Verification and Cost Optimal Nanosatellite Battery-Aware Schedule Production

P9 Project
Group Deis903e17
Software
Aalborg University
16th January 2017
Title: Verification and Cost Optimal Nanosatellite Battery-Aware Schedule Production

Project: SW9 project

Project period: 4th September - 16th January

Project group: Deis903e17

Members: Anders Lykke Matthiassen, Jacob Nielsen, Oliver Brun Købsted

Supervisor: René Rydhof Hansen, Kim Guldstrand Larsen

Pages: 43
Appendices: 4
Ended: 16th January

Synopsis:
It is important to have a reliable schedule that ensures communication is possible at expected times for systems operating in inaccessible areas, such as space, where communication is not always available. This project aims to develop a tool, easing the process of making schedules as well as reducing the chance of human errors. Said tool is made by utilising UPPAAL CORA and UPPAAL SMC, to produce a schedule, verify it, and finally output data relevant to the schedule, in one automated process. The tool takes a payload and nanosatellite specification as input in order to construct a representative model made in UPPAAL CORA. The result thereof is an optimal schedule that has the greatest return in profit, while still adhering to a set of restrictions, formalised as payload dependencies and battery considerations. The schedule is verified using UPPAAL SMC that performs robustness checks in order to verify its stability. It is important to perform these checks as there are many uncertainties that may dramatically influence the schedule when shipped into orbit.

The content of the report is freely available, but publication (with source reference) may only take place in agreement with the authors.
Preface

This report documents the work of three 9th semester Software Engineering students during the fall of 2017. This report is part of the specialisation in the field Semantics and Verification. The report describes the choices, and research associated with the implementation of a system for generating schedules to nanosatellites, as well as a description of relevant parts for said system.

The system has been developed using Python 3.6.4, and UPPAAL CORA 64bit v4.0.2, and UPPAAL SMC 64bit v4.0.19, and should only be run on Linux. It is recommended to run the tool on a 64bit operating system to utilise all memory. The models presented in section 3.7 CORA Model and section 4.2 SMC Model are made in the previously mentioned versions of UPPAAL and function as part of the implemented system.

Disclaimer: The included figures of said models represent all functionality of the actual models, but may look different from the UPPAAL interface, as they have been made in Latex using TikZ library.

We would like to thank Lars Alminde for meeting and mail corresponding with us, to tell us about GomSpace’s work and discussing ours. As well as verifying the specification data used for our implementation.

Reading Guidance

The report is written in chronological order and should be read as such. Citations are made in accordance with the Vancouver style, meaning they are indicated by the use of numbers, and will look like: [1]. A complete list of cites is found in the bibliography and the cites are ordered by when they appear in the report. Citations at the end of a paragraph, to the right of the last period, references the entire paragraph, whereas cites written on the left side of a period references the given sentence.

Lastly, code found in listings may look different from what is found in the implemented code. No functionality has been changed, this is simply to make the code better presentable and easy to read.
# Table of Contents

Chapter 1  Introduction  

Chapter 2  Problem Analysis  

2.1 The GomX-3 Experience Article  

2.2 Schedule  

2.3 Improvements and Additions  

Chapter 3  Producing a Schedule  

3.1 Solution  

3.2 Workflow  

3.3 Reading the Input  

3.3.1 Payload Specification  

3.3.2 Configuration Specification  

3.4 UPPAAL  

3.5 UPPAAL CORA  

3.6 Battery Models  

3.7 CORA Model  

3.7.1 Processor  

3.7.2 Insolation  

3.7.3 PayloadWindow  

Chapter 4  Verifying a Schedule  

4.1 UPPAAL SMC  

4.2 SMC Model  

4.2.1 Processor  

4.2.2 Scheduler  

4.2.3 PayloadWindow  

4.2.4 EnergySource  

4.3 Final Output  

4.3.1 Gantt Chart  

4.3.2 Queries  

4.3.3 Data From SMC  

Chapter 5  Discussion  

5.1 Inaccuracies and Assumptions  

Chapter 6  Conclusion  

Chapter 7  Future Work  

Glossary
**List of Figures**

45

**List of Tables**

46

**Bibliography**

49

**Appendix A  Variable Loads**

51

**Appendix B  UPPAAL CORA Model Implementation**

53

B.1 Global Declarations

53

B.2 Processor

54

B.3 Scheduler

56

B.4 Insolation

57

B.5 PayloadWindow

58

B.6 System Declarations

59

**Appendix C  UPPAAL SMC Model Implementation**

61

C.1 Global Declarations

61

C.2 Processor

63

C.3 Insolation

63

C.4 EnergySource

64

C.5 Scheduler

65

C.6 PayloadWindow

66

C.7 System Declarations

66

**Appendix D  Raw Schedule Example**

67
When creating a schedule for an embedded system that is sent into space, it is important that the schedule is correct according to the specified behaviour, as bad schedules may result in a depleted battery or other unwanted consequences. Some satellites are only able to communicate with their control centres on Earth at given time intervals and will therefore need a schedule that lasts at least until the next communication opportunity. There are many factors which need to be considered for a schedule, such as time, power, and battery wear. Additionally, the harsh environment of outer space may affect the device and cause it to crash or otherwise hinder its execution. This means the schedule has to be robust, as some unfavourable events may occur, such as a sudden need for a reboot which may cause the system to become delayed. If such an event occur it should be possible to correct for the delay.

This project aims to accommodate this problem by providing a tool that can produce schedules for such systems and test the robustness of the schedules in order to calculate how confident the tool is in the schedule’s correctness.

**Problem statement**  
*How is it possible to produce a schedule that consider multiple requirements, and how is it possible to verify the robustness of such a schedule?*

Generating a schedule with respect to profit and energy consumption has already been done with the GomX-3 mission[1]. Their solution were to generate a schedule, that uses orbit and power predictions as input for the scheduler, which was modelled in UPPAAL Cost Optimal Reachability Analysis (UPPAAL CORA).

We have chosen their work as the starting point for our model, as we consider many of their findings relevant for solving the problem.

Our goal for this project is to produce a validated schedule which is produced by analysing a model, which models the behaviour and battery of a nanosatellite.

We will try to achieve this by creating a model in UPPAAL CORA which will take an input configuration that represent the environment, and the payloads the user is trying to schedule. The schedule will be tested in regards to its robustness by the use of UPPAAL Statistical Model Checking (UPPAAL SMC). When a schedule have been accepted it will be printed in a readable format, such that it can be inspected by the users. The process should be automated such that the only necessary input from the user is the specified configuration. The solution should be generalised so it may be used in other embedded systems and not just nanosatellites.
In this chapter we will present the previous work that have been done in regards to the GomX-3 nanosatellite and then introduce the aspects we want to improve upon along with terminology within this context.

2.1 The GomX-3 Experience Article

We introduced the GomX-3 article as the starting point for our project. They discuss the problem of resource management in regards to power consumption. They want to perform experiments with GomSpace on the GomX-3 nanosatellite, to collect insight on how to model and verify an efficient schedule for the nanosatellite. Efficient in the sense that they want to maximise their payload utilisation while in-orbit.

A payload is a piece of equipment or software which helps the nanosatellite achieve its goal. For example, the GomX-3 nanosatellite wants to use its transmitters or software defined radio module payloads. When we are referring to a payload we will refer to the usage of it, so when we are talking about payload X which uses the radio module to transmit or receive data, we only care about the effect of that payload and not what physical modules that are required e.g. if the nanosatellite need to slew we only want to know how long it will take and not how it is done.

GomSpace have several tools that assist them in producing a schedule for their nanosatellite, but some parts of the process is done by hand. This is a problem as it introduces a greater risk of human error in an environment where mistakes can be fatal for the missions. It is a highly complex task to manually plan the operations that the nanosatellite has to perform while still considering the battery capacity. This problem can be diminished by automating the process of scheduling the payloads, which Bisgaard et al. 2016[1] solved by modelling the GomX-3 nanosatellite in UPPAAL CORA by use of priced time automata. The model allowed them to produce a schedule which could be uploaded and execute on the nanosatellite.

The GomX-3 nanosatellite is equipped with solar panels that it use to power its hardware and charge its battery which it draws upon during eclipse, as the panels may only generate power when exposed to the sun(insolation). The battery’s capacity is a resource that has be tightly monitored as all of the nanosatellite’s operations consumes power, and it will not be able to perform any functions if it is depleted.[1]

The GomX-3 nanosatellite will enter a safe mode if the battery’s State of Charge (SoC) goes below a certain threshold. The safe mode prohibits the execution of all non vital
processes, in order to maintain power until the battery goes above the threshold. Bisgaard et al. 2016[1] made it the primary objective of the schedule to avoid entering safe mode, because no payload can be execute during this mode.

Bisgaard et al. 2016[1] made the observation that some payloads should be dependant on other payloads. If one payload represents the task of collecting data, the payload that is responsible for transmitting the data to Earth may only be executed after completing the aforementioned payload. It would be wasteful to schedule a payload that transmit data if no data have been collected! Additionally, they observed that they would need to schedule two transmitting payloads when one of their data collection payloads had been executed. This means that a simple dependency that checks whether or not a payload have been executed, is not enough. We will sometimes need to also check how many times the payload have been executed, and then later reset this counter in order to let this payload cycle restart.

The GomX-3 nanosatellite communicates with a specific centre one Earth when it has to transmit its data, but the centre is not always in line of sight and it is therefore not able to always transmit its data. In order to transmit its data, it must reside within an interval window which is defined by two timestamps, one for when the payload becomes available and one for when it expires. Bisgaard et al. 2016[1] states that when a data collection payload is started, it must also finish within the window, as payloads that are aborted early or started late are not considered interesting.

2.2 Schedule

A schedule is a set of instructions of when each payload is suppose to start and end, furthermore we define a valid schedule to obey the following criteria (i) payloads should only be executed when dependencies are uphold (ii) payloads with windows should only be executed during their windows (iii) the battery levels may not go below the specified threshold at any point during the entire schedule.

We want to discuss the type of real-time the schedule is in, as it will affect how we should design our model. Three categories exist within real-time systems, hard, firm, and soft. Hard defines that if a payload exceed its deadline it will have catastrophic consequences on the system or it may enter a fail state. A firm system have no critical consequence if a deadline is missed, but the data gathered by the payload will be of no value if it exceeds its deadline. Lastly soft describes a system where the value of the results are diminish when deadlines are missed[2].

We can contextualise these types of real time by examining what would happen if the nanosatellite start a data gathering payload, which it then has to abort because it exceeds its deadline. The hard case does not accurately describe what would happen, as it is not critical that the payload is completed. The nanosatellite would still continue to operate even if the payload was interrupted. Even the payloads that are responsible for receiving new schedules are not critical as the nanosatellite would still be functional. It is however very unfortunate if it does not receive a new schedule, as no work can be done until a new one is received.

As mentioned earlier in section 2.1 The GomX-3 Experience Article, when a data collection
payload is started, it must also finish within the window, as payloads that are aborted early or started late are not considered interesting. We do not believe that soft real-time fits the system description if the data no longer holds any interest when cut short. This leaves us with the last option, firm real-time, which do fit the system. It is okay if payloads are aborted as none of the are critical in order for the system to continue operating, but the value added by starting them are lost when they are aborted.

2.3 Improvements and Additions

In this section we will present the improvements we believe will be beneficial when producing a schedule, these will largely be based on the findings from Bisgaard et al. 2016[1], trying to further develop the automation of schedules with added capabilities. Bisgaard et al. 2016[1] wanted their schedule to be dynamic in the sense that the prices should be updated based on the schedule produced so far. However, it is not possible to change the prices while a query is running in UPPAAL CORA. They sidestepped this limitation by dividing their schedule into multiple disjointed subintervals. They would carry over the state of the model from one subinterval to the next and then update the prices based on the state. All of the subintervals/subschedules would then be conjoined into a single schedule. They noted that this was a trade-off between optimality and being dynamic because it made the schedule more greedy. We believe it is possible to avoid this solution, while still being able to choose between payloads based on the state.

A nanosatellite operating in space will be affected by the environment, such as Earth’s magnetic field or the radioactive sun rays. The schedule will therefore need to be robust, meaning being capable of handling errors, such as; the nanosatellite will sometimes have to restart at unforeseen times, or maybe some dust has covered parts of the solar panels which means less sunlight is absorbed and therefore the batteries will not be recharged as much as expected. This may cause the nanosatellite to fall behind schedule and it will therefore need to handle its new state, for example, some payloads may no longer be possible to execute as one more of their dependencies are not fulfilled. We believe that it is possible to improve upon the solution that Bisgaard et al. 2016[1] presented by introducing robustness, in order to deal with uncertainty.

Wognsen et al. 2015[3] presented a battery lifetime scoring function that evaluated how much wear a SoC profile will have on a battery. One of their major points were that a great amount of wear can be avoided by performing shallow Depth Of Discharge (DOD) cycling. DOD cycling refers to discharging to a certain SoC and the charging to 100% again. Shallow DOD cycling means that the discharges are relatively low such that the SoC stays above, for example, 75% instead of fully discharging the battery. It is desirable for the nanosatellite not to wear the battery out too quickly, as it will reduce the total battery capacity. A battery with a low maximum capacity may complicate the process of producing a schedule that will not drain the battery below the safety threshold. The article did also mention other aspects which may influence the battery lifetime such as dwelling at a certain SoC and the end of charge voltage which also affects the SoC. It would be beneficial to consider these aspects when producing the schedule, and finding a desirable ratio between payload utilisation and battery lifetime preservation.
The best way to preserve the battery’s capacity is to not execute any of the payloads, but this not a good idea as nothing useful is done, as pointed out by Wogensen et al. 2015[3]. They identified that the battery lifetime would be extended if the nanosatellite would not execute any payloads that significantly changed the SoC, this observation is also true for this case even though we are just considering the battery capacity.

In order to ensure that the nanosatellite is doing something worthwhile, we want to introduce the abstract profit variable. A payload’s profit signifies how much value it adds by executing it. Profit is not necessary expressed in currency, it may reflect that payload have gathered data or that data have been transmitted back to Earth. This allows us to prioritise such that the more profitable(valuable) payloads are executed more often than the not so profitable. This does not mean that the not so profitable payloads are never executed, but the scheduler will maximise the amount of profit gained over the duration of the schedule.

The scheduler will need to consider the requirements that was already specified by Bisgaard et al. 2016[1], staying above a SoC threshold, payload efficiency, payload windows etc. and the new ones we propose: maximising profit, reducing battery wear and being robust. These requirements results in the problem statement we presented in the introduction:

How is it possible to produce a schedule that considers multiple requirements, and how is it possible to verify the robustness of such a schedule?

We will try and answer this by modelling the nanosatellite such that the model is capable of considering all of these requirements, and then producing a schedule which can be verified in regards to its robustness.
Producing a Schedule

The requirements for the schedule were defined in the previous chapter. In this chapter we will inspect what we believe can be used to produce a valid schedule. Each of the components that are used to produce the schedule, such as the input configuration specification and the battery model, will be described in more detail in the following sections.

3.1 Solution

The solution we propose to the problem statement is as follows:

- Input a configuration that specifies all of the payloads, battery properties, schedule properties, and validation options
- Produce a UPPAAL CORA model that will find an optimal schedule
- Validate the schedule in UPPAAL SMC to test the robustness and act accordingly to the result
  - Rejected: modify the configuration and produce a new schedule for validation
  - Accepted: output schedule along with robustness results

By doing so we are able to produce an optimal schedule that is verified in regards to the robustness. It is optimal in regards to profit while upholding the requirements, such as minimum acceptable battery level. How the schedule is produced and how the robustness is tested will be explained at a later point.

3.2 Workflow

Prior to defining how the configuration is specified in section 3.3 Reading the Input, we will present the workflow of our solution which provides an overview of how the final system will work. Figure 3.1 displays a flowchart representation of the workflow.

Location 1 - Input that defines the payloads and nanosatellite configuration
The configuration for the system is read in order to properly model the nanosatellite and its payloads.

Location 2 - Generate UPPAAL CORA models from input
The payloads, windows, and other variables is modelled in the UPPAAL CORA model, as specified by the input configuration.
3. Producing a Schedule

**Figure 3.1.** Flowchart that displays the workflow

**Location 3 - Produce best schedule** Make UPPAAL CORA produce a optimal schedule.

**Location 4 - Read schedule to generate a UPPAAL SMC model** The model is able to simulate the execution of the schedule and test its robustness.

**Location 5 - Verify the schedule by running queries** UPPAAL SMC will run the queries on the model and output the results. These queries is made to validate and test the robustness of the schedule.

**Location 7 - Output schedule** The satisfied queries indicate that the schedule is valid, and it will therefore be outputted.

**Location 9 - Tune model parameters** If the queries were not satisfied, the schedule will be discarded and a new one will be produced. In order to produce a new schedule, we will provide UPPAAL CORA with a new configuration. The configuration will be loyal to the one specified by the user, loyal in the sense that it makes no significant changes such as completely removing a dependency, but certain values will be changed, such as the time to complete a payload, and the specified priority of the payload, in order to provoke new choices in the scheduler.
3.3 Reading the Input

In order to model the payloads, battery, and schedule properties, we will need some information from the user. The payloads are described through an input file where the dimensions represents the variables which will be used when producing the schedule. The other properties related to the nanosatellite, is described in a configuration file.

We have filled out both files to provide examples of how to use them. It is expected that the user sets all of the variables in order to represent their system.

3.3.1 Payload Specification

The payloads are defined by eight variables and are defined in a CSV file. An example of this file with five payloads can be seen in listing 3.1. These payloads are based on those from the GomX-3 nanosatellite[1].

The eight variables are Name, Time, Energy, Profit, Deadline, Dependencies, Window, and MaxRuns.

**Name** should indicate what the payload represents. When modelled, a payload’s name will be translated into a number in range 0 to N-1 where N is the amount of payloads. The numeric name will also be used when a payload is referenced in the produced schedule.

**Time** specifies how much time, in minutes, it will take to complete the payload. It is defined by a range in order to allow for uncertainty in regards to the timing of the payloads. It is valid to specify a number that is higher than the orbit duration, but a time may not be below 0.

**Energy** specifies how high a load, in mAh, the payload applies on the battery every minute it is being executed.

**Profit** is expressed within a range from 1 to 5, which signifies how valuable or profitable it is to complete a payload. 1 being the least profitable and 5 being the most. In the example from below, L1 and L2 represents the use of the L-Band transmitter that the GomX-3 nanosatellite use to communicate with other satellites, which allows it to collect valuable data. Which is why it has been given a profit of 5 as it is the most profitable payload. The first payload Slew represents the action of slewing the nanosatellite. This does not directly generate any profit for the nanosatellite as no data is gathered.

**Deadline** is a positive integer and is used to cancel payloads which are delayed too much. The minimum value for the deadline is that of the maximum execution time and the maximum is undefined. The scheduler may chose to wait before executing a payload and we can therefore risk that it is no longer relevant to execute the next payload in the queue. The decision for whether or not to cancel a payload in regards to its deadline goes as follows: *if the time we have spent waiting for the next payload to start plus the time it takes to complete the payload exceeds the deadline; then cancel*. We want to make sure that only payloads that are guaranteed to finish within their deadline start.

**Dependencies** contains a list of other payloads which the current payload is waiting for to be completed. A payload may only be executed if all payloads expressed in its dependencies have previously been completed. The user may specify a dependency of multiple executions of another payload. In the example, X is dependent on L1 and L2 to be completed twice, before it may self be executed.

Some payloads can only be executed in a certain time **Window**, such as when they are above the communication station on Earth. This is why we have added the variable which
restricts when payloads can be executed as it may only happen within the window. The window is specified by a range with the minimum value of 0 and the maximum is equal to the the orbit time. If a payload does not have an associated window it will be allowed to run at any given time, given its dependencies are fulfilled.

In addition we have added the concept of **MaxRuns** which indicates the amount of times each payload can be executed in a payload cycle. A payload cycle is completed when all of the payloads have been executed as many times as described by their MaxRuns value. Whenever a payload has been completed an associated counter, that belongs to the payload, is incremented. When all payloads have reached the MaxRuns value, their individual counter is set to 0 and a new payload cycle begins. All dependencies are also being reset when a new cycle begins, which means that payloads will need to wait for their dependencies to be completed again. This is done to avoid executing the same payload for the duration of the schedule as soon as its dependencies are fulfilled. MaxRuns is a constant and all payloads must be set to 1 or higher.

<table>
<thead>
<tr>
<th>Name</th>
<th>Time</th>
<th>Energy</th>
<th>Profit</th>
<th>Deadline</th>
<th>Window</th>
<th>MaxRuns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slew</td>
<td>2-5</td>
<td>10</td>
<td>1</td>
<td>85</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>L1</td>
<td>90-90</td>
<td>20</td>
<td>5</td>
<td>120</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>L2</td>
<td>90-90</td>
<td>20</td>
<td>5</td>
<td>120</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>X</td>
<td>15-15</td>
<td>10</td>
<td>3</td>
<td>30</td>
<td>0</td>
<td>2 0 0 45-60</td>
</tr>
<tr>
<td>UHF</td>
<td>15-15</td>
<td>15</td>
<td>1</td>
<td>45</td>
<td>0</td>
<td>0 0 1 0 20-65</td>
</tr>
</tbody>
</table>

**Listing 3.1.** Example of how five payloads can be defined

### 3.3.2 Configuration Specification

The nanosatellite and its battery is specified by a configuration which consists of a list of variables and constants. The configuration is read from an INI file which we have provided an example of below in listing 3.2. The configuration is divided into three sections System, CORA, and SMC.

The System specific variables describes the nanosatellite and the time constraints. The variables are:

- schedule_length
- orbit_time
- battery_capacity
- idle_cost
- safe_threshold
- soc
- rec_rate

The **schedule_length** defines how long the produced schedule should be, in minutes. **orbit_time** defines how long it takes to complete one orbit, in minutes. The **battery_capacity** is an constant that defines the maximum capacity of the battery. We do not expect that the user defines every single operation that the nanosatellite is capable of performing, which is why we have made an abstraction by introducing the **idle_cost** variable. **idle_cost** is the load that is imposes on the battery at all times, even when the nanosatellite is idling, not currently executing any of the defined payloads. This will allow us to disregard some of the payloads that the nanosatellite may perform as they
can be grouped together as one single background process which is always being executed. The user should only consider payloads which can be executed in parallel to the defined payloads and only chose those which energy and time cost are trivial. Failing to do so may result in a non representative model of their nanosatellite and therefore a schedule where our guarantees may not be valid.

Based on the findings from Bisgaard et al. 2016[1] we have decided to include the safe_threshold variable. This variable is used to determine when the nanosatellite’s SoC is too low to continue executing the schedule. If the nanosatellite go below this threshold, it is in danger of depleting the battery, which means it will not be able to communicate with Earth for a period of time. This will result in a huge loss of potential profit as we are no longer able to execute profitable payloads and should therefore be avoided. It is critical that the schedules we produce will never go below this threshold.

rec_rate defines the recharge rate such that the value is added to the battery’s capacity every minute that the nanosatellite is in insolation.

We have three variables that are related to the UPPAAL SMC model where two of them is used in its battery model.

- f_rate
- ac_width
- certainty
f_rate is a rate that is specific to the battery model we have chosen, and it is used to simulate the recovery effect of batteries. The recovery effect will be described later in section 3.6 Battery Models, together with the ac_width constant. The constant certainty specifies how certain we require that UPPAAL SMC should be in its results. UPPAAL SMC is described in section 4.1 UPPAAL SMC.

At this point the CORA section does not contain any variables, the section is used as a placeholder for future versions of the system, as the model could be expanded to include variables specific to UPPAAL CORA.

Now that payloads and nanosatellite is specified, we are able to start modelling the system.

### 3.4 UPPAAL

We have chosen to use UPPAAL Cost Optimal Reachability Analysis[4][5] and UPPAAL Statistical Model Checking[6][7] for producing and verifying the schedule. We will describe the common charismatics for both versions and then introduce them before their respective models are presented.

Common for all versions of UPPAAL, is that; it has global decelerations and templates. An example template can be seen in figure 3.2. One model may have several templates, each with its own local decelerations. The template itself consists of one to many locations and edges that connects the locations.

One location in each template must be initial in order for UPPAAL to determine the starting state. In addition a location can be urgent, meaning time is not allowed to pass while the location is active, or committed which is a stronger expression than urgent. Committed indicates that activating an outgoing transition is of priority. If any committed location in the model is part of the current state, at least one transition from a committed location must be part of the next transition. Edges may be decorated with; selects, guards, synchronisations, and updates. Selects are used for introducing new temporary variables, and are coloured yellow. Guards are used to ensure that an edge is not activated prematurely, and are coloured green. Synchronisations are used for activating multiple edges across templates simultaneously, and are coloured light blue. If an exclamation mark is used, it indicates that it is calling a synchronisation, whereas a question mark is receiving one. Finally updates are used to change variable values, and to call functions written in declarations, and are coloured dark blue.

Locations can be given a name and an invariant. An invariant must always be evaluated to true e.g. if a location have the invariant time <= 5, at time five there will be a chance of state. Invariants are coloured pink.

Also common for the versions of UPPAAL, is that queries can be written in order to ensure sustain properties are upheld, such as; is some location reachable, and will time ever exceed some amount.

### 3.5 UPPAAL CORA

UPPAAL Cost Optimal Reachability Analysis (UPPAAL CORA) is a branch of UPPAAL, that uses linearly priced timed automata to find optimal paths satisfying certain goals,
3.5. UPPAAL CORA

![Diagram of UPPAAL CORA template]

Figure 3.2. Example template

based on lowest accumulated cost\[4\]. Due to the underlying structure of UPPAAL CORA it is only possible to do reachability checks, and does not allow for liveness or deadlocks checks. Lastly, UPPAAL CORA cannot guarantee termination unless the modelled system is acyclic and clocks are bound by invariants\[8]\[5\].

UPPAAL CORA can be configured in a set of different way depending on the selected options, one of these options is the best diagnostic trace. Best mean that UPPAAL CORA ensures that the path found with this option selected is the best path with respect to minimising cost. Figure 3.3 will be used in an example that illustrates how cost accumulate over time.

![Diagram of full UPPAAL CORA template]

Figure 3.3. Example template for UPPAAL CORA

From figure 3.3 we see a UPPAAL CORA template with four locations and four edges connecting them. Three of the locations have invariants bound to the clock time and an associated cost rate. For every time unit that passed in the initial location A, the cost will increase by 1. The guard and invariant enforces that it will remain there for three to five units of time before transiting to one of the two neighbouring location, either B1 or B2.

In B1 the cost will increase with a rate of 3 per unit of time and will stay there until time reaches 10. Alternatively it may transition to B2 where the cost rate is 4 and will stay there until time is between 7 and 10. After transitioning to one of these locations it will finally reach location C.

When UPPAAL CORA traverse the model it will produce a number of traces that reflects the possible choices it is able to make, choices such as for how long to wait before taking a transition and which one to take. For example, it can wait for as long as A allows, and then
can take the transition to $\mathbf{B1}$, the cost of reaching $\mathbf{C}$ will then be 20 at time 10. Another possibility is to once again wait 5 time units in $\mathbf{A}$ before taking the transition to $\mathbf{B2}$ before transitioning to $\mathbf{C}$ at time 7, the cost of reaching $\mathbf{C}$ will then be 13 at time 7, it will then be able to wait in $\mathbf{C}$ until time 10 with no additional cost. UPPAAL CORA will now be able to discard the trace with the cost of 20 because it observed a better path.

$$E <> \text{Template}.C$$ (3.1)

To get the optimal path to $\mathbf{C}$ we set diagnostic trace to best and run the query seen in 3.1, the query asks if $\mathbf{C}$ is reachable. This will produce a path that is equal to best path from our example.

### 3.6 Battery Models

In order to implement the solution, it is necessary to monitor the SoC and correctly simulate the charging and discharging of the battery. To do this, we need some way of modelling a battery. This section will discuss three distinct battery models that all have different qualities. An important factor to consider when examining the battery models is the recovery effect, which can have a significant impact on the expected charge lifetime\[9\]. The recovery effect charges the battery after a discharge by diffusing the charge evenly in the battery. This means that a battery will regain some charge over time. The performance of the different models will be compared to actual measurements of a lithium-ion battery, which have been observed by Rakhmatov et al. 2003\[10\], unfortunately we are unable to obtain the specification for the lithium battery used in the paper as they are not listed. The measurements of the different battery models is the work of Jongerden and Haverkort 2009\[9\] who used the battery specification and measurements from Rakhmatov et al. 2003\[10\].

The first model, the Ideal model, is simplistic due to it only having two variables to determine the batteries lifetime($L$), capacity($C$) and load current($I$). Capacity relates to the battery’s amount of amp-hours, and load current is the constant discharge on the battery in amps. The calculation for finding the lifetime can be seen in equation (3.2).

$$L = \frac{C}{I}$$ (3.2)

A battery’s lifetime under constant and variable load, and the Ideal model’s estimate can be seen in table 3.1. The columns “Meas, min” hold the actual measured readings of the lithium-ion battery under different loads The next column, “Ideal, min”, holds the Ideal estimation. The Ideal model overestimate the expected lifetime in all cases. It would seem that lower amps with constant loads often results in better predictions than the other tests. This is not always the case as T5 is better at predicting the lifetime than T6, even though T5 has a higher load. During variable loads all cases overestimate by a substantial amount, the closest approximation is C5 during variable loads with a 18.5% overestimation. The data indicates that the Ideal model does a poor job of estimating the actual lifetime of a battery, which is properly because the model assumes linear effect for lifetime estimation. Additionally, for variables loads the Ideal model does not consider the recovery effect which further deviate the lifetime from the measured values.
3.6. Battery Models

<table>
<thead>
<tr>
<th>Test</th>
<th>I, amps</th>
<th>Meas, min</th>
<th>Ideal, min</th>
<th>∆T</th>
<th>%Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>222.7</td>
<td>141.0</td>
<td>181.3</td>
<td>40.3</td>
<td>28.58%</td>
</tr>
<tr>
<td>T2</td>
<td>204.5</td>
<td>156.6</td>
<td>197.4</td>
<td>40.8</td>
<td>26.05%</td>
</tr>
<tr>
<td>T3</td>
<td>108.3</td>
<td>307.8</td>
<td>372.8</td>
<td>65</td>
<td>21.12%</td>
</tr>
<tr>
<td>T4</td>
<td>107.5</td>
<td>312.0</td>
<td>375.6</td>
<td>63.6</td>
<td>20.38%</td>
</tr>
<tr>
<td>T5</td>
<td>94.9</td>
<td>358.2</td>
<td>425.4</td>
<td>67.2</td>
<td>18.76%</td>
</tr>
<tr>
<td>T6</td>
<td>84.3</td>
<td>397.2</td>
<td>478.9</td>
<td>81.7</td>
<td>20.57%</td>
</tr>
<tr>
<td>T7</td>
<td>75.5</td>
<td>448.2</td>
<td>534.8</td>
<td>86.6</td>
<td>19.32%</td>
</tr>
<tr>
<td>T8</td>
<td>28.0</td>
<td>1248</td>
<td>1442</td>
<td>194</td>
<td>15.54%</td>
</tr>
<tr>
<td>T9</td>
<td>19.5</td>
<td>1818</td>
<td>2071</td>
<td>253</td>
<td>13.92%</td>
</tr>
<tr>
<td>T10</td>
<td>3.0</td>
<td>12690</td>
<td>13458</td>
<td>768</td>
<td>6.05%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test</th>
<th>Meas, min</th>
<th>Ideal, min</th>
<th>∆C</th>
<th>%Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>54.5</td>
<td>70.8</td>
<td>16.3</td>
<td>29.91%</td>
</tr>
<tr>
<td>C2</td>
<td>73.3</td>
<td>91.9</td>
<td>18.6</td>
<td>25.38%</td>
</tr>
<tr>
<td>C3</td>
<td>88.3</td>
<td>108.5</td>
<td>20.2</td>
<td>22.88%</td>
</tr>
<tr>
<td>C4</td>
<td>136.0</td>
<td>163.0</td>
<td>27</td>
<td>19.85%</td>
</tr>
<tr>
<td>C5</td>
<td>182.7</td>
<td>216.5</td>
<td>33.8</td>
<td>18.50%</td>
</tr>
<tr>
<td>C6</td>
<td>59.0</td>
<td>74.7</td>
<td>15.7</td>
<td>26.61%</td>
</tr>
<tr>
<td>C7</td>
<td>61.1</td>
<td>66.9</td>
<td>15.8</td>
<td>30.92%</td>
</tr>
<tr>
<td>C8</td>
<td>55.0</td>
<td>70.8</td>
<td>15.8</td>
<td>28.73%</td>
</tr>
<tr>
<td>C9</td>
<td>54.9</td>
<td>70.8</td>
<td>15.9</td>
<td>28.96%</td>
</tr>
<tr>
<td>C10</td>
<td>142.7</td>
<td>171.3</td>
<td>28.6</td>
<td>20.04%</td>
</tr>
</tbody>
</table>

Table 3.1. Comparison of actual measurements and predictions from the Ideal model. The specification for the variable loads can be found in table A.1.

The next model, Peukert Model, introduces a few new variables in comparison to the Ideal model. To better represent the battery, a non-linear model is needed as it is naive to represent it linearly. To do so, Peukert updates the formula to what can be seen in equation (3.3).

\[
L = \left( \frac{C}{I^k} \right) \tag{3.3}
\]

- L - lifetime in hours
- C - capacity in AH
- I - load current in amps
- k - Peukert Exponent

According to Martin 1999[11] C is the battery’s capacity and k is a value that often lies between 1.2 and 1.7. This equation describes the lifetime for constant loads. It can be hard to determine the exact value of k without actual testing the battery, but it is sometimes provided by the manufacturers in their data sheets.

Table 3.2 showcases the estimates using Peukert. In most cases Peukert overestimate, except for one instance in test T10 where it underestimate with 3.17%. Peukert seems to become more accurate with smaller amps, alike to how the Ideal model behaved. However, it still has a high mismatch compared to measured values with variable loads, this is due to Peukert’s model is not taking the recovery effect into account.

The Kinetic Battery Model (KiBaM) uses two wells as an abstract concept for representing a battery. It consist of an available and a bound charge as shown in figure 3.4. The total capacity of the battery is represented by the two wells where their width is described by the constant c. Charge can only be drawn from the available charge well and when the available charge well’s height \( h_2 \) is lower than the bound charge well’s height \( h_1 \), charge start to flow from bound to available until both wells are at equal height. The rate of this flow is determined by the value k, which simulate the recovery effect of the battery. The calculation of \( y_1 \) and \( y_2 \) can be seen in equation (3.4) and equation (3.5) along with a short description of the variables used in the equations.
### Table 3.2. Comparison of actual measurements and predictions from the Peukert model. The specification for the variable loads can be found in table A.1.

<table>
<thead>
<tr>
<th>Test</th>
<th>I, amps</th>
<th>Meas, min</th>
<th>Peukert, min</th>
<th>$\Delta T$</th>
<th>Peukert, min</th>
<th>$\Delta C$</th>
<th>%Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>222.7</td>
<td>141.0</td>
<td>154.5</td>
<td>13.5</td>
<td>9.57%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>204.5</td>
<td>156.6</td>
<td>168.4</td>
<td>11.8</td>
<td>7.54%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>108.3</td>
<td>307.8</td>
<td>321.3</td>
<td>13.5</td>
<td>4.39%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>107.5</td>
<td>312.0</td>
<td>323.7</td>
<td>11.7</td>
<td>3.75%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>94.9</td>
<td>358.2</td>
<td>367.5</td>
<td>9.3</td>
<td>2.60%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T6</td>
<td>84.3</td>
<td>397.2</td>
<td>414.4</td>
<td>17.2</td>
<td>4.33%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T7</td>
<td>75.5</td>
<td>448.2</td>
<td>463.6</td>
<td>15.4</td>
<td>3.44%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T8</td>
<td>28.0</td>
<td>1248</td>
<td>1270</td>
<td>22</td>
<td>1.76%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T9</td>
<td>19.5</td>
<td>1818</td>
<td>1835</td>
<td>17</td>
<td>0.94%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T10</td>
<td>3.0</td>
<td>12690</td>
<td>12288</td>
<td>-402</td>
<td>-3.17%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 3.4. The two wells available(a) and bound(b)

\[
y_1(t) = cCe^{-k't} + \left(y_0k'c - I\right)(1 - e^{-k't}) - \frac{Ic(k't - 1 + e^{-k't})}{k'}
\]

\[
y_2(t) = (1 - c)Ce^{-k't} + y_0(1 - c)(1 - e^{-k't}) - \frac{I(1 - c)(k't - 1 + e^{-k't})}{k'}
\]

C is capacity in Ah, e is Euler’s number, k’ is $k/c(1 - c)$, k is a constant, t is time in hours, c is the ratio between available and bound charge, and I is the load current applied on the battery.

Table 3.3 shows the predictions when using KiBaM. Under constant load the results vary, and the most precise predictions can be found when the amps are above 110 or below 20. Interesting for variable loads, is that all of the predictions from KiBaM underestimate compared to the measured values, some are fairly high as C7 underestimate by 40.31%.

Looking at all the results for the three battery models, it show that Peukert’s model give overall better estimations for variable loads than Ideal and KiBaM. But the advantage of using KiBaM over Peukert’s model under variable load, is that KiBaM seems to always underestimate, which is attractive for our case. Underestimating ensures that we will never run into a case where the prediction cause the actual system to run out of energy.
### Constant load

<table>
<thead>
<tr>
<th>Test</th>
<th>I, amps</th>
<th>Meas, min</th>
<th>KiBaM, min</th>
<th>ΔT</th>
<th>%Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>222.7</td>
<td>141.0</td>
<td>139.9</td>
<td>-1.1</td>
<td>-0.78%</td>
</tr>
<tr>
<td>T2</td>
<td>204.5</td>
<td>156.6</td>
<td>156</td>
<td>-0.6</td>
<td>-0.38%</td>
</tr>
<tr>
<td>T3</td>
<td>108.3</td>
<td>307.8</td>
<td>331.4</td>
<td>23.6</td>
<td>7.67%</td>
</tr>
<tr>
<td>T4</td>
<td>107.5</td>
<td>312.0</td>
<td>334.1</td>
<td>22.1</td>
<td>7.08%</td>
</tr>
<tr>
<td>T5</td>
<td>94.9</td>
<td>358.2</td>
<td>384</td>
<td>25.8</td>
<td>7.20%</td>
</tr>
<tr>
<td>T6</td>
<td>84.3</td>
<td>397.2</td>
<td>437.5</td>
<td>40.3</td>
<td>10.15%</td>
</tr>
<tr>
<td>T7</td>
<td>75.5</td>
<td>448.2</td>
<td>493.3</td>
<td>45.1</td>
<td>10.06%</td>
</tr>
<tr>
<td>T8</td>
<td>28.0</td>
<td>1248</td>
<td>1401</td>
<td>153</td>
<td>12.26%</td>
</tr>
<tr>
<td>T9</td>
<td>19.5</td>
<td>1818</td>
<td>2029</td>
<td>211</td>
<td>11.61%</td>
</tr>
<tr>
<td>T10</td>
<td>3.0</td>
<td>12690</td>
<td>13417</td>
<td>727</td>
<td>5.73%</td>
</tr>
</tbody>
</table>

### Variable load

<table>
<thead>
<tr>
<th>Test</th>
<th>Meas, min</th>
<th>KiBaM, min</th>
<th>ΔT</th>
<th>%Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>54.5</td>
<td>36.3</td>
<td>-18.2</td>
<td>-33.39%</td>
</tr>
<tr>
<td>C2</td>
<td>73.3</td>
<td>55.7</td>
<td>-17.6</td>
<td>-24.01%</td>
</tr>
<tr>
<td>C3</td>
<td>88.3</td>
<td>71.4</td>
<td>-16.9</td>
<td>-19.14%</td>
</tr>
<tr>
<td>C4</td>
<td>136.0</td>
<td>123.6</td>
<td>-12.4</td>
<td>-9.12%</td>
</tr>
<tr>
<td>C5</td>
<td>182.7</td>
<td>175.7</td>
<td>-7</td>
<td>-3.83%</td>
</tr>
<tr>
<td>C6</td>
<td>59.0</td>
<td>41.1</td>
<td>-17.9</td>
<td>-30.34%</td>
</tr>
<tr>
<td>C7</td>
<td>51.1</td>
<td>30.5</td>
<td>-20.6</td>
<td>-40.31%</td>
</tr>
<tr>
<td>C8</td>
<td>55.0</td>
<td>38.1</td>
<td>-16.9</td>
<td>-30.73%</td>
</tr>
<tr>
<td>C9</td>
<td>54.9</td>
<td>34.8</td>
<td>-20.1</td>
<td>-36.61%</td>
</tr>
<tr>
<td>C10</td>
<td>142.7</td>
<td>131.7</td>
<td>-11</td>
<td>-7.71%</td>
</tr>
</tbody>
</table>

**Table 3.3.** Comparison of actual measurements and predictions from the KiBaM. Specification for the variable loads can be found in table A.1

Ideal is inferior to Peukert’s and KiBaM in almost all of the predictions, a summary of the three different battery models can be found below.

- Ideal model - linear representation of the battery with no support of recovery effect
- Peukert model - non-linear representation of the battery with no support of recovery effect
- KiBaM - abstract representation of the battery with support of recovery effect

UPPAAL CORA can not use KiBaM or Peukert’s model because it only support cost with a linear rate and only natural numbers, the Ideal model will implemented in the UPPAAL CORA model.

### 3.7 CORA Model

Based on the information gained during the previous sections we have constructed a model in UPPAAL CORA capable of producing schedules for the nanosatellite. The UPPAAL CORA model takes a set of payloads with some requirements which works as constrains, and other descriptive values, as described in section 3.3 Reading the Input, in order to produce a schedule that upholds the specifications. Such descriptions are fed to the model via our own translator program.

The UPPAAL CORA model is composed of four templates; Processor, Scheduler, Insolation, and PayloadWindow. They model different aspects of the nanosatellite and its environment, and are all described separately below.

The templates Processor and Scheduler, are roughly based on those presented by Bisgaard et al. 2016[1], but have been modified to fit our context as to handle the added parameters. The model is made to optimise in regards to profit, see section 3.3 Reading the Input, this is done as the battery level is rarely a concern[12]. We therefore chose the approach where we found the schedule that would yield the highest profit, according to the input specification, while having the battery level in mind. As KiBaM is pessimistic in the sense that it underestimates the SoC it would be a good choice to implement. However due to the complexity of KiBaM and the need for non-linear rates, UPPAAL CORA is not the right tool for this, therefore we have chosen to model the ideal battery model.

Since there is uncertainty in regards to how long it takes to complete some payloads, we chose to assume the worst. This means that it is likely that more energy will be consumed
during simulation than in orbit.

When setting the cost rate, see section 3.5 UPPAAL CORA, the user defined profit is used. When the user defines the profit, large numbers represents that the payload is more profitable than payloads with lower profit values. In UPPAAL CORA, when running a query where we extract the best trace, it is defined as the trace with the lowest cost. Because of this we have to calculate the cost as a function based on the defined profit, how this is done can be seen in listing 3.3.

The function `calcCost()` calculates the cost rate by subtracting the defined profit from 5. However, if the nanosatellite is within a window where another payload can only be performed, the cost rate will be multiplied by two in order to make it more expensive to execute. This is done to make it more attractive for the model to chose the payloads that are restricted by a window, versus those that may be executed at any time.

```c
void calcCost () {
    .
    .
    .
    else {runCost = 5 - Profit[active]; return;}
    for (i=0; i < Windows;i++){
        if( pen && RunInWindow[i][active] == 0){
            runCost = (5 - Profit[active]) *2; return;
        }
    }
}
```

*Listing 3.3.* Function `calcCost()`, used for calculating the cost rate

The only query that is run in the UPPAAL CORA model is the one seen in query 3.6. In this query `\(t_{time}\)` is the accumulated time, and `ScheduleLength` is the desired length of the schedule. This query is run with the trace setting `Best`, meaning it will find the schedule that results in the lowest cost, i.e. the highest profit. Therefore the result of this query is the schedule with the highest profit, and this is what the UPPAAL SMC model will be build upon.

\[ E <> t_{time} == ScheduleLength \] (3.6)

### 3.7.1 Processor

We will describe the Processor template first as it is the base for the entire model. It models the nanosatellite’s processor in the sense that it models when the processor should be idling or executing a payload, as well as consuming power while doing so, as seen in figure 3.5.

The processor template synchronize with the scheduler every time it is ready to execute a payload, the scheduler will then send back a synchronization if the Processor is allowed to do so. If it is not allowed to execute the payload it will start idling until a change in its environment occurs, indicated by the synchronization `win?`. If the Scheduler indicates it is possible to execute the payload, it may be the case that the environment does not allow it, e.g. if the payload is dependant on a window which the nanosatellite is not currently
in, the processor will try to select another payload to execute. If the Scheduler approves and all dependencies are fulfilled, the Processor will transition to **Running** where it will remain for the worst case execution time for the selected payload before it transitions to **Wait** where it will remain for the duration of the payloads deadline.

![Processor template](image)

**Figure 3.5.** Processor template

**Scheduler**

The scheduler always checks if there are enough remaining energy. When receiving the synchronisation **ready**? it will answer back with **run!**. The template is also responsible for deadlocking if the battery SoC goes below the safety threshold, thereby discarding the current trace which will force it to try another one. If all traces results in a deadlock it was not possible to produce a schedule with the specified parameters. The Scheduler template can be seen in figure 3.6.

![Scheduler template](image)

**Figure 3.6.** Scheduler template
3.7.2 Insolation

The model consists of two locations inSun and inEclipse, as seen in figure 3.7. The nanosatellite can only recharge when it has a clear line of sight to the sun, which is indicated by the location inSun. The recharging is done on the looping edge on inSun with the function increaseBattery(). This is done four times per orbit, rather than for each time unit, in order to minimize the state space. When half of the OrbitTime have passed, modelling the time it takes the nanosatellite to complete one half of its orbit, the model is forced to transition to inEclipse. It is assumed that the nanosatellite always spends exactly half its time in insolation.

\[
\begin{align*}
\text{ins} & \geq \text{OrbitTime} / 2 \\
& \quad \land \text{chargeCount} = 4 \\
\text{splitTime} & \leq \text{OrbitTime} / 8 \\
& \quad \land \text{ins} \leq \text{OrbitTime} / 2 \\
\text{splitTime} & > \text{OrbitTime} / 8 \\
\text{increaseBattery}() & \\
\end{align*}
\]

\[
\begin{align*}
\text{ins} & > \text{OrbitTime} \\
& \quad \land \text{chargeCount} = 8 \\
\text{ins} & := 0 \\
\text{splitTime} & = 0 \\
\text{chargeCount} & = 0 \\
\text{splitTime} & > \text{OrbitTime} / 8 \\
\text{subIdle}() &
\end{align*}
\]

\textbf{Figure 3.7.} Insolation template

A few assumptions have been made when this template was modelled, and there is an inaccuracy as we only recharge four times during insolation. This may result in a suboptimal schedule as executing a payload may cause the SoC to fall below the threshold, invalidating the trace, where the SoC should be higher potentially resulting in it being possible to execute the payload. However, we believe this to be an edge-case that will rarely occur. In figure 3.8 we see a graph over the SoC (left) while doing nothing but charging. On the right, is a close-up of the same graph. The blue line shows what the SoC should be and the orange what it is with our implementation. Clearly our implementation is not entirely accurate, but neither is the ideal battery model, and because of this we believe a small margin of error is acceptable given the significantly reduced execution time.

\textbf{Figure 3.8.} Graphs illustrating SoC, with different granularity. Bottom graph is a zoom in of the top graph.
To find the effect of execution time reduction we produced $2 \times 10$ schedules of 12 hours each, ten with our implementation and ten where recharge was added for every unit of time elapsed. The differences in time spend to produce these were quite significant and can be seen in table 3.4. The worst case when charging every time unit was 138.76% more than when using our implementation.

<table>
<thead>
<tr>
<th>Granularity 1(s)</th>
<th>Granularity 11(s)</th>
<th>Time Increase(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>39.63</td>
<td>21.13</td>
</tr>
<tr>
<td>Worst</td>
<td>64.68</td>
<td>27.09</td>
</tr>
<tr>
<td>Average</td>
<td>48.13</td>
<td>24.89</td>
</tr>
</tbody>
</table>

*Table 3.4. Execution times when generating a 12 hour schedule, using different granularity for insolation*

Our implementation would recharge at time 11, 22, 33 and, 44 as the orbit time were set to 90, which is why the granularity in the second column is set to 11. Additionally it is assumed that the nanosatellite are always starting in insolation, resulting in our model only being able to produce an accurate schedule for the nanosatellite that matches our starting point.

### 3.7.3 PayloadWindow

The last template is the PayloadWindow, this template have one instance for each defined window. The template is used to model the windows as described in section 3.3 Reading the Input. Windows is an important restriction when producing a schedule as some payloads may be pointless to execute outside of their respective windows. When a change to a window occurs it will synchronise with the processor on the channel win, if it is in Idle which indicates it may execute some payload. Otherwise it will transition to the next location updating some variables indicating if the nanosatellite is within the window or not. A figure of this template can be seen in figure B.1.

It is assumed that an orbit is constant, meaning if the orbit starts with the nanosatellite over Paris the orbit will also end there.
Verifying a Schedule

After having produced a schedule it is now time to verify the validity thereof. In this chapter we will describe how it is possible to test different aspects of the schedule, as well as provide the user with data about it.

4.1 UPPAAL SMC

UPPAAL SMC is one of the newer versions of UPPAAL, it works with statistical model checking for more efficient analyses, as it avoids the exhaustive exploration of the state-space. Statistical exploration have the added benefit of significantly less memory consumption, meaning that the computer’s physical memory is rarely the limit when running a query. This flexibility allows UPPAAL SMC to be used in other areas which was previous too memory intensive for other UPPAAL tools[7].

As mentioned in section 3.5 UPPAAL CORA, UPPAAL CORA only support linear rates, UPPAAL SMC facilitates non-linear rates and differential equations, which makes it possible to implement KiBaM, based on our information from section 3.6 Battery Models this will greatly improve our battery estimations. UPPAAL SMC looks similar to previous tools in UPPAAL in form of GUI, but it adds new functionality which will be showcased in the rest of this section. This will include the probability and simulate queries, and how the results of these queries can be interpreted and understood.

We will use the model in figure 4.1 as our example for a query that finds the probability for whether or not a specific location is reachable.

![Figure 4.1. Model for showcasing probability](image)

\[
Pr[<= 1](<> Process.L4)
\] (4.1)

With this model in place, we are now able to ask if it can reach a specified location by the query shown in query 4.1. Pr is a keyword to denote that we want to find the probability,
square brackets describe the amount of time to pass. It is possible to specify which clock to use by adding the variable name of the clock at the beginning, in between the square brackets. 

Alternative you can swap out time for discrete steps by using the hash(#) symbol in place of the clock variable. The expression to be checked is written within the parentheses. 

Process refers to the template, and the diamond (<> ) describe that a path can reach location L4. Another option is to use square bracket ([ ]) which mean that a property must be true for all possible states. Running query 4.1 returns the following probability range [0.18744,0.287382] with a certainty of 95%, when rounded to one decimal the result is 18.7%-28.7%.

The simulate query is another feature introduced in UPPAAL SMC, it is used to run the modelled system a number of time and capture the values of some specified variables at the same time. To illustrate this a new model is needed, we will use figure 4.2 along with its declarations in listing 4.1. Three clock are defined in this example, x, y, and, t.

\begin{verbatim}
clock x, y, t;
\end{verbatim}

\textit{Listing 4.1. Declarations}

\begin{verbatim}
x ’== 1 && y ’== 0 && t <= 10
x ’== 0 && y ’== 2 && t <= 10
t >= 5
x >= 5
t = 0
\end{verbatim}

\textit{Figure 4.2. Simple model}

We will run both of the queries in 4.2 which specifies that the model should be simulated once and five times respectively over the duration of 100 time units. The queries also captures the value of x and y.

\begin{verbatim}
simulate 1 [<= 100] {x,y} and simulate 5 [<= 100] {x,y}
\end{verbatim}

\textit{(4.2)}

The result of the two queries is displayed as graphs in figure 4.3 and figure 4.4, where the value axis is the value of x and y, and the time axis is the time spent. Examining the graph we can observe that the value x tends be about 50 at 100 time units. From that results a new question can be raised “What is the probability that x is above or equal to 50 when 100 time units have passed?” in 4.3 a query have been constructed to answer the question.

\begin{verbatim}
Pr[<= 150](<> x >= 50)
\end{verbatim}

\textit{(4.3)}
Like the previous probability example, the query results in a range that describes the probability that \( x \) will be higher or equal to 50 when 150 time units have passed, which in this case results in about a 98\% to 100\% probability. It is possible to get more information from the query in the form of seven different graphs (Probability Density Distribution, Probability Density Confidence Intervals, Probability Distribution, Probability Confidence Intervals, Cumulative Probability Distribution, Cumulative Probability Confidence Intervals, and Frequency Histogram).

Frequency Histograms shows the count on the vertical axis and run duration in time on the horizontal axis, each of the columns represent a time where a number of runs have fulfilled the property. The first column show that out of the 263 runs, three of them had
4. Verifying a Schedule

observed the value $x$ to be higher or equal to 50 at 80.8 time units. This histogram show that with a 99% certainty $x$ will at the earliest be 50 or higher at 80.8 and latest 112 time units.

![Figure 4.5. Frequency histogram generated from query 4.3](image)

The Cumulative Probability Distribution graph shows the accumulated probability over time. With a 99% certainty, at 112 time units there is almost a 100% probability for $x$ to be equal or above 50, and at 96.4 time units there is about a 50% probability.

![Figure 4.6. Cumulative probability distribution generated from query 4.3](image)

The graphs generated by UPPAAL SMC from the queries can provide useful insight about the probability for a query, an example of this can be seen the above graphs, figure 4.5 figure 4.6. When running the query it only returns the probability, but the graphs reveal that if we go to 112 or more time units, $x$ will always be 50 or higher.
4.2 SMC Model

UPPAAL SMC is chosen because UPPAAL CORA handles randomness poorly making it inadequate to perform robustness check, because of its deterministic approach, compared to UPPAAL SMC which can perform undeterministic choices. This section will focus on how a UPPAAL SMC model can be constructed to provide further insight about the schedules robustness produced by the UPPAAL CORA model. Additionally the UPPAAL SMC model will need variables mentioned in section 3.3 Reading the Input.

Since UPPAAL SMC allows clocks to have non-linear rate along with rational numbers, it is possible to implement KiBaM, see section 3.6 Battery Models. The advantage of using the battery model over ideal, implemented in UPPAAL CORA, is that KiBaM is pessimistic in the sense that KiBaM tends to underestimate the actual battery levels. Because of this KiBaM will be implemented in the UPPAAL SMC model, instead of the ideal.

Our UPPAAL SMC model consist of four templates plus one instance of a template for each window described in the payload specification. However, it is required that at least one window is defined in the input. This makes a total of 5 plus templates, which are Processor, Insolation, EnergySource, Scheduler, and PayloadWindow(i).

The UPPAAL SMC model is based on a UPPAAL implementation of KiBaM [13], and our UPPAAL CORA model. The UPPAAL implementation of KiBaM covers scheduling with uncertainty, combined with capturing battery levels during execution of the schedule. The rest of this section we will go over each of the important points in the model, this include the concept of recharging, windows for payloads, payload dependencies etc.

4.2.1 Processor

The initial version of this template was based on the UPPAAL implementation of KiBaM, which we have then modified to fit our context. The processor template describes the status of the nanosatellites processor, it indicates if it is currently executing a payload, waiting for a payloads deadline to be reached or idling.

As the UPPAAL SMC model receives a schedule from UPPAAL CORA it does not need to find a payload to execute, instead it will look for the next entry in the array describing what payloads to execute, such an array can be seen in listing 4.2. This template is responsible for updating the queue, and keeping track of how many payloads have been skipped, and the earnings, in regards to the specified profit.

The processor in this model also differs form the one in the UPPAAL CORA model in the way that in this model, the processor may remain in Running for a varying amount of time instead of just the worst execution time for the payload.

It synchronises with the scheduler template on the channel ready! and then awaits a synchronisation on either run? or skip?, while the scheduler determines what to do with the next payload.

```plaintext
int Queue[NumberOfPayloads] = {4, 0, 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 3, 0};
const int RunStart[NumberOfPayloads] = {0, 50, 100, 140, 190, 230, 280, 320, 370, 410, 460, 500, 550, 590, 640, 680};
```

Listing 4.2. Payload queue with start times, extracted from the UPPAAL CORA model.
4.2.2 Scheduler

When the location Ready from Processor is active, two transitions are available to either execute or skip the current payload, each of the transitions are dependent on the scheduler template seen in figure 4.8. This template is responsible for determining if a payload can be executed based on all of the restrictions defined, in the payload and environment description, by the user. This includes dependencies, windows, and the battery levels. It checks the windows, dependencies, and the maximum amount of executions per reset, whereas for the battery level it calculates an estimate of what it would be after executing the payload, this is done for both the available, bound and total level. This is necessary to ensure that a payload does not draw more power than currently available. All of these checks are done on the transition that broadcast \texttt{run!} indicating that it is safe to execute the payload.

If for some reason it is not possible to perform the transition it waits until we either have recharged enough energy or a chance in the windows have occurred, allowing for execution of the payload. If we are not able to perform the payload before the deadline, which is calculated by subtracting worst from the payload deadline, then the other transition is activated leading to a skip of the payload. With dependencies this can have significant consequences to the schedule because the next payload in the schedule may have depended the skipped payload thus resulting in a chain of skipped payloads.
4.2. SMC Model

4.2.3 PayloadWindow

The PayloadWindow template can be found in figure C.3, with three locations that each have an in and out going transition creating a loop between them. The overall functionality is similar to the template PayloadWindow in the UPPAAL CORA model. Like in the UPPAAL CORA model, one instance of PayloadWindow occurs for each defined window. However, in this model it is independent from the rest of the model, in the sense that there are no synchronisations, and is only responsible for keeping track of the window by updating a global variable when a change occur.

4.2.4 EnergySource

During the entire schedule we need to keep track of the battery level, as mentioned in the start of this section we are using KiBaM. In section 3.6 Battery Models we show the equations for calculating the available and bound charge as a function of time. In order to use it in our UPPAAL SMC model we need to change the equation, to not be a function of time and only give us the difference changes to the available and bound charge for each time unit. We are able to do this, because we store the value of available and bound charge as each time unit passes in two clocks, \(a\) and \(b\). In the initial state we assign \(a\) to \(\text{SoC} \times c\) and \(b = \text{SoC} \times (1 - c)\) respectively.

The equation used to calculate clock \(a\) can be seen in equation (4.4). The equation has two purposes, subtract the amount of energy being used by the currently executing payload represented with the variable \(i\), and add energy to \(a\) from \(b\) based on the difference between the two wells calculated by \(k \times \left(\frac{b}{1 - c} - a/c\right)\).

\[
a' = -i + k \times \left(\frac{b}{1 - c} - a/c\right) \tag{4.4}
\]

Similar for clock \(b\), it needs to subtract the amount of energy added to \(a\), and add the
4. Verifying a Schedule

amount of charge gained from \textit{RechargeRate}. \textit{insolation} is a boolean that is either one or zero depending on the model’s state and determines if \textit{b} gains charge.

\begin{equation}
\textit{b'} = -k*(b/(1-c)-a/c)+(\textit{insolation} * \textit{RechargeRate})
\end{equation}

An illustration can be seen below, showing how the difference is calculated along with \(k\)'s impact on how much charge that gets transferred from \(b\) to \(a\). Assume a battery has a capacity of 3000 and a ratio \(c = 0.2\) this means that the bound capacity will be 2400 and the available will be 600. If we were to calculate the imbalance like so \((b/(1-c)-a/c)\), this would result in zero because \(b/(1-c)\) would be 3000 and \(a/c\) would be 3000. Consider a payload that had just taken 100 from available in a single tick, the equation would now be \((2400/(1-0.2) - 500/0.2)\) resulting in a value of 500. The value would then be multiplied by \(k\) to give us the rate of how much to transfer from bound to available. The same is done for the bound capacity but instead of multiplying by \(k\) we multiply by minus \(k\) resulting in the value that should be subtracted from the bound charge.

With these changes to the UPPAAL SMC model we are able to run a schedule with payloads, that have windows and dependencies, with the capability of recharging the battery based on insolation. Now we only need to consider what queries should be run in order to give the most usable data for the users.

4.3 Final Output

The system will output two versions of the schedule and a result file that displays the robustness analysis. The raw schedule format is produced as a the result from UPPAAL CORA when we ask it to find the best trace for the chosen schedule duration. The other version of the schedule is one that have been transformed into a Gantt chart for better readability. The robustness analysis results is created by using UPPAAL SMC’s query feature in order to verify different properties of the schedule and model.

We will not discuss the raw version of the schedule, but a snippet of an example schedule can be found in appendix D Raw Schedule Example.

4.3.1 Gantt Chart

A Gantt chart is generated from the schedule, an example of this can be seen in figure 4.9. From the Gantt chart the user will be able to see when the individual payloads will be executed, additionally the insolation periods, and windows that specifies when some payloads are allowed to be executed, is shown in the chart. This allows the user to get a better understanding of the schedule and how the payloads are executed accordingly to their respective windows. On the right side of the chart, information about the different coloured bars are shown. They represent the payloads, insolation periods, and windows.

4.3.2 Queries

In order to verify the produced schedule and output relevant data, we have included several queries which can be run in UPPAAL SMC. We want to verify that we do not fall below the battery threshold, and the payloads that are scheduled are actually executed. Additionally,
we want to monitor different variables in order to present the user with all the relevant data we are able to extract.

All probability queries are run with a confidence factor of 95%.

**Verifying the Battery Capacity**

It is expected that battery usage in the UPPAAL SMC model differs from that in the UPPAAL CORA model as they use two different battery models, Ideal and KiBaM. The usage may also vary because of the possible differences in payload execution time. The time it takes to complete one payload in UPPAAL CORA are constant but in UPPAAL SMC it may vary, as we want to model the uncertainty, which is done by using the time range that was defined by the user in the payload description.

We expect that we will use more energy in the UPPAAL SMC model because KiBaM underestimates the energy consumption whereas the model used in UPPAAL CORA will overestimate. The UPPAAL SMC model is also guaranteed to use less or equal amounts of time to complete a payload. This is due to the UPPAAL CORA model using the worst case time when executing the payloads.

\[
Pr[\leq ScheduleLength] (<> a < 0.1)
\] (4.6)

Query 4.6 will result in a probability that reflects the risk of the battery’s available charge falling below 0.1. We added this query as depletion of the available charge can result in scheduled payloads being skipped.

\[
Pr[\leq ScheduleLength] (<> b < ((1-c)\times C)\times (ThresholdPercentage / 100))
\] (4.7)

Running query 4.7 results in the probability of the bound charge falling below the user specified threshold e.g. 40% of the maximum battery capacity. If this would occur, the nanosatellite would enter its safe mode. The query has been included as falling below the threshold is considered a fail state.

\[
Pr[\leq ScheduleLength] (<> b + a < C \times (ThresholdPercentage / 100))
\] (4.8)
Query 4.8 is the third and last query that may be used to ensure the battery capacity does not fall below the threshold. This query indicates that the sum of available and bound charge may not fall below the specified threshold.

All of these queries are relevant to verify that the battery will not go below the threshold. However, as it may take considerable time to run a probability query, as it is unpredictable how many simulations it will run, we have chosen to only include query 4.8 in the tool. This query is chosen above the others as it is ultimately the one describing the nanosatellite’s SoC.

Verifying the Payload Order

The probability for the case were at least one of the scheduled payloads are skipped, is given by running the query 4.9. A payload may be skipped because one of its restrictions are not upheld, or because there is not enough power to execute it. This query itself does not provide any information that can be used to identify what went wrong, but the user may inspect the results of the other queries in order to find the reason.

\[
Pr [\leq ScheduleLength] (\prec \text{skips} ! = 0)
\]  

(4.9)

\[
\text{simulate } 100 [\leq ScheduleLength \{\text{active, Processor.Running}\}]
\]  

(4.10)

Query 4.10 is used in conjunction with query 4.9. This is just 100 simulations which respectively represent one random trace. However in case any payloads were skipped during these simulations, it will be possible to find which was skipped at what time. What 4.10 does, is indicating what payload is supposed to execute at what time, and whether or not it was.

As these queries support each other and query 4.10 is just a 100 simulation and therefore more predictable in the time it takes to complete, both will be included in the tool.

\begin{figure}[h]
\centering
\begin{minipage}[b]{.49\textwidth}
\includegraphics[width=\textwidth]{Simulations100}
\end{minipage}\hspace{1cm}
\begin{minipage}[b]{.49\textwidth}
\includegraphics[width=\textwidth]{Simulations500}
\end{minipage}
\caption{100 simulations(left) versus 500 simulations(right) over available energy \(a\) in UPPAAL SMC}
\end{figure}

In query 4.10 it is specified that we want to run the simulation 100 times, the reason for this is that we believe 100 simulations is adequate to give a reasonable representation of the values’ ranges. In figure 4.10 we see two graphs, one where the simulation have been run 100 times and one where it have been run 500 times. The lowest observed value of \(a\) is
similar in both graphs, 182 in the one with 100 simulations and 181.9 in the one with 500 simulations. The highest value observed by the end of the simulations were 188.8 versus 189. In both cases the 500 simulations displays a wider range for the value, however the query running 100 simulations took 9 seconds, whereas the one with 500 took 55 seconds. With the small differences to the final output and the large amount of time saved we conclude that running 100 simulations will give an acceptable range in short time. Therefore we will use 100 simulations for all queries of the type simulation.

Other Information

Query 4.11 provides two time series that reflects the two wells of the battery, a and b. This is useful for the user as they might want to discard the schedule if it uses more energy than what they are comfortable with. Even if we do not go below the threshold, it may be to expensive to execute the schedule.

\[
simulate 100 [<= \text{ScheduleLength}] \{a, b\} \tag{4.11}
\]

\[
E [<= \text{ScheduleLength}; 100] (\text{max} : \text{earnings}) \tag{4.12}
\]

Query 4.12 finds the accumulated profit for the schedule. We will not judge a schedule based on this value, we will let the user decide if it is acceptable. However, the schedule is produced to maximise this value, see section 3.7 CORA Model, and should therefore be viable. The profit is an abstract variable and we will not be able tell whether or not it is satisfactory.

The three last queries, 4.10, 4.11, and 4.12, are tested with simulations. The simulations are great for giving the user insight in the behaviour of the nanosatellite when executing the schedule.

It does however not set any guarantees for the correctness or robustness of the schedule as it only displays the result of the path taken during each of the simulations.

### 4.3.3 Data From SMC

Above in section 4.3.2 Queries we discussed different queries relevant for collecting data from the schedule, here we will illustrate what this data may look like, and discuss the usefulness thereof. We will in particular focus on the queries with keyword `simulate`, as they are included for the purpose of extracting data. When the tool is executed by the user, 100 simulations of these queries will be run, but for the purpose of this section fewer will be run in order to better visualise the data.

First consider query 4.10, this will result in a graph showing what payloads was supposed to executed and when they would start and end, and whether or not they actually were executed. We have run this query through the UPPAAL GUI which generates a graph based on the result. The graph can be seen in figure 4.11. It should be noted that such graphs are not made available to the user via our tool, but all of the values needed to make such a graph is saved and made available to the user.
We can see in figure 4.11 that all payloads are executed except for the last two. This can be seen by looking at the parts where the red curve is different from $-1$, if the blue curve at any time in these periods differs from $-2$ it means that the payload was executed. The reason for the last two not being executed, is that executing them would result in depletion of the available charge, as can be seen in figure 4.12 on the left.

On the left in figure 4.12 we see the battery charges available(red), and bound(blue), over one simulation. For this example the starting capacity is 900 for available and 4500 for bound. It can be seen that the bound charge is relatively stable, meaning its capacity does not vary much, whereas the available charge gets close to depletion, resulting in the last two payloads in figure 4.11 being skipped.

In figure 4.12 on the right the available charge, a, have been simulated five times. This shows some interesting results, for some of the simulations the same skips occur as in figure 4.11 i.e. the last two, whereas some simulations result in the very last payload being executed but the two prior to it being skipped. This is a result of the feature in the UPPAAL SMC model that allows execution times of payloads to vary, and thus consume more or less power.
In this chapter we will discuss aspects of this project which could have been done differently whether it being a result of making a decision to exclude something, choosing one approach over another, or simplifications made to better the execution time of the system.

**Iterative Adjustment of the Nanosatellite Configuration**

Iterative adjustment of the nanosatellite configuration was never implemented, as we estimated that it was not possible to make a correct and non naive implementation within the time frame of this project. The downside of not having this is that the user manually has to adjust their configuration and it may be difficult to decide which parts of the input that have to be tuned, as it can be difficult to identify what changes will have a noticeable impact.

The user is also limited in what they are able to change, as it is not possible to modify all of the internal variables, and some values should never be changed. Some are hidden anyway to lower the required knowledge of the internal tools, UPPAAL CORA and UPPAAL SMC, in order to make the system easier to use.

Since the iterative adjustment of the nanosatellite configuration have become manual, we have supplied an updated version of the workflow which now handles a "No" and "Yes" equally in location 6.

**Battery Wear and Lifetime**

We did not implement any major features that would try to avoid straining the battery. But as Wognsen et al. 2015[3] expresses, it is important to predict the battery wear for battery powered systems as it is a central part of predicting the system’s lifetime. Wognsen et al. 2015[3] presented a scoring function to compare and evaluate the impact of battery usage profiles on cycle life. Their scoring system would have been simple to implement as the UPPAAL CORA model is always aware of the battery’s SoC. However, the authors conclude that the score does not indicate to what effect the usage profile will have on the battery life. They do note that it is expected that with a large enough amount of experiments, it would be possible to tailor the function to a specific battery type, such as the one the GomX-3 nanosatellite is equipped with.

It is difficult to use the scoring system without knowing what the scores represent in actual wear, as it makes it hard to determine how significant getting a high score is compared to a low one. However, as we have a large focus on battery in this project, it would have been sensible to include battery ware
Improving Insolation

As seen in figure 3.7 in section 3.7 CORA Model the initial location for the Insolation template is \texttt{inSun}. This should be changed to allow schedules to start in the other location \texttt{inEclipse}. Similar our clocks used in the template is always initially zero, this will create a problem for the user if they want to start their schedule at another time in the orbit. A solution could be to introduce a new initial committed location that has edges to \texttt{inSun} and \texttt{inEclipse}. On each of the new edges a guard will need to be placed, this should check the user defined value to indicate which starting location it should transition to. Additionally on both edges there should be an update, to set the clock to the correct time according to what the user has specified in the configuration specifications.

Zero Windows

As of now the models, both UPPAAL CORA and UPPAAL SMC, does not support there being no defined windows in the payload description as several variables are dependant on this. However, as windows is an important concept for our context we do not believe this to be a problem, as at least one window will most likely always be defined. Also there is a workaround by defining all payloads as being dependant on the window $0 - \textit{OrbitTime}$, meaning that they are not restricted by the window.

Battery Importance

During our meeting with Lars Alminde from GomSpace we discussed our approach as well as potential new approaches. He expressed that nanosatellites battery was not as important as initially assumed\cite{12}. This may be a combination of multiple factors like increased battery capacity and more efficient solar panels. Given that our meeting with
5.1 Inaccuracies and Assumptions

GomSpace was late in the project we decided to keep battery as an aspect of our problem statement.

5.1 Inaccuracies and Assumptions

During our development of the UPPAAL CORA and UPPAAL SMC models, we have made some decisions which may not always be an accurate representation of the outside world. In this section we will discuss each of these choices to evaluate the potential effect, these inaccuracies and assumptions may have on the produced schedule.

In section 3.3 Reading the Input we describe how a payload can have an associated window in which the payload can be executed, but the possibility of a payload having multiple windows was never explored, despite of our models supporting it. Unfortunately our payload descriptions semantics does not allow for multiple windows. This has the potential to make the produced schedule differ significantly from the actually optimal schedule, and we therefore believe this to be a significant inaccuracy.

Introducing the variable idle_cost to our configuration file, see section 3.3 Reading the Input, could reduce the computation time for producing a schedule in UPPAAL CORA. It is meant to reduce the number of payloads, by removing operational payloads that run frequently and adds no profit to the schedule and simulate their energy requirements via the idle_cost variable. This could give misleading results since idle_cost represent the average energy consumption for the operational payloads and continuously drain on the battery unlike payloads that only drains on the battery for short periods. However, by specifying the idle_cost to be zero and describe all payloads in the payload description this variable will not cause any misrepresentation to the produced schedule.

In section 3.7 CORA Model it was described that the user defined profit is used to calculate the cost in the UPPAAL CORA model. As UPPAAL CORA finds the best schedule in regards to the cost, profit works as a priority. Because of this it might be desirable for the user to define a wider range of priorities, especially in cases with many different payloads. However, we believe this to have a small effect on the produced schedule.

Given the restrictions of UPPAAL CORA, we modelled an Ideal battery model. This will according to the observations made during section 3.6 Battery Models overestimate the battery compared to a real battery, which may cause the produced schedules to consume more energy than what is available. The impact of this is mitigated with our UPPAAL SMC model having a more pessimistic battery model.

The Processor template in the UPPAAL CORA model, see section 3.7 CORA Model, we mention two factors that can have an effect on the optimal schedule. A payload is always executed based on its worst execution time, and deadlines force the model to wait a period after the payload has been executed. Payloads running worst case have the benefit of taking more energy, and since the Ideal battery model overestimate the actual value by doing so is believed to get closer to the actual. Unfortunately schedules produced this way is not as payload efficient as they could be, especially because of the delay to wait for the deadline. However, maximising the payload efficiency i.e. discarding the wait after executing a payload, would leave the schedule more vulnerable to errors potentially...
resulting in a chain reaction of payloads being skipped based on the individual payloads dependencies and windows when the schedule is verified via UPPAAL SMC. We believe this to have a considerable impact on the schedule, but where exist a workaround for this by simply defining the deadline to equal the maximum execution time.

Additionally the way dependencies are modelled, does not allow for all forms of payload dependencies such as being dependant on "A" or "B", which can lead to less optimal schedules if the user is not able model their payload dependencies within our system. This can have a significant impact on the produces schedule. However, there is a form of workaround for this e.g. for the GomX-3 case, after having run one L-band it was needed to run two X-band. In our system the user can express the X-band as having a maximum runs of four and being dependant on one L1 and one L2 which each have a maximum runs of 1. This would result in a schedule starting with one of each L-band followed by four X-band. Because of this we believe this inaccuracy to be of minor importance.

In section 3.7 CORA Model when describing insolation we made two inaccuracies for the sake of performance. Dividing the number of updates made to the battery during an orbit, and insolation period lasting precisely half the orbit length. Determining the severity of reducing the number of updates to the battery during an orbit, showed a big decrease in time taking to produce the schedule with the effect of an added inaccuracy which we concluded to be negligible.

Assuming that orbits are constant is not realistic, however the only knowledge we have about the effect of the irregularity of an orbits is, that the inaccuracy increases over time. We therefore do not know the effect this will have on the schedule.

Modifications to the UPPAAL SMC model were also done, as described in section 4.2 SMC Model, KiBaM is one of the factors that impact the UPPAAL SMC model, unlike in UPPAAL CORA where the battery model overestimated the battery, KiBaM underestimate resulting in few schedules being discarded even though the may be viable. We believe this to be a rare occurrence and will have little effect.

Our UPPAAL CORA model does not support knowledge from previous schedules, so the variable runs which keep track of which payloads has been executed will always be zero initially, making it impossible to produce a schedule based on runs from the previous schedule. This have a potential to greatly impact the schedule but also to be of no effect.

To conclude, our system implements several inaccuracies most with little to no effect but others with a more significant impact. We believe the payloads with multiple windows and the missing start values for runs to be significant and something that would require correction before actually using the system. The remaining we believe to have a small enough impact, either because of being negligible of because of the possible workarounds to continue using, with the exception of the perfect orbit as we simply do not know its impact.
In this chapter we conclude on the report and project as a whole to evaluate the degree of which the problems presented during chapter 2 Problem Analysis have been fulfilled. This culminated in the following problem statement:

“How is it possible to produce a schedule that consider multiple requirements, and how is it possible to verify the robustness of such a schedule”

To conclude on the problem statement and to what degree the different requirements have been fulfilled, we must examine and evaluate each of the requirements, and the robustness.

As described in section 3.7 CORA Model the SoC threshold is ensured by the UPPAAL CORA model, as exceeding this would result in a deadlock, causing the trace to be discarded. Furthermore, in the UPPAAL SMC model we precalculate the energy consumption to also enforce a payload can only be executed if it will not cause the SoC to fall below the threshold. Which these percussions installed, SoC will always be kept above this threshold.

After having completed a payload our UPPAAL CORA model will immediately start execution of the next payload, if any payload is available. However, we do not currently execute payloads as soon as they are available if the model is idling, it does not check if a payload is executable until a change in the PayloadWindow template occurs. Therefore the requirement for payload efficiency is only partially fulfilled.

If a payload is defined as only being able to execute in a window, it must never execute at any other time. This is ensured in both models, and the requirement for windows is therefore fulfilled.

As the schedule is produced in UPPAAL CORA with the setting to find the best trace based on the defined profit, we have archived the goal of maximising profit.

Ways of preventing excessive wear on the battery is only considered in our UPPAAL SMC model by enforcing that when the battery reaches maximum capacity, it will stop recharging and only start again when a percentage of the capacity has been drained. Given that more could have been done to capture the potential effect of battery wear this is considered partial done.

As discussed in section 5.1 Inaccuracies and Assumptions there are several assumptions...
and inaccuracies associated with producing our schedule. Despite of us considering several of these to have little to no impact on the schedule, others will most definitely impact it. As this is the case, it will cause a different schedule than what is the actual optimal one.

In conclusion we are able to construct a schedule with the given parameters using UPPAAL CORA, and partially check its robustness through the UPPAAL SMC model, which give some indication for the confidence in the produced schedule. This achieve our goal in the main statement. However, evaluating some of the requirements to being partially done we will state that we partially achieve our objectives in generating an optimal schedule and doing some robustness checking on it.
This chapter covers the ideas we believe is worth pursuing in the future, or was not pursued in this project due to the limited project period.

**Memory Considerations**

This topic concerns the physical memory on the nanosatellite in respect to payloads that is producing some amount of data during execution. During our meeting with GomSpace, we were introduced to the problem of sending data cost efficiently, as the nanosatellite have multiple windows where it can send data back to Earth. But depending on the geographical location, it would cost a fee to use other countries satellite dishes. It would be interesting to produce schedules that would take this problem into consideration while also balancing power and payload utilisation. This would make it so that some windows for the data sending payloads are more attractive than others because of their lower fee[12].

**Bandwidth** Had we implemented the aspect of physical memory, we could also have considered bandwidth. This could be used to determine how long it would take send and receive data to and from Earth. Or instead of having a dependency between sending and collecting data, it would be possible to collect a curtain amount of data till the storage is full and to then send a sub part of the collected data if there is not enough time to send it all.

**Start Orbit Time**

When the user is specifying the configuration, it is necessary to specify the start SoC. Something similar for this would be useful for specifying where in the orbit the nanosatellite will start. Currently, we will always start at orbit time zero but this might not reflect the nanosatellites actual position in the orbit.

A possible solution would be to allow the user to chose a number within the orbit length, which will be the starting time for the nanosatellite. Alternatively, this number could be an offset for the insolation/eclipse periods and the windows, such the that the start orbit time will still be at zero.

**Celestial Bodies Obstructing Line of Sight to the Sun**

The most common celestial body to obstruct the nanosatellite from recharging is the moon with the exception of Earth. It is questionable how much this would affect the generation of a schedule. If it were to have a real thread, the moon and nanosatellite would need to have close to similar orbital rotation, which is unlikely. We did not believe that this
problem could effect schedule’s quality, but it made us consider other similar problems. We have made the assumption that the nanosatellite will spent an equal amount of time in insolation and eclipse. This will not always reflect the actual time distribution. We believe that this can be solved by introducing a constant in the configuration. The constant is a percentage that describes the time spent in insolation or eclipse.

**Oval Orbits**

Our models assumes that an orbit is circular so it take equal time the travel a half orbit from any starting point in the orbit. This was not include to simplify the model and deemed unnecessary because the produced schedules only spanning between 12 to 24 hours.

**Payload Dependencies**

Currently a payload can be dependant on the successful execution of another payload. We have introduced dependencies in order to make it possible for the user to help decide the order of the payloads, such that it is only possible to send data after it have been collected. This feature can be improved upon as there is not always a one-to-one relation between the number of times a payload should be executed. In the GomX-3 experience paper[1], they observed that one L-band payload would require two X-band payloads in order to send all of the data that had been collected. Our models does not directly allow for such a relation as the completion of the dependency unlocks the execution of the dependant payload until a function locks it again. Currently, the locking function is only called when all payloads are no longer allowed to be executed because they have hit their maximum of allowed executions. This is what we are referring to when we state that it is possible to indirectly make this relationship, as the maximum allowed executions can be set to the same amount as the dependant payload requires it to execute. By this approch it can be defined that $L$ may be run once, and $X$ is dependant on $L$ and may be run twice. This however is more of a workaround and we believe that an actually implementation of these dependencies would increase the expressiveness of the payload descriptions.

**Satellite’s Attitude & Drag**

The attitude of the nanosatellite is not modelled, which results in the schedule not being as payload efficient as it could have been if we did consider the attitude. Bisgaard et al. 2016[1] declared that the X and L-band equipment is installed on opposite surfaces of the GomX-3 nanosatellite, which mean that only of them may be used at the time, as they have to be pointed towards Earth. They accommodated this by implementing logic in their model for slewing the nanosatellite into place before the payload was executed. Our approach is to advice the user into including the possible time it takes to slew the nanosatellite before a payload is executed. This has the effect that the worst case execution time of the payloads that are attitude dependent, is increased by that of time it takes to slew into place. Another effect is that the tool may produce a schedule were the nanosatellite has to slew back and forth multiple times in a row, instead of bundling payloads that needs the same orientation. Support for setting the nanosatellite’s attitude, and specifying the dependence of it on payloads would solve this inefficiency. The positive effect of not considering this aspect
is the reduced state space in the UPPAAL CORA model. However, We do believe that it is worth testing how a schedule may be produced differently, if the feature were to be implemented.

**Battery Wear**

Battery deteriorating is not supported in our solution, but since the user can theoretically calculate the actual capacity of the battery and set the maximum capacity to that value, it is possible to pass the information to the model. Furthermore, the scheduler is not trying to minimise battery wear when producing a schedule. However, a location have been added to the UPPAAL SMC model in order to avoid overcharging the battery, as explained in section 4.2 SMC Model.

Woglsen et al. 2015[3] expresses how it is important to find a balance between not wearing the battery and doing something worthwhile with the nanosatellite i.e. executing payloads. We believe that since our tool has an emphasis on battery usage, the ability to model battery wear and avoid schedules that have a high impact on it, will be sensible to implement.

**Restarts**

In section 2.3 Improvements and Additions we described a desire for incorporating unpredictable restarts into the schedule, in order to test the schedules robustness. However, this was never implemented. Implementing this could have been done in several ways e.g. restarts could have been added to the schedule at random when generating the UPPAAL SMC model, or a septate template could have been added to the UPPAAL SMC model, which would cause the processor and scheduler templates to leave their current location for a duration of time.

**Starting Values**

In section 5.1 Inaccuracies and Assumptions we discussed that, despite of the many input parameters fed to the system, we believe it could have been beneficial to include more in order to better represent the nanosatellites current state. Specificity we believe a representation of current amount of executions per payload should have been included. This is important as the last produced schedule may have ended with several payloads having been executed, potentially fulfilling some dependencies, meaning other payloads should be executable from the beginning of the new schedule.

**Multiple Windows per Payload**

As described in section 5.1 Inaccuracies and Assumptions, there are multiple stations on Earth with which the nanosatellite can communicate. It would be a beneficial addition to allow a payload to be associated with multiple windows. As this is not currently included it may have a significant impact on the produced schedule, since the nanosatellite may be ready to transmit data to Earth at time 20 but is only defined to have a window at time 70-80 despite of it, in reality, also having one at time 30-40. This could be included by making an update to the script feeding the payload description to the models.
Glossary

DOD  Depth Of Discharge. 5

KiBaM  Kinetic Battery Model. 15–17, 23, 27, 29, 31, 38, 46

SoC  State of Charge. 3, 5, 6, 11, 14, 17, 19, 20, 29, 32, 35, 39, 45

UPPAAL CORA  UPPAAL Cost Optimal Reachability Analysis. iii, v, 1, 3, 5, 7, 8, 12–14, 17, 18, 23, 27, 29–31, 35–40, 43, 45–47

UPPAAL SMC  UPPAAL Statistical Model Checking. iii, v, 1, 7, 8, 11, 12, 18, 23, 24, 26, 27, 29–32, 34–40, 43, 46

List of Figures

3.1  Flowchart that displays the workflow ................................. 8
3.2  Example template .......................................................... 13
3.3  Example template for UPPAAL CORA ............................. 13
3.4  The two wells available(a) and bound(b) ........................... 16
3.5  Processor template ....................................................... 19
3.6  Scheduler template ...................................................... 19
3.7  Insolation template ....................................................... 20
3.8  Graphs illustrating SoC, with different granularity. Bottom graph is a zoom in of the top graph. ................................. 20

4.1  Model for showcasing probability .................................... 23
4.2  Simple model .............................................................. 24
4.3  One simulation that captures the value of two variables \( x, y \) over 100 units of time ..................................................... 25
4.4  Five simulations that captures the value of two variables \( x, y \) over 100 units of time ..................................................... 25
4.5  Frequency histogram generated from query 4.3 .................. 26
List of Tables

3.1 Comparison of actual measurements and predictions from the Ideal model. The specification for the variable loads can be found in table A.1. 15
3.2 Comparison of actual measurements and predictions from the Peukert model. The specification for the variable loads can be found in table A.1. 16
3.3 Comparison of actual measurements and predictions from the KiBaM. Specification for the variable loads can be found in table A.1 17
3.4 Execution times when generating a 12 hour schedule, using different granularity for insolation 21
A.1 Data taken from Jongerden and Haverkort 2009[9] 51
3.1 Example of how five payloads can be defined .......................... 10
3.2 Example of how the environment can be defined ....................... 11
3.3 Function calcCost(), used for calculating the cost rate ............... 18
4.1 Declarations ................................................................. 24
4.2 Payload queue with start times, extracted from the UPPAAL CORA model 27
B.1 Global declarations .......................................................... 53
B.2 Declarations for Processor template ....................................... 54
B.3 Declarations for Processor template ....................................... 55
B.4 Declarations for Processor template ....................................... 56
B.5 Declarations for Scheduler template ....................................... 56
B.6 Declarations for Insolation template ....................................... 57
B.7 Declarations for PayloadWindow template ................................ 58
B.8 System Declarations .......................................................... 59
C.1 Global declarations .......................................................... 61
C.2 Global declarations .......................................................... 62
C.3 Declarations for Processor template ....................................... 63
C.4 Declarations for Insolation template ....................................... 63
C.5 Declarations for EnergySource template ................................... 64
C.6 Declarations for Scheduler template ....................................... 65
C.7 Declarations for PayloadWindow template ................................ 66
C.8 System declarations .......................................................... 66
D.1 The raw schedule format. Note the first transition at line 5 that indicate we enter a running state i.e. we start executing a payload. We stop executing the payload after 15 time units(minutes) have passed. ..................... 67
Bibliography


## Variable Loads

Table A.1. Data taken from Jongerden and Haverkort 2009[9]

<table>
<thead>
<tr>
<th>Cases</th>
<th>Timing, min</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>(0, 19.5, 26.0)</td>
</tr>
<tr>
<td>#2</td>
<td>(0, 31.0, 41.3)</td>
</tr>
<tr>
<td>#3</td>
<td>(0, 41.0, 54.6)</td>
</tr>
<tr>
<td>#4</td>
<td>(0, 74.6, 99.5)</td>
</tr>
<tr>
<td>#5</td>
<td>(0, 105.7, 140.9)</td>
</tr>
<tr>
<td>#6</td>
<td>(0, 19.5, 29.9)</td>
</tr>
<tr>
<td>#7</td>
<td>(0, 19.5, 22.1)</td>
</tr>
<tr>
<td>#8</td>
<td>(0, 23.4, 29.9)</td>
</tr>
<tr>
<td>#9</td>
<td>(0, 15.6, 22.1)</td>
</tr>
<tr>
<td>#10</td>
<td>(0, 0.5, 5.5, 10.5, 35.5, 60.5, 85.5, 110.5)</td>
</tr>
</tbody>
</table>
B.1 Global Declarations

```c
bool on = true; // in coulombs, status
const int N = 5;
const int ScheduleLength = 720;
const int Windows = 2;
const int Window[Windows][2] = {{10, 30}, {50, 80}};
const bool RunInWindow[Windows][N] = {{0, 1, 1, 1, 0}, {1, 0, 0, 0, 0}};
const int OrbitTime = 90;
const int BatteryMax = 5400;
const int BatteryCritical = 40;
typedef int [0, Windows - 1] id_t;
clock ins;
clock t_time;
broadcast chan ready, run, win;
urgent broadcast chan empty;
const int MaxRuns[N] = {9, 3, 2, 2, 1};
const int ChargeRate = 1;
const int IdleCost = 1;
int batteryCap = 4800;
int active = 0;
int runs[N] = {0, 0, 0, 0, 0};
int tRuns[N] = {0, 0, 0, 0, 0};
bool available[N] = {0, 0, 0, 0, 0};
bool runnable[N] = {0, 0, 0, 0, 0};
const int Depend[N][N] =
    {{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 2, 2, 0, 0}, {0, 0, 0, 0, 0}};
const int TaskTimes[N][3] =
    {{10, 15, 20}, {10, 15, 20}, {5, 10, 15}, {5, 10, 20}, {5, 15, 20}};
const int Profit[N] = {2, 4, 3, 1, 1};
const int Load[N] = {4, 6, 4, 6, 6};
```

Listing B.1. Global declarations
B.2 Processor

```c
int queue[N] = {-1,-1,-1,-1,-1};
clock x;
int runCost = 0;
void mayRun() {
    int i, j, ok;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if(runs[j] >= Depend[i][j]) {
                ok++;
            }
        }
        if(available[i] == 1 && runs[i] < MaxRuns[i]) {
            runnable[i] = true;
        } else {
            runnable[i] = false;
        }
    }
}
void setQueue() {
    int i, j;
    bool tempqueue[N] = {-1,-1,-1,-1,-1};
    for (i = 0; i < N; i++) {
        if (runnable[i] == true) {
            queue[j] = i;
            tempqueue[i] = true;
            j++;
        }
    }
    for (i = 0; i < N; i++) {
        queue[i] = tempqueue[i];
    }
}
void dequeue() {
    int a, i;
    a = queue[0];
    for (i = 0; i < N-1; i++) {
        queue[i] = queue[i+1];
    }
    queue[N-1] = a;
}
```

Listing B.2. Declarations for Processor template
void reset() {
    int i = 0;
    int all_done = 0;
    int j = 0;
    for (i = 0; i < N; i++) {
        if (runs[i] == MaxRuns[i]) {
            all_done ++;
        }
    }
    if (all_done == N) {
        for (j = 0; j < N; j++) {
            runnable[j] = 0;
            runs[j] = 0;
        }
    }
}

int runnableCount() {
    int count = 0;
    int i = 0;
    for (i = 0; i < N; i++) {
        if (runnable[i] != 0) {
            return 1;
        }
    }
    return 0;
}

bool checkBattery() {
    if (batteryCap - Load[active] * ((TaskTimes[active][0] + TaskTimes[active][1])/2) < BatteryMax * (BatteryCritical /100)) {
        return 1;
    } else {
        return 0;
    }
}

void updateBattery() {
    int usage = Load[active] * ((TaskTimes[active][0] + TaskTimes[active][1])/2);
    batteryCap -= usage;
    runs[active] ++;
    tRuns[active] ++;
}

void subIdle() {
    int idle_cost = IdleCost * (TaskTimes[active][1] - ((TaskTimes[active][0] + TaskTimes[active][1])/2));
    batteryCap -= idle_cost;
}

Listing B.3. Declarations for Processor template
void calcCost() {
    int i = 0;
    int j = 0;
    int Window_dep = 0;
    int pen = 0;
    for (i = 0; i < Windows; i++) {
        for (j = 0; j < N; j++) {
            if (RunInWindow[i][j] == 1) {
                Window_dep ++;
            }
        }
        if (Window_dep > 0) {
            for (i = 0; i < Windows; i++) {
                for (j = 0; j < N; j++) {
                    if (RunInWindow[i][j] == 1 && runnable[j] == 1 && active != j) {
                        pen ++;
                    }
                }
            } else {
                runCost = 5 - Profit[active];
                return;
            }
            for (i = 0; i < Windows; i++) {
                if (pen && RunInWindow[i][active] == 0) {
                    runCost = (5 - Profit[active]) * 2;
                    return;
                }
            }
        } else {runCost = 5 - Profit[active];}
    }
}

Listing B.4. Declarations for Processor template

B.3 Scheduler

bool threshold = true;
bool checkBattery() {
    if (batteryCap >= (BatteryMax / 100) * BatteryCritical) {
        return true;
    } else {return false;}
}

Listing B.5. Declarations for Scheduler template
B.4 Insolation

```
1 clock splitTime;
2 int chargeCount = 0;
3 void increaseBattery()
4     chargeCount ++;
5     if(BatteryMax &lt;= batteryCap + ((ChargeRate * OrbitTime)-(IdleCost * OrbitTime))/8){
6         batteryCap = BatteryMax;
7     }
8     else{ batteryCap += ((ChargeRate * OrbitTime)-(IdleCost * OrbitTime))/8;}
9     splitTime = 0;
10 }
11 void subIdle()
12     batteryCap -= (IdleCost*(OrbitTime/8)); splitTime = 0; chargeCount ++;
```

Listing B.6. Declarations for Insolation template
B.5 PayloadWindow

```c
clock wtime;
void alwaysAvailable () {
  int i = 0; int count = 0;
  for(i=0; i &lt; N; i++) {
    if(available[i] == 1) { count++;
  }
  if(count == 0) {
    for(i = 0; i &lt; N; i++) {
      if(RunInWindow[id][i] == 0) {
        available[i] = 1;
      }
    }
  } else {
    for(i = 0; i &lt; N; i++) {
      if(RunInWindow[id][i] == 0 &amp;&amp; available[i] == 1) {
        available[i] = 1;
      } else { available[i] = 0;
      }
    }
  }
}
void setRunnable () {
  int i = 0;
  for (i = 0; i &lt; N; i++) {
    if(RunInWindow[id][i] == 1) {
      available[i] = 1;
    }
  }
}
void removeRunnable () {
  int i = 0;
  for (i = 0; i &lt; N; i++) {
    if(RunInWindow[id][i] == 1) {
      available[i] = 0;
    }
  }
}
```

*Listing B.7. Declarations for PayloadWindow template*
B.6  System Declarations

Listing B.8. System Declarations

system Processor, Scheduler, Insolation, PayloadWindow;

\textbf{Figure B.1.} PayloadWindow template
C.1 Global Declarations

```c
bool on = true;
const int ScheduleLength = 720;
const int N = 5;
// KiBaM
const double C = 5400.0;
const double soc = 4800.0;
const double c = 1.0/6, k = 2.324e-4, k2 = k/c/(1-c);
const double ThresholdPercentage = 40.0/100.0;
const double Threshold = ((1-c)*C)*(ThresholdPercentage);
const double MaxB = (1-c)*C;

clock a = c*soc;
clock b = (1-c)*soc;

clock t; // time since release, accumulated execution time
broadcast chan ready, skip;
urgent broadcast chan run, capacityFull;
int skips = 0;

// Time to pass one orbit
const int OrbitTime = 90;

// Charge
bool insolation = true;
const double RechargeRate = 0.2;

// Discharge
double i = 1.0;
const double IIdle = 1.0;
const int NumberofPayloads = 16;

// Defined when a payload is active
int B = 0; // best
int W = 0; // worst
int D = 0; // deadline
```

Listing C.1. Global declarations
// Restriction related
const int Windows = 2;
const int Window[Windows][2] = {{10, 30}, {50, 80}};
const bool RunInWindow[Windows][2] = {{0, 1, 1, 0}, {1, 0, 0, 0}};
typedef int[0, Windows - 1] id_t;
int inWindow[Windows] = {0, 0};
const int Depend[2][N] =
    {{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}};
const int MaxRuns[N] = {9, 3, 2, 2, 1};
int runs[N] = {0, 0, 0, 0};
bool runnable[N] = {0, 0, 0, 0, 0};
int totalRuns[N] = {0, 0, 0, 0, 0};
const int Profit[N] = {2, 4, 3, 1, 1};
int earnings = 0;
int Costs[N] = {4, 6, 4, 6, 6};

// Tasks
int active = -1;
clock totalTime;
int payloadNumber = 0;
//from cora
int Queue[NumberOfPayloads] = {4, 0, 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 3, 0};
const int RunStart[NumberOfPayloads] = {0, 50, 100, 140, 190, 230, 280, 320, 370, 410, 460, 500, 550, 590, 640, 680};

Listing C.2. Global declarations
### C.2 Processor

```plaintext
clock x;

void enqueue(){
  if (payloadNumber < NumberOfPayloads) {
    i = Idle + Costs[active];
    B = Payloads[active][0];
    W = Payloads[active][1];
  }
  else { on = false; }
}

void deadline(){
  if (payloadNumber < NumberOfPayloads) {
    D = Payloads[active][2];
  }
  else { on = false; }
}

void dequeue(){
  i = Idle;
  if (payloadNumber >= NumberOfPayloads) {
    on = false;
  }
}

void setActive() { active = Queue[payloadNumber]; }

bool done() { return payloadNumber >= NumberOfPayloads; }

void skipped() { skips ++; }
```

**Listing C.3.** Declarations for Processor template

### C.3 Insolation

```plaintext
clock charge;
```

**Listing C.4.** Declarations for Insolation template
charge <= OrbitTime/2
insolation = false

charge <= OrbitTime
insolation = true,
charge :=0

charge >= OrbitTime/2

Figure C.1. Insolation template

C.4 EnergySource

\[ \text{double } c_C = ((1-c)\cdot C) - 10.0; \]

Listing C.5. Declarations for EnergySource template

\[ a' = -i + k \cdot \frac{b}{(1-c)} - a/c \]
\[ b' = -k \cdot \frac{b}{(1-c)} - a/c + (\text{insolation} \cdot \text{RechargeRate}) \]

on && b <= Threshold
\[ \text{on = false} \]

b > MaxB
\[ \text{capacityFull!} \]

on && b <= Threshold
\[ \text{on = false} \]

a' = -i + k \cdot \frac{b}{(1-c)} - a/c
\[ \&\& \quad \text{b' = -k \cdot \frac{b}{(1-c)} - a/c} \]

b < c_C
\[ \text{capacityFull!} \]

Figure C.2. EnergySource template
C.5 Scheduler

```c
double checkB(double a, double b, double t) {
    int j = 0;
    double temp_b = b, temp_a = a;
    for (j = 0; j < W; j++) {
        temp_a = temp_a - on*i + k*(temp_b/(1-c) - temp_a/c);
        temp_b = temp_b - k*(temp_b/(1-c) - temp_a/c);
    }
    temp_b = temp_b + (D-W)*IIdle;
    if (payloadNumber < NumberOfPayloads - 1) {
        temp_b = temp_b + (RunStart[payloadNumber+1] - t) * IIdle;
    }
    if (temp_b + temp_a < C*(ThresholdPercentage/100)) { return 0; }
    return temp_b;
}

bool mayRun() {
    // Are there a Window constraint?
    int i, j, ok; int count = 0; int inWin;
    for (i = 0; i < Windows; i++) {
        if (RunInWindow[i][active] == 1 && inWindow[i] == 1) {
            inWin = 1;
        } else if (RunInWindow[i][active] == 1) {
            count ++;
        }
    }
    if (count == Windows) { inWin = 1; }
    // Are there a dependency constraint? or have the payload been run to much?
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (runs[j] >= Depend[active][j]) {
                ok ++;
            }
        }
        if (ok == N && inWin == 1 && runs[i] < MaxRuns[i]) {
            return true;
        } else { return false; }
    }
    return true;
}
```

Listing C.6. Declarations for Scheduler template
C.6 PayloadWindow

```c
clock wtime;

void setRunnable()
{
    if (inWindow[id] == 0) { inWindow[id] = 1; }
    else { inWindow[id] = 0; }
}
```

Listing C.7. Declarations for PayloadWindow template

![Diagram of PayloadWindow templates]

C.7 System Declarations

```c
system Processor, EnergySource, Scheduler, Insolation, PayloadWindow;
```

Listing C.8. System declarations
Listing D.1. The raw schedule format. Note the first transition at line 5 that indicate we enter a running state i.e. we start executing a payload. We stop executing the payload after 15 time units (minutes) have passed.