More elaborate scenarios

The developed methodology was used to find acceptance tests for more elaborate configurations. Each time, a specific assumption was relaxed, allowing more complicate task sets. A test was designed for the configurations with different `clix` lengths, different periods or the combination of both. Afterwards, it was illustrated how other scenarios (e.g., multiple `clix`) could be reduced to these base configurations. This made it possible to adapt the acceptance tests for these new task configurations, without losing sufficiency. The results were used in the case study about smart metering (see Chapter 8).

9.5 Conclusion

This thesis investigated how availability guarantees could be provided on a system with different, mutually distrusting stakeholders. We proposed an approach using acceptance tests, that is based on the Aion architecture, with adaptations to facilitate enforcement. Scheduling in this environment was discussed to emphasize the difficulties when combining the EDF policy and bounded atomicity, and to pinpoint attack vectors. We developed a design method for acceptance test that uses an SMT-solver to guide the design process and to prove sufficiency of the acceptance test. The approach was deployed in a smart metering system, providing availability guarantees and a reduction in the number of needed processors.

Appendices

Appendix A

Enforcement mechanisms

It is imperative that the assumptions made in the different task configurations, can be enforced. This can be done by using the hardware and the trusted scheduler to monitor the behavior of a task. As the scheduler task is executed every time an interrupt occurs, interrupts will be a powerful tool to make enforcement possible. The following subsections will shortly describe how to enforce a tasks contract.

Enforcing periodicities

To enforce the periodicity of a task, the scheduler will schedule a 'sleep' timer for the the beginning of the following period. When this timer interrupts, the scheduler will run and the task can be put in the ready queue, to be scheduled by EDF. When finished executing, a task can be put in a sleep queue until its next period.

Enforcing fixed clix length

The Aion [1] architecture is able to ensure at hardware-level that a task cannot issue a `clix` longer than the fixed maximum length. This upper bound is set by the system administrator. Aion also flags nested `clix` as violations (see 2.4.2). The scheduler can regain control after the `clix` by scheduling a timer right after the start of the tasks execution.

Enforcing different clix length

The task configuration in Section 7.1 allows each task to specify a different `clix` length. The acceptance test assumes that these `clix` lengths are used by the tasks. Therefore, an adaptation to the Aion is needed in this case. The upper bound on `clix` should be adaptive, and should be allowed to be set by the scheduler. Before each `clix` , the scheduler will set the adaptive `clix` bound to the value that the task mentioned in its contract.

Enforcing multiple clix

To avoid that a task would execute longer than it required in its contract, the scheduler should regain control after each `clix` section. The scheduler can keep count of how much atomicity sections have already been carried out by that task.

Appendix B

Implementation code

B.1 Scheduling simulator

The implementation of the scheduling simulator has been done in Python. The code can be found on Github[1].

B.2 Constraint models for task configurations

The different constraint models have been implemented using the Python API for Z3. This Section lists the code used for modeling the system for four task configurations. The base configuration assumes default period lengths and default `clix` lengths. The other three task configurations extend the base scenario, by either allowing task specific period lengths or task specific `clix` lengths, or both. This code is also available on the same Github page as the scheduling simulator.

B.2.1 Task configuration with default periods and default clix lengths

```
from z3 import *

# In this case, only one period length has to be simulated,
    to check if it is possible
CLIX_BOUND = 7
NR_TASKS = 4
DEFAULT_PERIOD = 21 # T1: D1: 20, D2: 40,  -> max_diff
    between 2 runs is 2*p - 2*b
# RELEASE by default at time-point 0


# Print the schedule in a more readable way
```

[1]https://github.com/emericgoossens/MasterThesis2022

```
def print_schedule(sched_matrix):
    numbers = ""
    print("——————————SCHEDULE
        ————————————————————————————————————")
    for j in range(DEFAULT_PERIOD):
        if j < 10:
            numbers += "␣␣␣" + str(j) + "␣␣␣"
        else:
            numbers += "␣␣" + str(j) + "␣␣␣"
    print(numbers)
    for i in range(NR_TASKS):
        matrix_row = ""
        for j in range(DEFAULT_PERIOD):
            if sched_matrix[i][j]:
                matrix_row += "␣True␣␣"
            else:
                matrix_row += "␣False␣"
            if j % DEFAULT_PERIOD == (DEFAULT_PERIOD - 1):
                matrix_row = matrix_row[:-1]
                matrix_row += "|"
        print(matrix_row)
    print("
        ——————————————————————————————————————————————————————
        ")


# Based on https://ericpony.github.io/z3py-tutorial/guide-
    examples.htm
# Matrix with the tasks on the rows (NR_TASKS) and the
    timepoints on the columns (DEFAULT_PERIOD)
X = [[Bool("x_%s_%s" % (i+1, j+1)) for j in range(
    DEFAULT_PERIOD)]
        for i in range(NR_TASKS)]


# Return if some task is running at the given time
def some_task_is_running(sched, time):
    return Or([sched[i][time] for i in range(NR_TASKS)])


# Return if all tasks have finished before the given time
def all_tasks_finished(sched, time):
    return And([task_finished_at_time_point(sched, i, time)
        for i in range(NR_TASKS)])
```

```
# Return the number of cycles the task has run till the
   given time point
def nr_cycles_ran_before_time(sched, taskNr, time):
    return Sum([If(sched[taskNr][j], 1, 0) for j in range(
        time)])


# Return if the given task has already finished before the
   given time
def task_finished_at_time_point(sched, taskNr, time):
    return nr_cycles_ran_before_time(sched, taskNr, time) >=
         CLIX_BOUND


# Return if the task-run was non-interrupted (2 transitions,
    one start and end)
# or doesn't run at all (0 transitions)
def atomicity_of_task(sched, taskNr):
    nr_transitions = Sum([
        Sum([If(Xor(sched[taskNr][j], sched[taskNr][j + 1]),
            1, 0), # There is a transition
             If(And(j == 0, sched[taskNr][j]), 1, 0),  # the
                 task starts at begin of timeframe
             If(And(j + 1 == DEFAULT_PERIOD - 1, sched[
                taskNr][j + 1]), 1, 0)]) # the  task ends at
                 end of timeframe
         for j in range(DEFAULT_PERIOD - 1)])   # OBS_WINDOW -
             1 because the j + 1 will point till the next
            time point
    return Or(nr_transitions == 2, nr_transitions == 0)


# Constraints
# Each cell is true or false: is already implied by the
   cells being of type bool

# Only one task can run at the same moment (in the same
   column, only one true-value)
no_overlap_c = [Sum([If(X[i][j],1,0) for i in range(NR_TASKS
   )]) <= 1 for j in range(DEFAULT_PERIOD)]

# A task can only start after release: is already done by
   having some finite number of time points starting at 0
```

```
# A task should run for maximal a certain amount of time (
    <= DEFAULT_CLIX_LENGTH)
run_time_c = [nr_cycles_ran_before_time(X, i, DEFAULT_PERIOD
    ) <= CLIX_BOUND for i in range(NR_TASKS)]

# To meet its requirements, a task should at least run the
    clix_length (scheduling goal, >= clix_length)
sched_goal_c = [task_finished_at_time_point(X, i,
   DEFAULT_PERIOD) for i in range(NR_TASKS)]
# The negation of the scheduling goal, has some task missed
    its deadline?
neg_sched_goal = Or([Not(task_finished_at_time_point(X, i,
   DEFAULT_PERIOD)) for i in range(NR_TASKS)])

# Atomicity (no preemption possible)
atomicity_c = [atomicity_of_task(X, i) for i in range(
   NR_TASKS)]

# ----EDF Constraints----
# There will be at each moment one task running, or all
    tasks have finished running
no_idling_when_tasks_ready_c = [Or(some_task_is_running(X, j
    ), all_tasks_finished(X, j)) for j in range(
   DEFAULT_PERIOD)]
# The task running, will be that with the earliest deadline:
     in this simple case all the tasks have the same deadline
    ,
#   so no constraint is needed...

# ----Acceptance test----
acc_test = [NR_TASKS * CLIX_BOUND <= DEFAULT_PERIOD for i in
     range(NR_TASKS)]

# Add all constraints to the solver
s = Solver()
s.add(no_overlap_c)
s.add(run_time_c)
s.add(atomicity_c)
s.add(no_idling_when_tasks_ready_c)

# ----Check if the acceptance test is sufficient----
# There are different phrasings, only one may be used at a
    time (uncomment the needed one)!!!:
# - The acceptance test implies schedulability
# s.add(Implies(And(acc_test), And(sched_goal_c)))
# - It should not be possible to have a situation that
    satisfies the acc_test and misses deadlines
s.add(And(acc_test + [neg_sched_goal]))
# - Or in this simple case, a manually check of the
    acceptance test is also possible, by checking if some
# bad schedule can be found
# s.add(neg_sched_goal)

# It can be interesting to look at valid schedules too. By
    enabling this goal (and commenting out the above ones)
# s.add(sched_goal_c)


value = s.check()
print(value)
if value == sat:
    m = s.model()
    schedule = [[m.evaluate(X[i][j]) for j in range(
        DEFAULT_PERIOD)] for i in range(NR_TASKS)]
    print_schedule(schedule)

print(s.statistics())
```

B.2.2 Task configuration allowing task-specific clix lengths

```
from z3 import *

# This configuration builds on top of the simplest
    configuration (see SamePeriodSameBudgetSameRelease.py).
# The release is assumed to be at 0 and the period will be
    fixed too (and the same for all tasks)
# However, now the clix length can vary from task to task
# In this case, only one period length has to be simulated
# CLIX_BOUND = 7      => No default clix length
NR_TASKS = 3
DEFAULT_PERIOD = 22 # T1: D1: 20, D2: 40,  -> max_diff
    between 2 runs is 2*p - 2*b
#RELEASE by default at timepoint 0


# For visual purposes
def print_schedule(sched_matrix):
    numbers = ""
    print("----------------SCHEDULE
        ----------------------------------------")
    for j in range(DEFAULT_PERIOD):
        if j < 10:
            numbers += "␣␣␣" + str(j) + "␣␣␣"
        else:
            numbers += "␣␣" + str(j) + "␣␣␣"
    print(numbers)
    for i in range(NR_TASKS):
        matrix_row = ""
        for j in range(DEFAULT_PERIOD):
            if sched_matrix[i][j]:
                matrix_row += "␣True␣␣"
            else:
                matrix_row += "␣False␣"
            if j % DEFAULT_PERIOD == (DEFAULT_PERIOD - 1):
                matrix_row = matrix_row[:-1]
                matrix_row += "|"
        print(matrix_row)
    print("
       --------------------------------------------------------------------------------------------
       ")

# ------------------Additional rules ----------------------------------


# List with the clix-length for the tasks
clix_length = [Int("clix_%s" % (i+1)) for i in range(
   NR_TASKS)]


# The clix length cannot be smaller than 0 and not be
    greater than the DEFAULT_PERIOD
# (because in the latter case it is trivial that it is
    unschedulable)
clix_c = [And(clix_length[i] > 0, clix_length[i] <=
   DEFAULT_PERIOD) for i in range(NR_TASKS)]


# Return the clix length of a given task.
def get_clix_length(taskNr):
    return clix_length[taskNr]


# ------------------Adapted code
   -------------------------------------------------------------------------------------------------
```

```
# The differences are located at the points where first the
    CLIX_BOUND parameter was used...
# Now the getter for clix length is used instead


# Based on https://ericpony.github.io/z3py-tutorial/guide-
    examples.htm
# Matrix with the tasks on the rows (NR_TASKS) and the
    timepoints on the columns (DEFAULT_PERIOD)
X = [[Bool("x_%s_%s" % (i+1, j+1)) for j in range(
    DEFAULT_PERIOD)]
        for i in range(NR_TASKS)]


# Return if some task is running at the given time
def some_task_is_running(sched, time):
    return Or([sched[i][time] for i in range(NR_TASKS)])


# Return if all tasks have finished before the given time
def all_tasks_finished(sched, time):
    return And([task_finished_at_time_point(sched, i, time)
        for i in range(NR_TASKS)])


# Return the number of cycles the task has run till the
    given time point
def nr_cycles_ran_before_time(sched, taskNr, time):
    return Sum([If(sched[taskNr][j], 1, 0) for j in range(
        time)])


# Return if the given task has already finished before the
    given time
def task_finished_at_time_point(sched, taskNr, time):
    return nr_cycles_ran_before_time(sched, taskNr, time) >=
         get_clix_length(taskNr)


# Return if the task-run was non-interrupted (2 transitions,
     one start and end)
# or doesn't run at all (0 transitions)
def atomicity_of_task(sched, taskNr):
    nr_transitions = Sum([
        Sum([If(Xor(sched[taskNr][j], sched[taskNr][j + 1]),
                1, 0),  # There is a transition
                 If(And(j == 0, sched[taskNr][j]), 1, 0),  # the
                      task starts at begin of timeframe
                 If(And(j + 1 == DEFAULT_PERIOD - 1, sched[
                     taskNr][j + 1]), 1, 0)]) # the  task ends at
                      end of timeframe
          for j in range(DEFAULT_PERIOD - 1)])  # OBS_WINDOW -
                1 because the j + 1 will point till the next
              time point
    return Or(nr_transitions == 2, nr_transitions == 0)


# Constraints
# Each cell is true or false: is already implied by the
    cells being of type bool

# Only one task can run at the same moment (in the same
    column, only one true-value)
no_overlap_c = [Sum([If(X[i][j],1,0) for i in range(NR_TASKS
    )]) <= 1 for j in range(DEFAULT_PERIOD)]

# A task can only start after release: is already done by
    having some finite number of time points starting at 0

# A task should run for maximal a certain amount of time (
    <= clix_length)
run_time_c = [nr_cycles_ran_before_time(X, i, DEFAULT_PERIOD
    ) <= get_clix_length(i) for i in range(NR_TASKS)]

# To meet its requirements, a task should at least run the
    clix_length (scheduling goal, >= clix_length)
sched_goal_c = [task_finished_at_time_point(X, i,
    DEFAULT_PERIOD) for i in range(NR_TASKS)]
# The negation of the scheduling goal: has some task missed
    its deadline?
neg_sched_goal = Or([Not(task_finished_at_time_point(X, i,
    DEFAULT_PERIOD)) for i in range(NR_TASKS)])

# Atomicity (no preemption possible)
atomicity_c = [atomicity_of_task(X, i) for i in range(
    NR_TASKS)]

# ----EDF Constraints----
# There will be at each moment one task running, or all
    tasks have finished running
```

```
no_idling_when_tasks_ready_c = [Or(some_task_is_running(X, j
    ), all_tasks_finished(X, j)) for j in range(
    DEFAULT_PERIOD)]
# The task running, will be that with the earliest deadline:
     in this simple case all the tasks have the same
# period/deadline, so no constraint is needed...

# ---Acceptance test---
# This acceptance test is intuitive: if all the execution
    times fit into the period length,
# then the system is schedulable.
acc_test = Sum(clix_length) <= DEFAULT_PERIOD

# Add all constraints to the solver
s = Solver()
s.add(no_overlap_c)
s.add(run_time_c)
s.add(atomicity_c)
s.add(no_idling_when_tasks_ready_c)

s.add(clix_c)

# ---Check if the acceptance test is sufficient---
# There are different phrasings, only one may be used at a
    time (uncomment the needed one):
# - The acceptance test implies schedulability
# s.add(Implies(acc_test, And(sched_goal_c)))
# - It should not be possible to have a situation that
    satisfies the acc_test and misses deadlines
s.add(And(acc_test, neg_sched_goal))
# - Or in this simple case, you can also manually check the
    acceptance test and then check if some
# bad schedule can be found
# s.add(neg_sched_goal)

# It can be interesting to look at valid schedules too. By
    enabling this goal (and commenting out the above ones)
# s.add(sched_goal_c)

# With this line, it can be shown that the acceptance test
    condition is also a necessary condition (not only
    sufficient)
# Therefore it is an exact acceptance test.
# s.add(And(Not(acc_test), And(sched_goal_c)))
```

```
value = s.check()
print(value)
if (value == sat):
    m = s.model()
    schedule = [[m.evaluate(X[i][j]) for j in range(
        DEFAULT_PERIOD)] for i in range(NR_TASKS)]
    clix = [m.evaluate(clix_length[i]) for i in range(
        NR_TASKS)]
    print(clix)
    print_schedule(schedule)

print(s.statistics())
```

B.2.3 Task configuration allowing task-specific periodicities

```
from z3 import *

# This configuration builds on top of the simplest
    configuration (see SamePeriodSameBudgetSameRelease.py).
# The release is assumed to be at 0 and the budget will be
    fixed too (and the same for all tasks)
# However, now the period can vary from task to task. It
    will be bound to be a common divisor of the
    OBSERVATION_WINDOW
# In this case, only one OBSERVATION_WINDOW length has to be
     simulated
CLIX_BOUND = 3
NR_TASKS = 3
OBSERVATION_WINDOW = 20
MAX_RUNS = OBSERVATION_WINDOW # this constant is needed to
    make the model solvable, to make the for-loops usable...
                    # It should be at least OBS_WINDOW/
                        CLIX_BOUND
#RELEASE by default at timepoint 0

# ----------------Additional rules ----------------------

# List with the clix-length for the tasks
period_length = [Int("period_%s" % (i+1)) for i in range(
    NR_TASKS)]


# The period has to be bigger than 0 and cannot be greater
    than the OBSERVATION_WINDOW.
```

```
# It furthermore has to be a divisor of the
    OBSERVATION_WINDOW.
period_c = [And(period_length[i] > 0, period_length[i] <=
    OBSERVATION_WINDOW,
                 OBSERVATION_WINDOW % period_length[i] == 0)
              for i in range(NR_TASKS)]


# Return the period of the given task
def get_period(taskNr):
     return period_length[taskNr]


# Return whether the timepoint is in between begin and end,
    including the begin and end-point.
def in_between(time, begin, end):
     return And(begin <= time, time <= end)


# PeriodNr starts at 0 and ends at (OBSERVATION_WINDOW/
    period_length) - 1
def get_begin_of_period(taskNr, periodNr):
     return periodNr * get_period(taskNr)


def get_begin_of_period_given_timepoint(taskNr, time):
     return (Sum([If(get_begin_of_period(taskNr, periodNr) <=
          time, 1, 0) for periodNr in range(MAX_RUNS)])
              - 1) * get_period(taskNr)


# Return the number of cycles the task has run in between
    the two time points
def nr_cycles_ran_in_between_timepoints(sched, taskNr, begin
    , end):
     return \
          Sum([If(
              And(in_between(j, begin, end - 1), sched[taskNr
                  ][j]) #end - 1 because in-between includes
                  the endpoints
              , 1, 0)
              for j in range(OBSERVATION_WINDOW)])


# Return whether the task has run less than one clix within
```

```
    it 's period
def task_will_run_not_longer_than_one_clix_per_period(sched ,
     taskNr , periodNr):
     return nr_cycles_ran_in_between_timepoints(sched , taskNr
         ,
                                                           get_begin_of_period
                                                               (taskNr ,
                                                               periodNr),
                                                           get_begin_of_period
                                                               (taskNr ,
                                                               periodNr +
                                                                1)) \
             <= CLIX_BOUND


# Return whether the task has run at least a complete clix
    this period
def task_has_fully_run_this_period(sched , taskNr , periodNr):
     return nr_cycles_ran_in_between_timepoints(sched , taskNr
         ,
                                                           get_begin_of_period
                                                               (taskNr ,
                                                               periodNr),
                                                           get_begin_of_period
                                                               (taskNr ,
                                                               periodNr +
                                                                1)) \
             >= CLIX_BOUND


def get_nr_of_runs(taskNr):
     return OBSERVATION_WINDOW/get_period(taskNr)


# NrOfPeriods is the number of periods this task has to run
    within the observation window (see get_nr_of_runs(...))
# periodNr will range from 0 to NrOfPeriods - 1
def task_has_run_fully_all_periods(sched , taskNr ,
    NrOfPeriods):
     return And([Or(task_has_fully_run_this_period(sched ,
         taskNr , periodNr), periodNr >= NrOfPeriods)
                 for periodNr in range(MAX_RUNS)])

# ----------------Adapted code
    ------------------------------------------------------------------
```

```
# The differences are located at the points where first the
    DEFAULT_PERIOD parameter was used...
# In most cases, this was where an iteration over all the
    time points was needed (there OBSERVATION_WINDOW is used
    instead)
# In some cases, for the completion of a task however, the
    task period (get_period(taskNr)) has to be taken into
    account
# Because a task with a short period can run multiple times
    in the OBSERVATION_WINDOW time.

# Print the schedule in a more readable way
def print_schedule(sched_matrix, periods):
    numbers = ""
    print("----------SCHEDULE
        -------------------------------------")
    for j in range(OBSERVATION_WINDOW):
        if j < 10:
            numbers += "   " + str(j) + "   "
        else:
            numbers += "  " + str(j) + "   "
    print(numbers)
    for i in range(NR_TASKS):
        matrix_row = ""
        for j in range(OBSERVATION_WINDOW):
            if sched_matrix[i][j]:
                matrix_row += " True  "
            else:
                matrix_row += " False "
            if j % periods[i].as_long() == (periods[i].
                as_long() - 1):
                matrix_row = matrix_row[:-1]
                matrix_row += "|"
        print(matrix_row)
    print("
        ---------------------------------------------------------------
        ")


# Based on https://ericpony.github.io/z3py-tutorial/guide-
    examples.htm
# Matrix with the tasks on the rows (NR_TASKS) and the
    timepoints on the columns (OBSERVATION_WINDOW)
X = [[Bool("x_%s_%s" % (i+1, j+1)) for j in range(
    OBSERVATION_WINDOW)]
        for i in range(NR_TASKS)]


# Return if some task is running at the given time
def some_task_is_running(sched, time):
    return Or([sched[i][time] for i in range(NR_TASKS)])


# All tasks have finished their periodic run
# has run longer than the default clix length, between the
    begin of period and time
def finished_periodic_run(sched, time, taskNr):
    return nr_cycles_ran_in_between_timepoints(sched, taskNr
        ,
    get_begin_of_period_given_timepoint(taskNr, time),
    time) \
            >= CLIX_BOUND


# Return if all tasks have done all their work (so all the
    necessary runs) before the given time
# The task needs to have finished its run within the current
     period.
def all_tasks_finished_their_run(sched, time):
    return And([finished_periodic_run(sched, time, i) for i
        in range(NR_TASKS)])


# Return if the task-run was non-interrupted (2 transitions,
     one start and end)
# or doesn't run at all (0 transitions)
def atomicity_of_one_run(sched, taskNr, periodNr):
    current_period_begin = get_begin_of_period(taskNr,
        periodNr)
    period = get_period(taskNr)
    nr_transitions = Sum([If(
        in_between(j, current_period_begin,
            current_period_begin + period - 2),
        # Because j + 1 == current_period_begin+period - 1
            should be last point
        Sum([If(Xor(sched[taskNr][j], sched[taskNr][j + 1]),
             1, 0), # There is a transition
              If(And(j == current_period_begin, sched[taskNr
                  ][j]), 1, 0), # the  task starts at begin
```

```
                of timeframe
              If(And(j + 1 == current_period_begin + period -
                  1, sched[taskNr][j + 1]), 1, 0)]),
              # the task ends at end of timeframe
          0)
          for j in range(OBSERVATION_WINDOW - 1)])  #
             OBS_WINDOW - 1 because the j + 1 will point till
             the next time point
      return Or(nr_transitions == 2, nr_transitions == 0)


# All runs have to be atomic
def atomicity_of_task(sched, taskNr):
      return And([Or(atomicity_of_one_run(sched, taskNr,
         periodNr), periodNr >= get_nr_of_runs(taskNr))
                for periodNr in range(MAX_RUNS)])

# FOR EARLIEST DEADLINE


# Return whether the task that starts at this time point is
    the task with the nearest deadline.
def starting_task_has_nearest_deadline(sched, time):
      # If the task has started now, then its deadline should
         be the nearest one.
      # Task has started clix:
      #         * is running now + is only running first cycle
      # If this is the case, the task should be the one with
         the smallest period
      # All other ready tasks should have higher period
      return And([
          Implies(
              And(sched[i][time],
                  nr_cycles_ran_in_between_timepoints(sched, i
                      ,
                  get_begin_of_period_given_timepoint(i, time)
                      , time + 1) == 1),
              is_ready_with_nearest_deadline(sched, time, i))
          for i in range(NR_TASKS)])


# Return the following deadline for the given task compared
    to the time point.
# This is equal to the begin of the next period.
def get_deadline_given_time_point(time, taskNr):
    return get_begin_of_period_given_timepoint(taskNr, time)
        + period_length[taskNr]


# Return whether the given task is the ready task with the
    nearest deadline.
def is_ready_with_nearest_deadline(sched, time, taskNr):
    # Either the other tasks have not the nearest deadline,
        or they ran already within their period.
    return And([Or(get_deadline_given_time_point(time,
       taskNr) <= get_deadline_given_time_point(time, j),
                finished_periodic_run(sched, time, j))
              for j in range(NR_TASKS)])



# Constraints
# Each cell is true or false: is already implied by the
    cells being of type bool

# Only one task can run at the same moment (in the same
    column, only one true-value)
no_overlap_c = [Sum([If(X[i][j], 1, 0) for i in range(
   NR_TASKS)]) <= 1 for j in range(OBSERVATION_WINDOW)]

# A task can only start after release: is already done by
    having some finite number of time points starting at 0

# A task should run for maximal a certain amount of time (
    <= CLIX_BOUND)
run_time_c = [And([Or(
    task_will_run_not_longer_than_one_clix_per_period(X, i,
    periodNr), periodNr >= get_nr_of_runs(i))
                for periodNr in range(MAX_RUNS)])
             for i in range(NR_TASKS)]
#
# To meet its requirements, a task should at least run the
    clix_length (scheduling goal, >= clix_length)
sched_goal_c = [task_has_run_fully_all_periods(X, i,
    get_nr_of_runs(i)) for i in range(NR_TASKS)]
# The negation of the scheduling goal, has some task missed
    its deadline?
neg_sched_goal = Or([Not(task_has_run_fully_all_periods(X, i
    , get_nr_of_runs(i))) for i in range(NR_TASKS)])

# Atomicity (no preemption possible)
```

```
atomicity_c = [atomicity_of_task(X, i) for i in range(
    NR_TASKS)]

# ——EDF Constraints——
# There will be at each moment one task running, or all
    tasks have finished running
# So for each time point j: either some task is running, or
    all tasks have currently finished their periodic run.
no_idling_when_tasks_ready_c = [Or(some_task_is_running(X, j
    ), all_tasks_finished_their_run(X, j))
                                    for j in range(
                                        OBSERVATION_WINDOW)]

# The task running, will be that with the earliest deadline:
earliest_deadline_first_c = [
    starting_task_has_nearest_deadline(X, j) for j in range(
    OBSERVATION_WINDOW - 1)]

# ——Acceptance test——
# The following acceptance test is not sufficient. It only
    checks if everything would fit inside the observation
# window, but does not take care of different periodicities.
# acc_test = Sum([get_nr_of_runs(i) * CLIX_BOUND for i in
    range(NR_TASKS)]) <= OBSERVATION_WINDOW
# Sufficient acceptance test (but very restrictive). It
    ensures that each period is at least big enough to
    contain
# one execution of each task. This is sufficient.
acc_test = And([NR_TASKS * CLIX_BOUND <= period_length[i]
    for i in range(NR_TASKS)])

# Add all constraints to the solver
s = Solver()
s.add(no_overlap_c)
s.add(run_time_c)
s.add(atomicity_c)
s.add(no_idling_when_tasks_ready_c)

s.add(period_c)
s.add(earliest_deadline_first_c)

# ——Check if the acceptance test is sufficient——
# There are different phrasings:
# - The acceptance test implies schedulability
# s.add(Implies(acc_test, And(sched_goal_c)))
# - It should not be possible to have a situation that
    satisfies the acc_test and misses deadlines
s.add(And(acc_test, neg_sched_goal))
# - Or in this simple case, you can also manually check the
    acceptance test and then check if some
# bad schedule can be found
# s.add(neg_sched_goal)

# Check if there are at least some schedulable task sets
    that satisfy the acceptance test
# To be sure that the acceptance test is not far too
    restrictive
# s.add(And(sched_goal_c), acc_test)

# Checking if the acceptance test is necessary: if this
    gives unsat then a system is only schedulable if accepted
# If a test is both necessary and sufficient, then it is
    exact.
# s.add(Not(acc_test), And(sched_goal_c))

# print(s.assertions())
value = s.check()
print(value)
if value == sat:
    m = s.model()
    schedule = [[m.evaluate(X[i][j]) for j in range(
        OBSERVATION_WINDOW)] for i in range(NR_TASKS)]
    periods = [m.evaluate(period_length[i]) for i in range(
        NR_TASKS)]
    print("Periods:␣" + str(periods))
    print_schedule(schedule, periods)

print(s.statistics())
```

B.2.4 Task configuration allowing both periodicities and clix lengths to be specified by task

```
from z3 import *

# This configuration builds on top of the config with
    different periods and default clix
# (SameBudgetDifferentPeriodv2.py).
# The configuration with default periods, but different
    budgets (clix-length) is merged into it.
```

```
# The release is assumed to be at 0 and the period can vary
    from task to task.
# However, now the budget(clix-length) can vary from task to
     task too.
# It will be upper bounded by the period of the task, and
    lower bounded by 1.
# In this case, only one OBSERVATION_WINDOW length has to be
     simulated

# DEFAULT_CLIX_LENGTH = 3    => No default clix length
NR_TASKS = 3
OBSERVATION_WINDOW = 28
MAX_RUNS = OBSERVATION_WINDOW # this constant is needed to
    make the model solvable, to make list comprehension
    possible
                    # It should be at least OBS_WINDOW to be
                        sure it covers all possible clix length
                        scenarios
#RELEASE by default at timepoint 0

# ----------------Additional rules From
    SamePeriodDifferentBudget
    ------------------------------------------------------------

# List with the clix-length for the tasks
clix_length = [Int("clix_%s" % (i+1)) for i in range(
    NR_TASKS)]


# The clix length cannot be smaller than 0 and not be
    greater than the OBSERVATION_WINDOW
# (in the latter case it is trivial that it is unschedulable
    ,
# because the periods are also restricted to be less than
    the OBSERVATION_WINDOW)
clix_c = [And(clix_length[i] > 0, clix_length[i] <=
    OBSERVATION_WINDOW) for i in range(NR_TASKS)]


# Return the clix length of the given task
def get_clix_length(taskNr):
     return clix_length[taskNr]


# ----------------Adapted code From
```

SameBudgetDifferentPeriodv2

```
# The differences are located at the points where first the
   DEFAULT_CLIX parameter was used...
# In these cases the get_clix_length method is used instead.

# Print the schedule in a more readable way
def print_schedule(sched_matrix, periods):
    numbers = ""
    print("--------------------SCHEDULE
       ----------------------------------------------------")
    for j in range(OBSERVATION_WINDOW):
        if j < 10:
            numbers += "␣␣␣" + str(j) + "␣␣␣"
        else:
            numbers += "␣␣" + str(j) + "␣␣␣"
    print(numbers)
    for i in range(NR_TASKS):
        matrix_row = ""
        for j in range(OBSERVATION_WINDOW):
            if sched_matrix[i][j]:
                matrix_row += "␣True␣␣"
            else:
                matrix_row += "␣False␣"
            if j % periods[i].as_long() == (periods[i].
               as_long() - 1):
                matrix_row = matrix_row[:-1]
                matrix_row += "|"
        print(matrix_row)
    print("
       ------------------------------------------------------------------------------------
       ")


# List with the clix-length for the tasks
period_length = [Int("period_%s" % (i+1)) for i in range(
   NR_TASKS)]


# The period has to be bigger than 0 and cannot be greater
   than the OBSERVATION_WINDOW.
# It furthermore has to be a divisor of the
   OBSERVATION_WINDOW.
period_c = [And(period_length[i] > 0, period_length[i] <=
    OBSERVATION_WINDOW,
                  OBSERVATION_WINDOW % period_length[i] == 0)
            for i in range(NR_TASKS)]


# Return the period of the given task
def get_period(taskNr):
    return period_length[taskNr]


# Return whether the timepoint is in between begin and end,
    including the begin and end-point.
def in_between(time, begin, end):
    return And(begin <= time, time <= end)


# PeriodNr starts at 0 and ends at (OBSERVATION_WINDOW/
    period_length) - 1
def get_begin_of_period(taskNr, periodNr):
    return periodNr * get_period(taskNr)


def get_begin_of_period_given_timepoint(taskNr, time):
    return (Sum([If(get_begin_of_period(taskNr, periodNr) <=
         time, 1, 0) for periodNr in range(MAX_RUNS)])
            - 1) * get_period(taskNr)


# Return the number of cycles the task has run in between
    the two time points, this doesn't include the end point.
def nr_cycles_ran_in_between_timepoints(sched, taskNr, begin
    , end):
    return \
         Sum([If(
            And(in_between(j, begin, end - 1), sched[taskNr
                ][j]) #end - 1 because in-between includes
                the endpoints
            , 1, 0)
            for j in range(OBSERVATION_WINDOW)])


# Return whether the task has run less than one clix within
    it's period
def task_will_run_not_longer_than_one_clix_per_period(sched,
     taskNr, periodNr):
```

```
    return nr_cycles_ran_in_between_timepoints(sched, taskNr
        ,
                                                    get_begin_of_period
                                                        (taskNr,
                                                        periodNr),
                                                    get_begin_of_period
                                                        (taskNr,
                                                        periodNr +
                                                         1)) \
            <= get_clix_length(taskNr)


# Return whether the task has run at least a complete clix
    this period
def task_has_fully_run_this_period(sched, taskNr, periodNr):
    return nr_cycles_ran_in_between_timepoints(sched, taskNr
        ,
                                                    get_begin_of_period
                                                        (taskNr,
                                                        periodNr),
                                                    get_begin_of_period
                                                        (taskNr,
                                                        periodNr +
                                                         1)) \
                            >= get_clix_length(taskNr)


def get_nr_of_runs(taskNr):
    return OBSERVATION_WINDOW/get_period(taskNr)


# NrOfPeriods is the number of periods this task has to run
    within the observation window (see get_nr_of_runs(...))
# periodNr will range from 0 to NrOfPeriods - 1
def task_has_run_fully_all_periods(sched, taskNr,
    NrOfPeriods):
    return And([Or(task_has_fully_run_this_period(sched,
        taskNr, periodNr), periodNr >= NrOfPeriods)
                for periodNr in range(MAX_RUNS)])


# Based on https://ericpony.github.io/z3py-tutorial/guide-
    examples.htm
# Matrix with the tasks on the rows (NR_TASKS) and the
    timepoints on the columns (OBSERVATION_WINDOW)
```

```
X = [[Bool("x_%s_%s" % (i+1, j+1)) for j in range(
    OBSERVATION_WINDOW)]
      for i in range(NR_TASKS)]


# Return if some task is running at the given time
def some_task_is_running(sched, time):
    return Or([sched[i][time] for i in range(NR_TASKS)])


# Given task has finished its periodic run:
# has run longer than the default clix length, between the
    begin of period and time
def finished_periodic_run(sched, time, taskNr):
    return nr_cycles_ran_in_between_timepoints(sched, taskNr
        ,
    get_begin_of_period_given_timepoint(taskNr, time),
                                                time) \
            >= get_clix_length(taskNr)


# Return if all tasks have done all their work (so all the
    necessary runs) before the given time
# The task needs to have finished its run within its current
     period.
def all_tasks_finished_their_run(sched, time):
    return And([finished_periodic_run(sched, time, i) for i
        in range(NR_TASKS)])


# Return if the task-run was non-interrupted (2 transitions,
     one start and end)
# or doesn't run at all (0 transitions)
def atomicity_of_one_run(sched, taskNr, periodNr):
    current_period_begin = get_begin_of_period(taskNr,
        periodNr)
    period = get_period(taskNr)
    nr_transitions = Sum([If(
        in_between(j, current_period_begin,
            current_period_begin + period - 2), # Because j +
             1 == current_period_begin+period - 1 should be
            last point
        Sum([If(Xor(sched[taskNr][j], sched[taskNr][j + 1]),
             1, 0), # There is a transition
              If(And(j == current_period_begin, sched[taskNr
                ][j]), 1, 0),  # the  task starts at begin
                of timeframe
             If(And(j + 1 == current_period_begin + period -
                1, sched[taskNr][j + 1]), 1, 0)]), # the
                task ends at end of timeframe
        0)
        for j in range(OBSERVATION_WINDOW - 1)]) #
           OBS_WINDOW - 1 because the j + 1 will point till
           the next time point
    return Or(nr_transitions == 2, nr_transitions == 0)


# All runs have to be atomic
def atomicity_of_task(sched, taskNr):
    return And([Or(atomicity_of_one_run(sched, taskNr,
       periodNr), periodNr >= get_nr_of_runs(taskNr))
                for periodNr in range(MAX_RUNS)])

# FOR EARLIEST DEADLINE


# Return whether the task that starts at this time point is
   the task with the nearest deadline.
def starting_task_has_nearest_deadline(sched, time):
    # If the task has started now, then its deadline should
       be the nearest one.
    # Task has started clix:
    #       * is running now + is only running first cycle
    # If this is the case, the task should be the one with
       the smallest period
    # All other ready tasks should have higher period
    return And([
        Implies(
            And(sched[i][time],
                nr_cycles_ran_in_between_timepoints(sched, i
                   ,
                get_begin_of_period_given_timepoint(i, time)
                   , time + 1) == 1),
            is_ready_with_nearest_deadline(sched, time, i))
        for i in range(NR_TASKS)])


# Return the following deadline for the given task compared
   to the time point.
# This is equal to the begin of the next period.
```

```
def get_deadline_given_time_point(time, taskNr):
    return get_begin_of_period_given_timepoint(taskNr, time)
        + period_length[taskNr]


# Return whether the given task is the ready task with the
    nearest deadline.
def is_ready_with_nearest_deadline(sched, time, taskNr):
    # Either the other tasks have not the nearest deadline,
        or they ran already within their period.
    return And([Or(get_deadline_given_time_point(time,
        taskNr) <= get_deadline_given_time_point(time, j),
                    finished_periodic_run(sched, time, j))
                for j in range(NR_TASKS)])


# Constraints
# Each cell is true or false: is already implied by the
    cells being of type bool

# Only one task can run at the same moment (in the same
    column, only one true-value)
no_overlap_c = [Sum([If(X[i][j], 1, 0) for i in range(
    NR_TASKS)]) <= 1 for j in range(OBSERVATION_WINDOW)]

# A task can only start after release: is already done by
    having some finite number of time points starting at 0

# A task should run for maximal a certain amount of time (
    <= get_clix_length(taskNr))
run_time_c = [And([Or(
    task_will_run_not_longer_than_one_clix_per_period(X, i,
    periodNr), periodNr >= get_nr_of_runs(i))
                    for periodNr in range(MAX_RUNS)])
                for i in range(NR_TASKS)]
#
# To meet its requirements, a task should at least run the
    clix_length (scheduling goal, >= clix_length)
sched_goal_c = [task_has_run_fully_all_periods(X, i,
    get_nr_of_runs(i)) for i in range(NR_TASKS)]
# The negation of the scheduling goal, has some task missed
    its deadline?
neg_sched_goal = Or([Not(task_has_run_fully_all_periods(X, i
    , get_nr_of_runs(i))) for i in range(NR_TASKS)])
```

```
# Atomicity (no preemption possible)
atomicity_c = [atomicity_of_task(X, i) for i in range(
    NR_TASKS)]

# ——EDF Constraints——
# There will be at each moment one task running, or all
    tasks have finished running
# So for each time point j: either some task is running, or
    all tasks have currently finished their periodic run.
no_idling_when_tasks_ready_c = [Or(some_task_is_running(X, j
    ), all_tasks_finished_their_run(X, j))
                                            for j in range(
                                               OBSERVATION_WINDOW)]

# The task running, will be that with the earliest deadline:
earliest_deadline_first_c = [
    starting_task_has_nearest_deadline(X, j) for j in range(
    OBSERVATION_WINDOW - 1)]

# ——Acceptance test——
# The acceptance test should take care of the different
    periods and different clix-lengths.
# A simplistic adaptation of the acceptance test for the
    sameBudgetDifferentPeriod config, is not sufficient:
# acc_test = And([NR_TASKS*get_clix_length(i) <=
    period_length[i] for i in range(NR_TASKS)])

# Another acceptance test can be based on the fact that the
    total utilization should be less than the OBS Window
# However, this is not sufficient, because some tasks could
# have clix that are bigger than the task with the smallest
    period
# acc_test = Sum([get_clix_length(i)*get_nr_of_runs(i) for i
     in range(NR_TASKS)]) <= OBSERVATION_WINDOW

# Adding additional constraint to avoid tasks from having
    too big clix compared to the min period, this is however
# still not sufficient, because if the task with largest
    period starts right before a new small period, then it
    will
# finish it clix and the other tasks will be in time
    pressure...
# MAX_CLIX = 4
# MIN_PERIOD = MAX_CLIX
# acc_test = And([Sum([get_clix_length(i)*get_nr_of_runs(i)
    for i in range(NR_TASKS)]) <= OBSERVATION_WINDOW,
#                   And([period_length[i] >= MIN_PERIOD for i
    in range(NR_TASKS)]),
#                   And([clix_length[i] <= MAX_CLIX for i in
    range(NR_TASKS)])])

#

# By adding additional constraints on the period lengths, a
    sufficient acceptance test is found.
#
MAX_CLIX = 4
MIN_PERIOD = 4
def sum_clix_of_smaller_tasks(taskNr):
     return Sum([If(period_length[i] <= period_length[taskNr
         ], clix_length[i], 0) for i in range(NR_TASKS)])

acc_test = And([Sum([get_clix_length(i)*get_nr_of_runs(i)
    for i in range(NR_TASKS)]) <= OBSERVATION_WINDOW,
                   And([period_length[i] >= MIN_PERIOD for i in
                        range(NR_TASKS)]),
                   And([clix_length[i] <= MAX_CLIX for i in
                       range(NR_TASKS)]),
                   And([Or(period_length[j] ==
                       OBSERVATION_WINDOW,
                            sum_clix_of_smaller_tasks(j) +
                                MAX_CLIX - 1 <= period_length[j])
                         for j in range(NR_TASKS)])
                   ])

# Add all constraints to the solver
s = Solver()
s.add(no_overlap_c)
s.add(run_time_c)
s.add(atomicity_c)
s.add(no_idling_when_tasks_ready_c)
#
s.add(period_c)
s.add(clix_c)
s.add(earliest_deadline_first_c)


# # A schedulable configuration (for OBS_WINDOW == 20), to
    test the model of the system
# s.add([period_length[0] == 4, period_length[1] == 5,
    period_length[2] == 10])#, X[1][10] == True, X[1][1] ==
    True])
# s.add([clix_length[0] == 1, clix_length[1] == 1,
    clix_length[2] == 3])
# # Another example
# s.add([period_length[0] == 2, period_length[1] == 5,
    period_length[2] == 20])
# s.add([clix_length[0] == 1, clix_length[1] == 2,
    clix_length[2] == 2])
# # Last example
# s.add([period_length[0] == 4, period_length[1] == 5,
    period_length[2] == 10])
# s.add([clix_length[0] == 2, clix_length[1] == 1,
    clix_length[2] == 3])

# For OBS_WINDOW = 21
# s.add([period_length[0] == 3, period_length[1] == 7])

# For OBS_WINDOW = 28
# Not schedulable with EDF
# s.add([period_length[0] == 7, period_length[1] == 14,
    period_length[2] == 4,
#          clix_length[0] == 1,  clix_length[1] == 5,
    clix_length[2] == 1])
# s.add(X[1][15] == True, X[1][16] == True, X[1][17] == True
    , X[1][18] == True, X[1][19] == True)

# Schedulable
# s.add(period_length[0] == 7, period_length[1] == 14,
    period_length[2] == 4,
#         clix_length[0] == 1,  clix_length[1] == 4,
    clix_length[2] == 1)


# ---Check if the acceptance test is sufficient---
# There are different phrasings:
# - The acceptance test implies schedulability
# s.add(Implies(acc_test, And(sched_goal_c)))
# - It should not be possible to have a situation that
    satisfies the acc_test and misses deadlines
s.add(And(acc_test, neg_sched_goal))
# - Or in this simple case, you can also manually check the
    acceptance test and then check if some
# bad schedule can be found
# s.add(neg_sched_goal)
# Check if there are at least some schedulable task sets
    that satisfy the acceptance test
# To be sure that the acceptance test is not far too
    restrictive
# s.add(And(sched_goal_c), acc_test)

# Checking if the acceptance test is necessary: if this
    gives unsat then a system is only schedulable if accepted
# If a test is both necessary and sufficient, then it is
    exact
# s.add(Not(acc_test), And(sched_goal_c))


# print(s.assertions())
value = s.check()
print(value)
if value == sat:
    m = s.model()
    schedule = [[m.evaluate(X[i][j]) for j in range(
        OBSERVATION_WINDOW)] for i in range(NR_TASKS)]
    periods = [m.evaluate(period_length[i]) for i in range(
        NR_TASKS)]
    clixs = [m.evaluate(get_clix_length(i)) for i in range(
        NR_TASKS)]
    print("Periods:␣" + str(periods))
    print("Clix-length:␣" + str(clixs))
    print_schedule(schedule, periods)

print(s.statistics())
```

Bibliography

[1] F. Alder, J. Van Bulck, F. Piessens, and J. T. Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1357–1372, New York, NY, USA, 2021. Association for Computing Machinery.

[2] E. Baccelli, C. Gundogan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wahlisch. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE internet of things journal*, 5(6):4428–4440, 2018.

[3] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at *www.SMT-LIB.org*.

[4] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *Proceedings - Design Automation Conference*, volume 2015- of *DAC '15*, pages 1–6. ACM, 2015.

[5] A. Burns and R. Davis. A survey of research into mixed criticality systems. *ACM computing surveys*, 50(6):1–37, 2018.

[6] A. Burns and A. Wellings. *Real-time systems and programming languages*. International computer science series. Addison-Wesley, Harlow, 2nd ed., repr. edition, 1997.

[7] S. Cleemput, M. A. Mustafa, and B. Preneel. High assurance smart metering. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, volume 2016-, pages 294–297. IEEE, 2016.

[8] L. de Moura and N. Bjørner. Model-based theory combination. *Electronic notes in theoretical computer science*, 198(2):37–49, 2008.

[9] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] L. de Moura and N. Bjørner. z3 - a tutorial. https://www.cs.colostate.edu/ cs440/spring19/slides/z3-tutorial.pdf, s.d.

[11] European Parliament. Consolidated text: Directive 2012/27/eu of the european parliament and of the council of 25 october 2012 on energy efficiency, amending directives 2009/125/ec and 2010/30/eu and repealing directives 2004/8/ec and 2006/32/ec (text with eea relevance), 2021.

[12] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with ARM trustzone. *CoRR*, abs/1704.05600, 2017.

[13] Z. Ji, I. Ganchev, M. O'Droma, L. Zhao, and X. Zhang. A cloud-based car parking middleware for iot-based smart cities: Design and implementation. *Sensors*, 14(12):22372–22393, 2014. Copyright - Copyright MDPI AG 2014; Last updated - 2018-10-05.

[14] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on computer systems*, EuroSys '14, pages 1–14. ACM, 2014.

[15] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2016.

[16] P. Kumar, Y. Lin, G. Bai, A. Paverd, J. S. Dong, and A. Martin. Smart grid metering networks: A survey on security, privacy and open research issues. *IEEE Communications surveys and tutorials*, 21(3):2886–2927, 2019.

[17] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, jan 1973.

[18] P. Maene, J. Gotzfried, R. de Clercq, T. Muller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE transactions on computers*, 67(3):361–374, 2018.

[19] R. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun. Enabling trusted scheduling in embedded systems. In *ACM International Conference Proceeding Series*, ACSAC '12, pages 61–70. ACM, 2012.

[20] J. T. Mühlberg, S. Cleemput, M. A. Mustafa, J. Van Bulck, B. Preneel, and F. Piessens. An implementation of a high assurance smart meter using protected module architectures. In S. Foresti and J. Lopez, editors, *Information Security Theory and Practice*, pages 53–69, Cham, 2016. Springer International Publishing.

[21] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3), jul 2017.

[22] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-time systems*, 10(2):179–210, 1996.

[23] I. Stellios, P. Kotzanikolaou, M. Psarakis, C. Alcaraz, and J. Lopez. A survey of iot-enabled cyberattacks: Assessing attack paths to critical infrastructures and services. *IEEE Communications surveys and tutorials*, 20(4):3453–3495, 2018.

[24] N. Uribe-Perez, L. Hernandez, D. de la Vega, and I. Angulo. State of the art and trends review of smart metering in electricity grids. *Applied Sciences*, 6(3):68, 2016.

[25] P. Van Aubel and E. Poll. Smart metering in the netherlands: What, how, and why. *International journal of electrical power & energy systems*, 109:719–725, 2019.

[26] J. Van Bulck, J. Mühlberg, and F. Piessens. Vulcan: Efficient component authentication and software isolation for automotive control networks. In *ACM International Conference Proceeding Series*, volume Part F132521, pages 225–237. ACM, 2017.

[27] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem-overview of methods and survey of tools. *ACM transactions on embedded computing systems*, 7(3):1–53, 2008.

[28] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE transactions on computers*, 58(9):1250–1258, 2009.