Designing a Tool-Chain For Generating Battery-Aware Contact Plans Using UPPAAL

Exploring how UPPAAL can be used for scheduling the Ulloriaq case

Master Thesis

August Kørvell & Kristoffer Degn – sv104f19

Aalborg University
Software

Summary

Constellations of micro LEO satellites are increasing in popularity for data transfer and communication between far-away points on Earth. An example of this is the Ulloriaq mission, proposed by GOMspace. The goal of this mission is to collect observations and remote sensing data from an area above Greenland and deliver it to a ground station in Aalborg. The proposed constellation has some constraints in the form of limited battery resources and the fact that a persistent end-to-end connection cannot be established between the source and the destination, which makes it difficult to schedule. We propose solving this by modelling the constellation as a Delay-Tolerant network using Timed Automata and UPPAAL.

To achieve this, we have designed and implemented a tool-chain for generating and validating contact plans (CP) for the mission. The tool-chain consists of four steps. The constellation has first been modelled as a network of Priced Timed Automata (PTA) and implemented using a store-carry-forward messaging scheme, specifically as a Delay-tolerant network. Called the *Ulloriaq Scheduling Model*. The satellite batteries are modelled in the PTA as a linear battery, due to its discrete behavior. UPPAAL CORA is then used to find a cost-optimal trace for the model, which reaches a predefined goal state. A downside of this is that the linear battery model used is naive compared to a real-life scenario. The generated trace is then parsed to a custom tool called Ulloriaq Contact Plan Extractor. This tool parses the trace from CORA and outputs a contact plan. To make up for the limitations of the PTA, a Stochastic Timed Automata (STA) has been used, which models the Kinetic Battery model with added uncertainties on the battery loads and the initial state of the satellite batteries. This model is called the *Ulloriaq Post-Processing Model*. The contact plan is then fed to the *Ulloriaq Post-Processing Model*, which calculates the probability of any of the satellite's batteries dying using, UPPAAL SMC. The final result is a verified contact plan.

Over all, we were successful in modelling the constellation proposed for the Ulloriaq case using UPPAAL CORA and UPPAAL SMC. The tool-chain designed for the purpose can take raw data concerning the involved satellites orbits and generate verified contact plans. The process is however not completely automated yet, and still need some manual work. We ran a number of experiments on the two models, testing the different heuristics implemented for the *Ulloriaq Scheduling Model* as well as the effect of changing different parameters about the constellation and mission as a whole.

The results from our many experiments were successful and helped to increase our confidence in the model, especially when comparing to previous work done for the same mission, where the results were made by modelling the constellation as a Mixed Integer Linear Programming problem and using a solver and custom plug-in to generate the contact plans.

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
9220 Aalborg Øst
http://www.aau.dk

Title:
Designing a Tool-Chain For Generating Battery-Aware Contact Plans Using UPPAAL

Theme:
Model checking

Project Period:
Spring 2019

Project Group:
sv104f19

Participant(s):
August Malling Kørvell
Kristoffer Mathiasen Degn

Supervisor(s):
René Rydhof Hansen
Kim Guldstrand Larsen
Isabella Kaufmann

Copies: 0

Page Numbers: 101

Date of Completion:
14/06/2019

Abstract:

Constellations of micro LEO satellites are increasing in popularity for data transfer between far-away points on Earth. An example of this is the Ulloriaq mission. The goal of which it is to collect data above Greenland and deliver it, through a network of CubeSats to an Aalborg. The proposed constellation has constraints in the form of limited battery resources and the fact that a persistent end-to-end connection cannot be established between the source and the destination, which makes it difficult to schedule. We propose solving this by modelling the constellation as a DTN using TAs and UPPAAL. We were successful in implementing a tool-chain for generating and validating contact plans (CP), using UPPAAL CORA to find a trace through a system of PTAs, modelling a network of satellites, optimized for battery usage and fast data delivery. A CP was then extracted from this trace using a custom made tool. Lastly, the CP was given as input to an STA, which verified it using KiBaM with added uncertainty to the loads and initial battery charge. The resulting CPs were compared to the results from a similar attempt at modelling the mission, using MILP. The conclusion from the comparison was that even though the CPs were not identical, their overall behavior were very similar.

Contents

1 **Introduction** **3**

2 **Preliminaries** **5**
- 2.1 The Ulloriaq case 5
- 2.2 Communication Protocols 7
  - 2.2.1 Delay-Tolerant Networking 8
  - 2.2.2 Contact Plans 9
- 2.3 UPPAAL 10
  - 2.3.1 Verification 14
- 2.4 UPPAAL CORA 14
  - 2.4.1 Optimal Scheduling 16
- 2.5 UPPAAL SMC 16
  - 2.5.1 Querying 17
- 2.6 Battery Models 18
  - 2.6.1 Linear Battery Model (LiBaM) 19
  - 2.6.2 Kinetic Battery Model (KiBaM) 19

3 **Ulloriaq as a Mixed-Integer Linear Programming Problem** **23**
- 3.1 Preliminaries for MILP 23
  - 3.1.1 Linear Programming 23
  - 3.1.2 Integer Programming 24
  - 3.1.3 Mixed Integer Linear Programming 25
- 3.2 Ulloriaq Case MILP Model 25
  - 3.2.1 Generation of Contact Plans from MILP 29

4 **Models and Toolchain** **31**
- 4.1 *Ulloriaq Scheduling Model* 32
  - 4.1.1 Model Architecture 32
  - 4.1.2 High-level Model Behaviour 33
  - 4.1.3 Heuristics 37
  - 4.1.4 Model description 38
  - 4.1.5 Cost 50
  - 4.1.6 Time granularity 50
- 4.2 Contact Plan Extractor 51

4.2.1 UTAP 51
4.2.2 Ulloriaq Contact Plan Extractor 51
4.3 *Ulloriaq Post-Processing Model* 54
4.3.1 Uncertainty 57
4.3.2 Query 58

5 Experiments **61**
5.1 Experiment environment 61
5.1.1 Settings 62
5.1.2 Queries 63
5.1.3 Running an experiment 64
5.1.4 Understanding the results 64
5.2 Experiment 1: Number of satellites 66
5.3 Experiment 2: Maximum data collection 68
5.4 Experiment 3: Simulation length 71
5.5 Experiment 4: Impact of heuristics 73
5.6 Experiment 5: Cost 78
5.7 Comparison with MILP 82

6 Discussion **85**
6.1 Sources of error 85
6.2 Future work 86
6.3 UPPAAL CORA 87

7 Conclusion **89**
7.1 Results 89

Bibliography **93**

A Contact Plans **95**
A.0.1 Number of satellites 95
A.0.2 Simulation Length 97
A.0.3 Heuristics 98

Preface

Aalborg University, 14/06/2019

This report, and the work described with in, has been done by August Kørvell and Kristoffer Degn, between February 2019 and June 2019. The authors would like to thank our supervisors René Rydhof Hansen, Kim Guldstrand Larsen and Isabella Kauffmann. Furthermore, we would like to thank proff. Holger Hermanns of Saarland University and his group of PhD students, for their cooperation in the development of our models.

Aalborg University, 14/06/2019

Kristoffer Mathiasen Degn
kdegn14@student.aau.dk

August Malling Kørvell
akarve14@student.aau.dk

Reading Guide

Some of the content in this report are inspired by a previous report [13], written by the same authors, these are:

- Chapter 1: Introduction
- Section 2.1: The Ulloriaq case
- Section 2.2: Communication Protocol
- Section 2.3: UPPAAL
- Section 2.4: UPPAAL CORA
- Section 2.5: UPPAAL SMC
- Section 2.6: Battery Models
- Chapter 3: Ulloriaq as a Mixed-Integer Linear Programming Problem

1 | Introduction

Large scale networks of satellites in Low-Earth orbit (LEO) are becoming popular as a means of providing cost-efficient data transfer or communication. An example of this is SpaceX and their Starlink Mission [21]. The goal of the Starlink mission is to provide broadband internet to parts of the globe where this has been difficult to achieve with conventional methods. On May 15th, 2019, SpaceX was successful in launching 60 satellites and are planning to expand the network to involve thousands of satellites.

Similar to Starlink Mission, GOMspace [9] has proposed a potential mission called Ulloriaq, which is Greenlandic for 'star'. The goal of this mission is to collect observations and remote sensing data from an area above Greenland and deliver it to a ground station in Aalborg [12]. GOMspace currently has two CubeSats, GomX-4A and 4B, in orbit [16], commissioned by the Danish Ministry of Defence and the European Space Agency. A CubeSat is a microsatellite consisting of units (U) of 10 cm x 10 cm x 10 cm cubes, weighing in the area of 1 kg. The two GomX-4 satellites are 6U satellites, meaning that they each have six units in a 2x3 pattern. Their goal is to demonstrate different miniaturized technologies, including inter-satellite and high-speed downlink communication, which are considered critical for the potential Ulloriaq case to be a success. As a consequence of using CubeSats, battery-power is a limited resource, which makes usage of faster, more power-hungry inter-satellite communication equipment difficult. Because of this, it is important to be able to plan ahead and minimize the battery load. The constellation is further described in Section 2.1.

The authors of [12] modelled the Ulloriaq case as a Mixed Linear Programming (MILP) problem, optimizing for battery-usage and data-delivery-latency. They structured the model as a delay tolerant network and modelled the battery using a linear battery model. To solve the MILP problem, the authors used Systems Tool Kit (STK) and a plug-in called Contact Plan Designer. The resulting Contact Plan generated as a solution to the MILP problem was then used as input to a stochastic kinetic battery model, to determine the risk of any of the satellite's batteries depleting using a more realistic battery model.

As a continuation of the work done with the MILP model, we propose solving the scheduling problem by modelling the Ulloriaq constellation using Timed Automata (TA), to explore what benefits this approach could provide over the existing MILP approach. This was done using different tools from the UPPAAL family, in particular, UPPAAL CORA and UPPAAL SMC. These are used as steps in a tool-chain which takes data related to the satellites orbits and internal state, such as the state of charge of the batteries, and in the end produces a schedule in the form of a validated contact plan, which can be used to schedule a satellite. To achieve this, the first step is for the CORA engine to find a trace that satisfies the goal for the current mission. In this case, it would be to collect and deliver a certain amount of data. This goal-query is then applied to the Ulloriaq modelled as a network of Priced Timed Automata. With the help of CORA and a fitting goal-query, the trace is optimized to use the least amount of battery-power while maximizing the speed at which data is transferred from Greenland to Aalborg. The trace is then fed to a custom tool called *Ulloriaq Contact Plan Extractor*, which parses the trace generated by CORA, and outputs the corresponding contact plan. Lastly, to validate the contact plan, a post-processing step is performed, where the contact plan is fed to a KiBaM model, implemented as a Stochastic Timed Automata in SMC. This model calculates the probability of the generated contact plan depleting the satellite's battery, and produces corresponding battery usage graphs. Furthermore, the SMC model also adds some uncertainties to different variables to better model a real-life scenario concerning jitter and battery charge levels.

Chapter 2 contains a description of the Ulloriaq case, and the underlying theory of the case and the models presented in this paper is described. In Chapter 3, the model used in [12] and the results from the paper are presented and explained to give a basis for comparison. After this, the models made for this project are presented in Chapter 4. Lastly, to measure the success of the models created, we have conducted experiments with different aspects of the model and compared the results from using UPPAAL with those from the MILP model from [12], the results for both are found in Chapter 5.

2 | Preliminaries

This chapter contains the preliminaries necessary to understand the different models made during this project. Firstly, the Ulloriaq is described, and the different principles used to handle different hurdles in the problem domain. After this, a description is given of the main tools used in this paper; UPPAAL, UPPAAL CORA and UPPAAL SMC, as well as a general description of contact plans and the different battery models applicable to a satellite like the ones used in Ulloriaq.

2.1 The Ulloriaq case

The case used as a base for this project is defined in [12]. The case is called the Ulloriaq constellation and was proposed by a company called GOMspace[9]. GOMspace proposes having a constellation of CubeSats in a train-like formation, forming a ring around the Earth, as shown in Figure 2.1. The goal of the mission is to collect observations and remote sensing data of airplanes flying above the Greenland area and deliver it to a ground station in Aalborg. The purpose of these flights is to maintain Danish sovereignty of the Greenland territory.

The proposed constellation consists of ten of GOMspace's GomX-4 satellites [10]. The GomX-4 type of satellite is a 6U CubeSat. Each satellite is equipped with two inter-satellite antennas pointing towards its forwards and backward neighbors. The satellites would form a ring, around the Earth, spaced out equally, with a high revisit-rate over Greenland. Because of the nature of the orbit, the time in which a satellite is above an area is different for each orbit around the Earth. This is also the case for the time in which each satellite is in sunlight and eclipse.

Two of the satellites would also be equipped with a High-Speed Link (HSL) transponder, to perform high-speed communication with the ground station in Aalborg. These satellites would be placed across from each other, as shown by the red-colored satellites on Figure 2.1. Every satellite would have the ability to send data to it's forward- or backward neighbor, to facilitate delivering the data to the satellites equipped with HSL transponders. A satellites

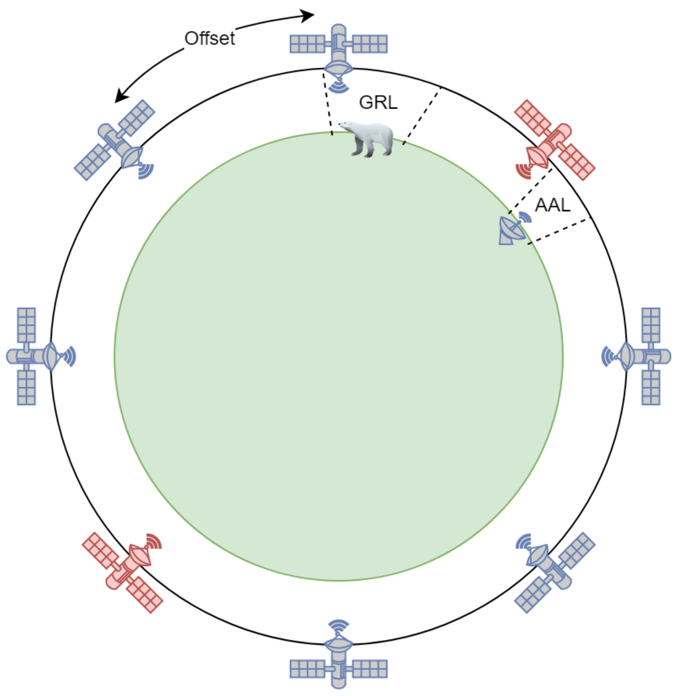


Figure 2.1: The proposed Ulloriaq constellation

neighbors are indicated by a line between two satellites on Figure 2.1. Lastly, every satellite would have the ability to collect data from the designated area over Greenland. The Greenland and Aalborg areas are marked on Figure 2.1 using a cone with a polar bear and antenna for each area, respectively.

The different constraint values for the Ulloriaq case are listed in Table 2.1. The values are split into three categories: Battery constants, Data transfer constants, and mission constants. The battery constants include; The different capacities for the batteries, the background battery load required for the satellite to function, the battery load required for transmission and the rate at which the battery is recharged while in sunlight. The satellites do not necessarily start at their maximum battery charge, since a period to be scheduled always continues a previous schedule. This is represented by the *Initial charge* constant, which will be different for each period being scheduled. This also means that to be able to generate a new schedule after the current one, the satellites must be left with enough battery charge. Because of this, a minimum charge for the battery has been defined.

The data transfer constants represent the constants associated with inter-satellite communication, data collection, and HSL transmissions. The size of the satellites data buffer is not defined and is assumed to be big enough not to cause memory constraints. Lastly, the constants related to the mission are the amount of data being fed to the satellites to collect from

Battery constants	
Initial charge	221644.8 J
Maximum charge	277056.0 J
Minimum charge	166233.0 J
Background load	4.630 J/s
Transmission load	13.651 J/s
Recharge rate	15.472 J/s
Data transfer constants	
Inter-sat transmission speed	10 kb/s
Greenland collection speed	10 kb/s
Aalborg transmission speed (HSL)	100 Mb/s
Mission constants	
Data fed to satellites	187.5 MB
Contact plan length	48 h (172,800 seconds)

Table 2.1: Constant values as seen in [12]

above Greenland and the length for which a schedule should be generated.

2.2 Communication Protocols

Since the goal of the Ulloriaq case is to collect data and deliver it through a network, it is important to consider how to structure this communication. There is a list of constraints for the case that must be taken into consideration when deciding how to do this best, these are:

1. Only two satellites are capable of transmitting data to Aalborg, and these are spread evenly out in the constellation.
2. Battery is a limited resource and is only replenished while in sunlight.
3. The Aalborg transmission speed is much faster than the inter-satellite transmission speed.

Because of the constraints above, it is not possible to have a persistent connection from the collection zone, Greenland, to the destination, Aalborg. This is because there is not always a satellite above both Greenland and Aalborg simultaneously and even if there is, one of the satellites needed to complete the connection might not have sufficient battery charge to do so. This makes an end-to-end protocol, transmitting real-time data, unfavorable since it would not be able to establish a full connection most of the time. Another downside of only transmitting data real-time is that it would only be possible to utilize 0.01 % of the HSL band when delivering data to Aalborg, limiting the throughput to 10 kb/s when connected. To better utilize the connection to Aalborg and to take the above constraints into account, the authors of [12] suggests modelling the constellation as a Delay-Tolerant Network (DTN).

2.2.1 Delay-Tolerant Networking

A delay-tolerant network (DTN) is a disruptive communication architecture, originally designed and studied to be used as an architecture for an interplanetary internet [8]. Beyond this, the architecture is well suited for environments where a constant connection cannot be guaranteed. Communication happens over a network, consisting of multiple nodes, which might be unreachable for some time.

DTNs uses a store-carry-forward messaging scheme, meaning that data can be sent from a source node to a destination node through some relay nodes. This is extended such that data can be stored on a node, in a buffer, before a connection can be made to another node, at which point the data will be forwarded. As an example, compare DTNs to the postal service (e.g., Post Nord, UPS, etc.). The postal service can be thought of as a delay-tolerant network, where the payload, in that case, a letter, is stored in a location (node), such as a mailbox or post office until it can be moved (forwarded) to its destination [8]. For a DTN to function optimally, it is essential that the storage available is well distributed over the nodes of the network and that it is persistent.

A connection between two nodes is called a contact. The time at which contacts should take place in order to reach a goal optimally, and without violating any constraints, can be calculated beforehand. This is done using a list of time-periods in which different kinds of communication can take place. The generated collection of contacts, e.g., the times in which an action should take place, is called a contact plan.

For the Ulloriaq case, the ten satellites, as well as Greenland and Aalborg, would be considered nodes of the network. In this case, data would be *stored* at Greenland until a satellite passes above, at which point the data would be *forwarded* to said satellite. The satellite would then *carry* the data until it is possible to *forward* it to another satellite closer to a downlink, or towards a satellite equipped with an HSL transponder. The data would be carried and transmitted until the data reaches an HSL-enabled satellite, which would then *carry* the data until it is above Aalborg, at which point it would *forward* the data to its destination, Aalborg.

An example of the constellation proposed for the Ulloriaq case can be seen on Figure 2.2. The Figure is split into four-time slices. Figure 2.2a shows a constellation of three satellites, one equipped with an HSL transponder, orbiting above the earth. On the next Figure 2.2b, the middle satellite is collecting data from Greenland, which can be seen as Greenland forwarding data to the satellite. Figure 2.2c shows the middle satellite forwards the data to the HSL-enabled satellite, and the leftmost satellite collects data from Greenland. Between figures 2.2c and 2.2d, the HSL-enabled satellite carries the data, waiting to be in range of Aalborg. On the final Figure 2.2d, the HSL-enabled satellite forwards the data to Aalborg, and the leftmost satellite forwards its data to the middle satellite.

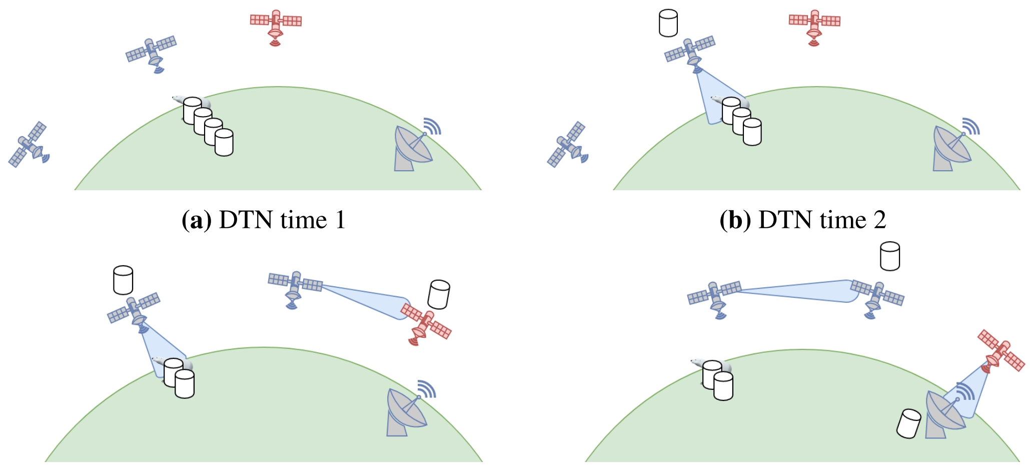

(a) DTN time 1 **(b)** DTN time 2 **(c)** DTN time 3 **(d)** DTN time 4

Figure 2.2: Example of DTN

2.2.2 Contact Plans

In the context of a Delay-Tolerant network, a contact is an opportunity for two nodes to communicate, meaning that data can be forwarded. A contact described by: The node forwarding, the node receiving, the start time for the communication and the end time for the communication *(fromNode, toNode, startTime, endTIime)*. A contact plan (CP) is then a set of contacts for the different nodes in the DTN [8], chosen to adhere to some constraints. A Contact Plan is a subset of all possible contacts.

Since the satellites are in a train-like formation, in the same orbit, they are always within distance to establish an inter-sat connection. Greenland and Aalborg are however only able to communicate with satellites that are within their physical zone. As such, ignoring all other constraints, it would be possible to have a Contact Plan with two contact, one for each direction, for each satellite neighbor pair from the start of the period being scheduled all the way till the end, as well as a contact each time a satellite was above either Greenland or Aalborg. In reality, this is however not possible, due to the physical, and mission-related constraints, such as the state of the satellite's batteries.

A contact plan can be visualized by a Gantt-chart, where each row represents a contacts between either two nodes, either two satellites or a satellite and one of the earth stations. An example of such a Gantt-chart can be seen in Figure 2.3. This example CP is based on the communications examples from Figure 2.2.1. Here, the left-most satellite is *sat0*, the middle one is *sat1* and the HSL equipped satellite is *sat2*. The first thing that happens is that *sat1* collects data from Greenland, it then forwards this to *sat2*, while simultaneously *sat0* collects data from Greenland. *sat0* then forwards the data to *sat1*, while *sat2* gets in position

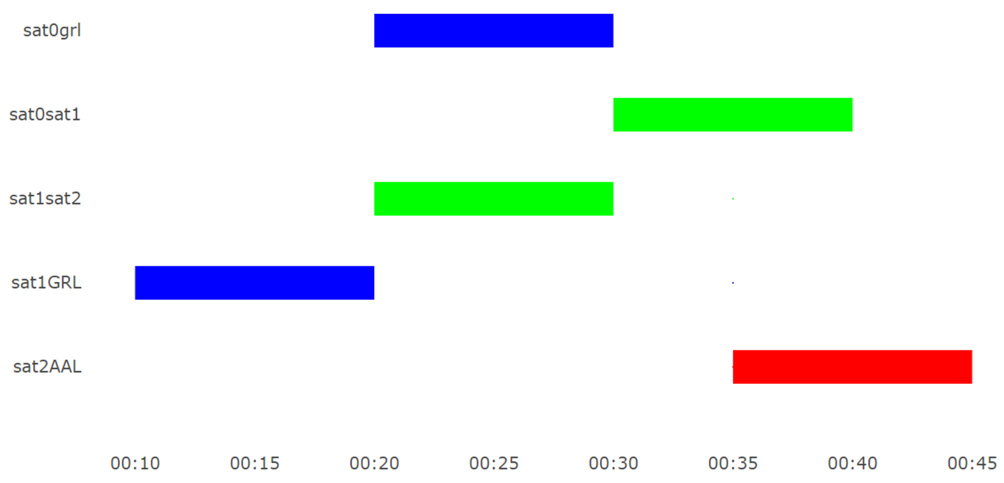


Figure 2.3: Example contact plan – Green: InterSat, Blue: Greenland, Red: Aalborg

to forward the data to Aalborg. When *sat2* is in position, it forwards the data, while *sat0* finishes forward its data.

2.3 UPPAAL

UPPAAL is a tool used for modelling and verification of real-time systems. The models are made using networks of Timed Automata (TA), extended with data types [22]. This section contains a description of the UPPAAL tool as well as the syntax and informal behavior of Timed Automata. The rest of this section is based on [14] and [4].

Modelling and verification in UPPAAL are done by first creating a model, using one or more networked TAs. Different queries can then be performed on the model using the UPPAAL Verifier to verify various aspects of it, such as liveliness. The model can also be explored using the UPPAAL Simulator, which allows the user to step through the model.

A Timed Automaton is an extension of a finite-state machine, with the addition of clocks. A clock is a variable, which evaluates to a real number. In UPPAAL, all clocks progress synchronously. On top of this, UPPAAL supports custom data structures, declared using a C like scripting language.

An example of a TA can be found in Figure 2.4, modelling a simple satellite. The modelled satellite can do three things: collect data from Greenland, send data to a neighboring satellite, or receive data from a neighboring satellite. The content of this example satellite is presented in Table 2.2. The example satellite can do three things, besides being in an

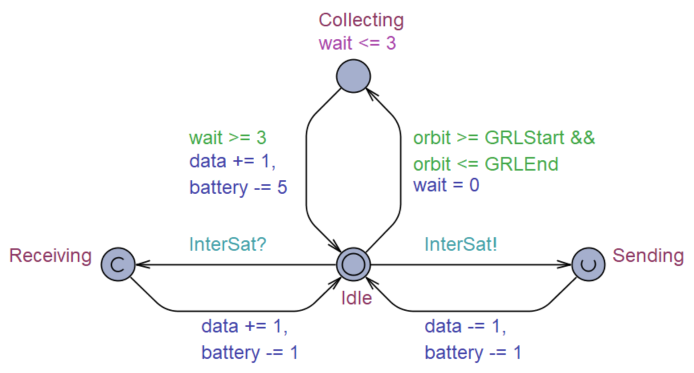


Figure 2.4: TA model of a simple satellite

idle state: Receive, collect, and send data. Each action represents an action a real satellite could take, and the corresponding transitions drain a fictional battery as well. Furthermore, the collecting action has constraints that prevent it from happening when the satellite is not above Greenland.

Locations

What would traditionally be called a state in a finite-state machine is called a location in UPPAAL. These locations are connected by edges; both described later in this section. Locations are defined by a circle in the graphical representation of a TA. The example TA seen on Figure 2.4 has four locations: `Idle`, `Collecting`, `Sending` and `Receiving`. A location can be either urgent, committed or neither.

When the system is in an urgent location, time may not pass. A committed location is more strict than an urgent location. In a committed location, time may not pass either, but in addition to this, the next action taken must involve an outgoing edge, going from a committed location. An urgent location is annotated by a 'U' in the center of a location, and a committed location is annotated by a 'C'. An example of an urgent location is seen on the `Sending` location and a committed location on the `Receiving` location.

Each TA must contain an initial location, which is the starting point for that TA. An initial location is annotated by a double circle, as seen on the `Idle` location.

Each location can have an invariant. An invariant is a Boolean expression assigned to a location. This expression must be true for the system to be in that location. An example of how an invariant can be used is seen on the `Collecting` location. Here the invariant `wait <= 3`, where wait is a clock, combined with the two edges to and from the location (explained below), forces the system to stay in the `Collecting` location for exactly three

Locations	
Idle	Represents when the satellite idling in its orbit
Collecting	Represents when the satellite is collecting data
Sending	Represents when the satellite is sending data
Receiving	Represents when the satellite is receiving data
Clocks	
orbit	Clock used to represent how far the satellite is in its orbit
wait	Clock used to delay the satellite during data collection
Channels	
InterSat	Channel used to communicate with neighboring satellites
Constants	
GRLStart	Time for when the satellite enters Greenland. In this example 10.
GRLEnd	Time for when the satellite leaves Greenland. In this example 25.
Variables	
data	Represents the data in the satellites buffer
battery	Represents the state of charge for the satellite battery. The initial charge of the battery is 100

Table 2.2: Content of TA on Figure 2.4

time units.

Edges

Edges are transitions going from one location to another or looping to the same location. An edge can be fired separately or synchronously with an edge on another TA using a channel. The edge going from `Idle` to `Sending` on Figure 2.4 are fired synchronously with the edge going from `Idle` to `Recieving` on a different TA. The synchronization happens over the channel `InterSat!`, where the satellite firing the edge going to `Sending` is sending, denoted by the exclamation mark post-fixed to the channel name. Likewise, the receiving TA has a question mark post-fixed to the channel name. Using this method, the sending TA can only fire the edge when there is another TA ready to fire its receiving edge. An alternative to this is to use a broadcast channel. A TA sending on a broadcast channel can fire even when there is no receiving TA, and any number of TAs can receive at the same time on the same broadcast channel.

Each edge can have an associated guard. A guard is a Boolean expression, and the edge can only be fired if the guard evaluates to true. An example of a guard can be seen on the satellite TA on the edge going from `Idle` to `Collecting` and the one going from `Collecting` to `Idle`. The guard defined as `wait >= 3` limits the edge to only being fired when the clock *wait* evaluates to 3 or greater. The guard on the edge going from `Idle` to `Collecting` is only enabled when the orbit clock is within the span defined by the constants *GRLStart* and *GRLEnd*.

When firing an edge, updates can be made to the data included in the model. This is done by adding an `update` expression to an edge. An example of one such update can be seen on the edges going to `Idle` from `Receiving` and `Sending`. Both updates make a call to a function, defined in the declaration of the template.

Behavior

The state of a system of TAs describes the current location for all TAs, the clock values and the values of its associated variables. For the initial state of a system, all TAs are in their initial location, and all clocks evaluate to zero. The system changes state by making a transition, this can be to either delay or to take an action, by firing an edge. The behavior, or semantics, of TAs, are in fact defined as a transition system, meaning that the behavior of a system can be expressed as a list of transitions. An example of this, using the example TA on Figure 2.4 starting from the initial location at time zero, could look like the following. The variables being updated are surrounded by curly brackets:

1. Delay 10 {orbit = 10, wait = 10}
2. Action `Idle` –> `Collecting` {wait = 0}
3. Delay 3 {orbit = 13 wait = 3}
4. Action `Collecting` –> `Idle` {data = 1, battery = 95}
5. Action `Idle` –> `Collecting` {wait = 0}
6. Delay 3 {orbit = 16 wait = 3}
7. Action `Collecting` –> `Idle` {data = 2, battery = 90}
8. Delay 6 {orbit = 22 wait = 9}
9. Action `Idle` –> `Receiving` {InterSat?}
10. Action `Sending` –> `Idle` {data = 3, battery = 88}
11. Action `Idle` –> `Receiving` {InterSat?}
12. Action `Sending` –> `Idle` {data = 4, battery = 87}
13. Action `Idle` –> `Receiving` {InterSat?}
14. Action `Sending` –> `Idle` {data = 5, battery = 86}

A set of transitions, such as the one above, from one location to another, is called a trace. UPPAAL can be used to find such a trace for a system, from the initial state to some defined state. This is further described in Section 2.3.1. On this example trace, the satellite delays until it is above Greenland, and then start to collect data, by firing the edge from `Idle` to `Collecting`. When in `Collecting`, the satellite must delay for three time units before it can collect the data and return to `Idle`. This delay represents the time it takes for the satellite to collect the data. The satellite collects data from Greenland twice. After this, the satellite satellites delays waiting for one of its neighbors to transmit data to it. When another satellite is ready and broadcasts on the `InterSat!` a connection is established, and the satellite receives one data by going to `Receiving` and back to `Idle`. This is repeated three times.

2.3.1 Verification

The goal of using UPPAAL is to be able to verify different properties for systems, modelled as networks of TAs. This can be done either through the GUI built for UPPAAL, where a query can be defined and settings regarding the verification can be selected, such as the algorithm used for exploring state-space, or it can be done through a command-line interface.

UPPAAL supports five different types of queries, namely:

A<> p	- For all paths p is *eventually* satisfied
E<> p	- There exists a path where p is *eventually* satisfied
A[] p	- For all paths p is *always* satisfied
E[] p	- There exists a path where p is *always* satisfied
p –> q	- If p is satisfied q will eventually be satisfied

Where *p* and *q* are states defined as formulas. Some examples queries for the TA on Figure 2.4, could be:

- *E<> data >= 5 && battery > 85 &&orbit <= 360*
  The query is satisfied if there exists a trace where the satellite will be in a state where the satellite has collected 5 or more data, the battery charge is greater than 85 and the time is within 360 time units. The example query above could be a witness-trace for this query.
- *A[] battery > 0*
  The query is satisfied if no satellite can run out of charge.

UPPAAL allows for three different search orders: Breadth First, Depth First, and Random Depth-First. The result of running a query is a Boolean value for whether the property is satisfied or not. It is possible to ask UPPAAL to generate a witness-trace that satisfies a query. This trace can be either: Some trace, the shortest trace or the fastest trace. Here shortest means the trace with fewest transitions and fastest means fastest in regards to the time used in the model.

2.4 UPPAAL CORA

UPPAAL CORA is a tool in the UPPAAL family of tools. CORA is an abbreviation for Cost Optimal Reachability Analysis [3][20]. It uses an extended Timed Automata, with the notion of price, called Priced Timed Automata (PTA) [2]. An example of a PTA can be seen in Figure 2.5, which is a modified version of the TA from Figure 2.4. The price of a PTA is implemented by assigning a cost to a transition. For a delay transition, the cost is added by a given rate on a transition, accumulating while delaying at that location. For an action transition, the cost is added as a discrete cost for taking that action. An example of a cost rate can be seen on the `Idle` location. Here the cost increases with 1 for each time unit delayed as defined by `cost'==1`. For the discrete cost, an example can be seen on the edge going from `Idle` to `Collecting`. This results in the satellite being penalized for idling and for sending data to other satellites.

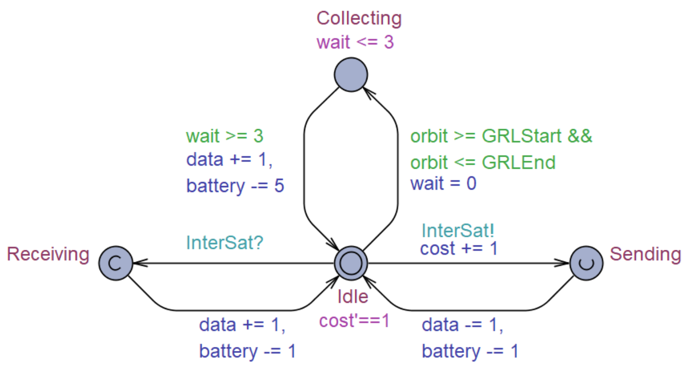


Figure 2.5: PTA model of a simple satellite

The behavior of the PTA is also modified by adding a cost to each transition. Recalling the trace for the simple satellite, when extended with the notion of cost, will look as follows:

1. Delay 10 {orbit = 10, wait = 10, cost=10}
2. Action `Idle` –> `Collecting` {wait = 0, cost=10}
3. Delay 3 {orbit = 13 wait = 3, cost=10}
4. Action `Collecting` –> `Idle` {data = 1, battery = 95, cost=10}
5. Action `Idle` –> `Collecting` {wait = 0, cost=10}
6. Delay 3 {orbit = 16 wait = 3, cost=10}
7. Action `Collecting` –> `Idle` {data = 2, battery = 90, cost=10}
8. Delay 6 {orbit = 22 wait = 9, cost=16}
9. Action `Idle` –> `Receiving` {InterSat?, cost=16}
10. Action `Sending` –> `Idle` {data = 3, battery = 88, cost=16}
11. Action `Idle` –> `Receiving` {InterSat?}
12. Action `Sending` –> `Idle` {data = 4, battery = 87, cost=16}
13. Action `Idle` –> `Receiving` {InterSat?}
14. Action `Sending` –> `Idle` {data = 5, battery = 86, cost=16}

Here, the cost is incremented when the satellite delays while in the `Idle` location and when firing the edge with a cost associated. The cost for the trace is the sum of the cost of each transition.

2.4.1 Optimal Scheduling

As with UPPAAL, the goal when using CORA is to find a cost-optimal trace that satisfies a query. On top of the verification options possible in UPPAAL, as presented in Section [22], it is possible to ask for an optimal trace to a state, from the initial state. The types of queries possible are identical to those of UPPAAL, but both new search order and options for the trace to be generated are different. Three new search orders are available:

- **Smallest heuristic value first**
  This search order uses an optional meta variable built into CORA called *heur*, short for heuristic. When declared, the algorithm used for the query explores the states ordered from smallest to largest.
- **Best first**
  This search order orders the states to be searched through by their current cost.
- **Random best depth first**
  Is a depth-first strategy, where the waiting list is ordered by the current cost of the state, with random restarts.

To utilize the added cost, it is possible to ask CORA to find the best trace to reach a goal-state, where best is the trace with the lowest accumulated cost. UPPAAL CORA uses a branch and bound algorithm, which function like the algorithm used in classic UPPAAL, but with the addition of pruning of state-space. UPPAAL CORA allows the user to maintain a built-in variable called *remaining*, doing so can help the branch and bound algorithm used for searching the state-space to be more aggressive in its pruning. The variable should be maintained such that it is always equals to the minimum cost.

2.5 UPPAAL SMC

UPPAAL SMC [5], short for Statistical Model-Checker, is yet another tool in the UPPAAL family of tools. Like UPPAAL CORA, SMC utilizes an extended version of the classical TA, namely a Stochastic Timed Automata (STA). An STA adds stochasticity to the classic TA. Furthermore, STAs are hybrid systems, meaning that they can model both discrete and continuous behavior. An example of a satellite modelled as an STA can be seen in Figure 2.6. The stochasticity allows for non-deterministic choices to be made probabilistically.

When exploring the model probabilistically, each decision is made non-deterministically. The different ways this can happen is represented in the example in Figure 2.6. The example satellite has three locations, `Idle`, which is the initial location, `Receiving` and `Collecting` . When in `Idle` the satellite can either collect data from Greenland or receive data from

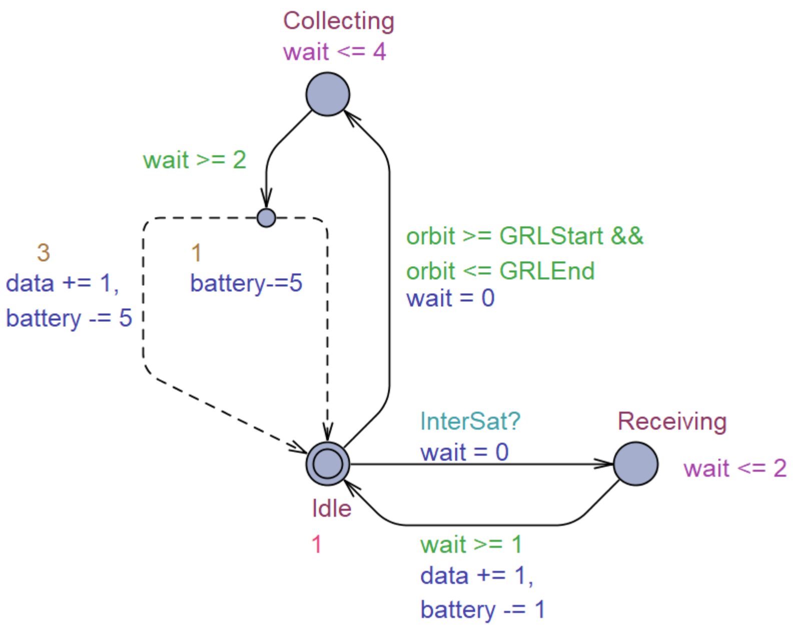


Figure 2.6: STA model of simple satellite

its neighbor. Contrary to the TA and the PTA example satellites, the STA example satellite cannot send data to its neighbors. The satellite only receives data when a supplying satellite broadcasts on the `InterSat?` channel. An example of a simple supplier can be seen in Figure 2.7. The satellite will go from `Idle` to `Collecting` when within the start and end time for Greenland with a exponential rate of *1*, as seen by the number *1* below the name of the location and the guard `orbit >= GRLStart && orbit <= GRLEnd` on the edge going to the location. Defining an exponential rate is one of two ways to define a delay; the second one can be seen on the `Collecting` location. Here a bound delay is defined by the invariant `wait <= 4` and the guard `wait >= 2`. Together these define a bound delay of $[2, 4]$, which is explored with a uniform distribution. When there are multiple edges to choose from, they are fired uniformly, unless a distributing is defined. For the edges going from `Collecting` to `Idle` a distribution is defined, this distribution can be seen on the dotted lines going to `Idle`. This distribution represents the data collection failing with a rate of $\frac{1}{4}$ and succeeds with a rate of $\frac{3}{4}$. The simple supplier on Figure 2.7 supplies the satellite with an exponential rate of $\frac{1}{3}$, written as $1 : 3$.

2.5.1 Querying

Using STAs allows for two new forms of queries. It is possible to query for the probability of reaching some state or to simulate a number of runs.

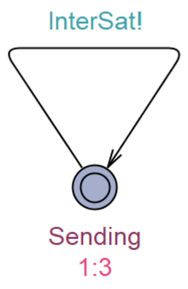


Figure 2.7: STA simple supplier

Pr[<=bound](<>p) calculates the probability of reach state *p* within *bound* time. A probability query for the STA satellite from Figure 2.6 could be:

- Pr[<=180](<> sat.data > 15 && sat.battery > 25)

Which calculates the probability of the satellite having more than 15 data and more than 25% charge left in the battery, the query is bounded to only make runs up to 180 time units. Running this query on the model gives a probability of 44.2% - 54.2% with confidence of 0.95. The calculations are based on 402 runs.

simulate N [<=bound] {E1,...,Ek} makes *N* simulations for *bound* time units. Measuring the values of expressions E1 - Ek. After performing a simulation, the user can choose to explore the monitored values over time as a diagram. An example of such a simulation diagram can be seen in Figure 2.8. The example simulation monitors the value of the satellite battery (green), how much the satellite has (red), when the satellite collects data (turquoise) and when the satellite receives data from the supplier (purple).

2.6 Battery Models

When modelling cyber-physical systems, where battery is a limited resource, it is important to consider how to model the battery. There are many different approaches for modelling a battery, the simplest being the Linear Battery Model (LiBaM), which is only concerned with the state of charge (SOC) of the battery. [11] gives an overview of different approaches. On the opposite end of the complexity scale are the Electro-chemical model and the Electrical-circuit model. These are much closer to how a battery actually works in the real world, however, at the cost of being complex to implement. To use the Electro-chemical model, it is necessary to have a very detailed understanding of the battery being modelled, since constants such as the thickness of the electrodes and the initial salt concentration needed for the model to work.

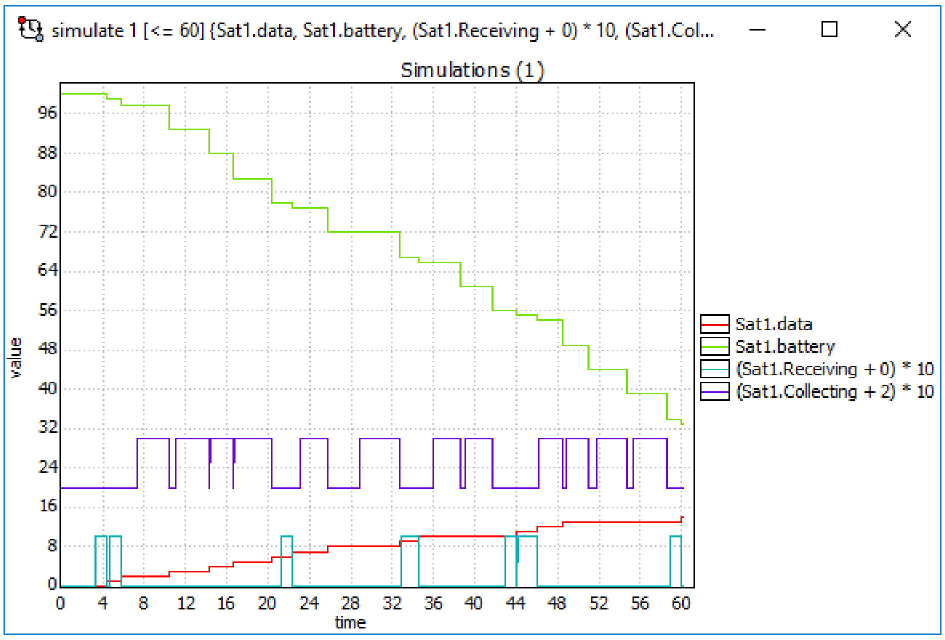


Figure 2.8: Result for: *simulate 1 [<= 60] {sat.data, sat.battery, (sat.Receiving + 0) * 10, (sat.Collecting + 2) * 10 }*

A good compromise between the simple and naive model, and the complex models are to use an Analytical model. One such model is the Kinetic Battery Model (KiBaM). In this chapter contains a description of both the Linear Battery Model and the Kinetic Battery Model.

2.6.1 Linear Battery Model (LiBaM)

LiBaM is a simple and naive representation of a battery. The battery is represented only by its state of charge, which can be thought of as a well of liquid with a pipe a pipe at the bottom. An illustration of this can be seen on Figure 2.9. The flow to or from the pipe represents the current flow of the battery.

Even though this model is naive compared to other battery models, it is still useful, since it is easy to implement and most important, can be modelled linear, as the name indicates.

2.6.2 Kinetic Battery Model (KiBaM)

KiBaM is an analytically representation of a lithium-ion battery. It models the chemical reactions in the battery using two wells, *Bound charge* and *Available charge*. An illustration of the KiBaM wells can be seen on Figure 2.10. Like with LiBaM, each well contains a liquid, representing the charge. [23]

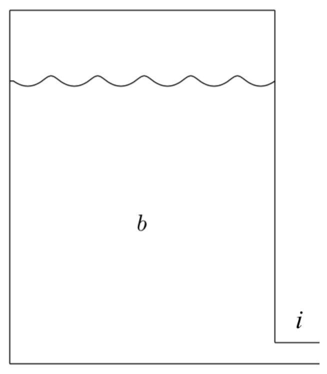


Figure 2.9: LiBaM state of charge

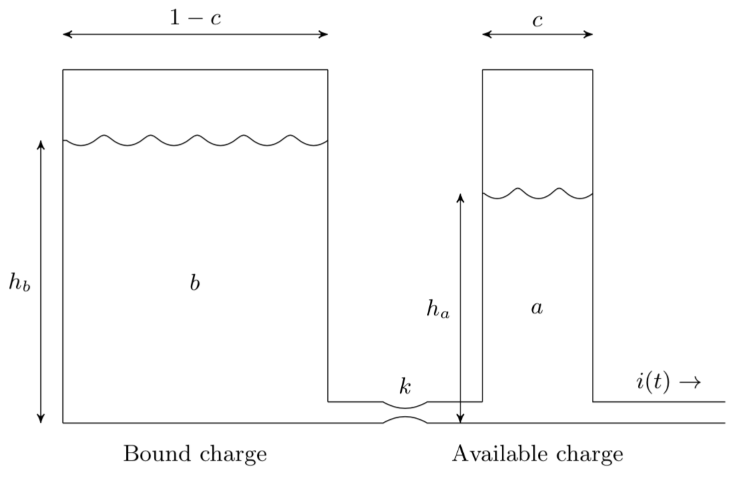


Figure 2.10: The two wells of the KiBaM [23]

Charge flow from *bound charge*, to *available charge* trough the valve connecting them, defined by constant *k*. The rate at which this exchange happens can be expressed by the differential expressions on Equation 2.1. Here *i* is the rate at which charge leaves the battery. The width of the two wells are c and $1-c$ respectively. *a* and *b* represents the current charge in reach well. h_b and h_a are the height of the liquid in the two wells. These can also be written as $h_a = \frac{a}{c}$ and $h_b = \frac{b}{1-c}$.

$$\begin{cases} \frac{\mathrm{d}a}{\mathrm{d}x} = -i + k(h_b - h_a) = -i + k(\frac{b}{1-c} - \frac{a}{c}) \\ \frac{\mathrm{d}b}{\mathrm{d}x} = -k(h_b - h_a) = -k(\frac{b}{1-c} - \frac{a}{c}) \end{cases} \tag{2.1}$$

KiBaM can be added to a model just by expressing the two differential equations from Equation 2.1 in the model and then maintaining the *i* variable. This simplicity makes it an ideal choice for modelling the satellite batteries for the Ulloriaq constellation. A downside however is the continuous behavior required, which is difficult to model using Timed Automata. A solution to this is to use UPPAAL SMC and make the KiBaM as an STA, which is a hybrid system, meaning it allows for both discrete and continuous behavior.

3 | Ulloriaq as a Mixed-Integer Linear Programming Problem

This chapter contains a description of how the authors of [12] modelled the Ulloriaq case as a Mixed-Integer Linear Programming (MILP) problem. The MILP problem is solved using a tool called System Tool Kit and a plugin called Contact Plan designer. The product of this is then a contact plan, which is run through a verification step, which verifies if the contact plan is valid using a more realistic battery model, namely KiBaM.

3.1 Preliminaries for MILP

Before the MILP problem is presented in Section 3.2, some fundamental theory necessary to understand how the case is modelled is presented. The following sections are based on [18].

3.1.1 Linear Programming

Linear Programming (LP) is a method for finding an optimal solution to a given problem, based on a set of constraints, modelling the different limitations and properties of the problem, as well as an objective function, representing the desired properties to optimize for. This optimization can be either to find the maximum or minimum value for the objective function. An example of this could be some of the properties in Ulloriaq case, such as battery constraints, or data transmission rates.

In general, the format of an LP problem is as follows:

$$\begin{aligned} \text{Maximize} &: cx \\ \text{Subject to} &: Ax \leq b \\ \text{And} &: x \geq 0 \end{aligned}$$

Where x is a vector consisting of the variables to be determined, A is a matrix, and b and c are vectors of known coefficients.

Since LP allows for continuous variables, it is used to solve inherently continuous problems. A solution to an LP problem is an assignment of real values to the continuous variables satisfying the constraints and minimizing the objective function.

3.1.2 Integer Programming

While LP in itself allows for variables that range over the reals, Integer Programming (IP) allows only for variables that range over the integers. This makes it more suited for the category of discrete problems, such as how to assign workers to jobs to maximize profit.

A concrete IP model of a simple satellite could be:

$$\text{Maximize: } a_1 + 2a_2 \tag{3.1}$$

Subject to:

$$a_1 - a_2 < 0 \tag{3.2}$$

$$1000 - 10a_1 + 15a_2 > 0 \tag{3.3}$$

$$a_1 \geq 0 \tag{3.4}$$

$$a_2 \geq 0 \tag{3.5}$$

In this simple example, the objective is to maximize the number of actions taken, without depleting the battery. The satellites can perform two different actions a_1 and a_2. The variables a_1 and a_2 in the model denote how many times the satellite has taken each action. Equation 3.1 denotes the objective function, which is to maximize the number of times the two actions have been performed. Furthermore, we get that it is twice as valuable to do action a_2 than a_1. The first constraint, seen on Equation 3.2, requires that a_2 is performed more than a_1. The second constraint, seen on Equation 3.3, represents the state of charge of the satellite battery. The initial charge is 1000, action a_1 requires 10 power units to run, and a_2 requires 15 power units to run. The total power units used must never be more than the initial state of charge of the battery. The last two constraints, on equations 3.4 and 3.5 makes sure that neither action can be performed a negative amount of times.

This problem is then used as input to a solver, using any of several possible algorithms for solving IP problems. The solution from which will be an assignment of integers to the two variables in the objective function.

3.1.3 Mixed Integer Linear Programming

Mixed Integer Linear Programming(MILP) is a third variant, and the one used in [12] to solve the Ulloriaq scheduling problem. Mixed Integer Linear Programming allows for variables to be either integers or reals. Since MILP allows for this mixing of variables, it is well suited for the situations where a mix of continuous and discrete variables need to be included in the problem. Which is the case in the Ulloriaq case, where some variables are integers, and others are continuous.

Outside of each of these types of LP fitting different kinds of problems, different algorithms and solvers exist to solve different categories of problems as well. In the case of the MILP model for the Ulloriaq case, a simplex solver is used.

3.2 Ulloriaq Case MILP Model

The following section is a brief overview of each part of the MILP model presented in [12]. The objective of the model is to obtain a battery-aware contact plan, detailing the traffic flow through the system, delivering the data with the lowest latency possible.

The data-flow in this model is based on a more general problem: "Multi-commodity flow problem", in which multiple commodities (such as data flowing from multiple sources) flow between nodes of a network. Specifically equations 3.2 - 3.5 corresponds to this problem. A state in the model consists of a discrete time episode, where the current connections between satellites and ground-stations are stable. An edge in the model is a connection between two nodes - a node can be either a satellite or one of the earth stations. Furthermore, Aalborg counts as sinks in this system, as this is where the data is deposited and Greenland as a source node. Table 3.1 contains a list of the variables used throughout the MILP model.

$$\text{minimize:} \quad \sum_{k=1}^{K}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{y=1}^{N}\sum_{z=1}^{N} w(t_k) \cdot X_{k,i,j}^{y,z} + Y_{k,i,j} - C_{k,i}$$

Expression 3.1: The Objective function from [12]

Equation 3.1 is the objective function used in the model. $\mathrm{X}_{\mathrm{k,i,j}}^{\mathrm{y,z}}$ represents the flow of data from satellite y to satellite z, at some state k, using the edges going from satellite y to satellite z, and the edge between node i and j. This is multiplied by $w(t_k)$, which is a cost function that increases over time. $\mathrm{Y}_{\mathrm{k,i,j}}$ is a binary variable used to check connectivity between two satellites, checking the nodes in between the desired nodes. $w(t_k) * X_{k,i,j}^{y,z} + Y_{k,i,j}$ works to penalize later flow of data. $\mathrm{C}_{\mathrm{k,i}}$ is the charge left at the current satellite when the current state ends. Which works to minimize the amount of battery used.

Input	
N	Number of nodes in the system
K	A set of states, detailing time-evolving network connectivity
t_k	Start time of state k
i_k	Duration of state k
$x_{k,i,j}$	Capacity of node i to node j contact at state k
$b_{max,i}$	Satellite *i*'s max data buffer capacity
$d_k^{i,j}$	Traffic from node i to node j at the beginning of state k
p_i	Maximum amount of simultaneous links at node i
M	Used for solving the problem using the simplex method
$c_{min,i}$	Minimum battery charge for node i at all times
$c_{max,i}$	Maximum battery charge for node i at all times
$c_{0,i}$	Satellite i's initial battery charge
$c_{r,C}^{i}$	Recharge rate for satellite i, when exposed to sun
$c_{r,T}^{i}$	Discharge rate for a satellite i, dependent on which communication equipment is used
$c_{r,B}^{i}$	Discharge rate for a satellite i due to background load
Output	
$X_{k,i,j}^{y,z}$	Traffic from y to z during state k, using edges i to j
$B_{k,i}^{y,z}$	Satellite i's buffer utilization at the end of state k, due to the traffic from y to z
$Y_{k,i,j}$	Binary variable used for link selection from i to j at state k
$C_{k,i}$	Battery charge levels at satellite i at the end of state k

Table 3.1: Variables used in the Ulloriaq MILP model

This objective function is used due to its ability to increase cost over time, thus optimizing for low latency with the least amount of battery charge used.

$$\sum_{j=1}^{N} X_{k,j,i}^{y,z} - \sum_{j=1}^{N} X_{k,i,j}^{y,z} = B_{k,i}^{y,z} - (B_{k-1,i}^{y,z} + d_k^{i,z}) \quad \forall k, i, y, z$$

Expression 3.2: Models data flowing between satellites, from [12]

The constraints on equations 3.2 to 3.5 are all elements that model the flow of data based on the aforementioned "Multi-commodity flow problem". The constraint on Equation 3.2 represents data transformation as time progresses. The movement of data between satellites is represented by the left side of Equation 3.2, while the right side represents the storage of internal data. $\sum X_{k,j,i}^{y,z}$ and $\sum X_{k,i,j}^{y,z}$ represents the data being transported between nodes *j* and *i*. $B_{k,i}^{y,z}$ represents the amount of data stored at satellite *i* at the end of the current state, and $B^{y,z}_{k-1,i}$ represents the same, but for the previous state. Finally, $d^{i,z}_{k}$ represents the total amount of traffic in the buffer of satellite *i* at the end of the current state.

$$\sum_{y=1}^{N}\sum_{z=1}^{N} B^{y,z}_{k,i} \leq b^{i}_{\text{max}} \quad \forall k, i, y, z$$

$$B^{y,z}_{0,i} = 0 \quad \forall i, y, z$$

Expression 3.3: Models the bounds and initial status of internal buffers, from [12]

The constraint on Equation 3.3, models the bounds of the internal data buffers of the satellites. $B^{y,z}_{k,i}$ still represents the amount of data flowing into satellite *i*, at state *k*. b^{i}_{max} represents the maximum amount of data that can be stored at satellite *i*. The second line on Equation 3.3 sets the initial state of the data buffers of the satellites to be 0, at state *k*, which in this case is 0.

$$\sum_{y=1}^{N}\sum_{z=1}^{N} X^{y,z}_{k,i,j} \leq x_{k,i,j} \quad \forall k, i, j$$

Expression 3.4: Models a limit on the maximum connection speed between satellites, from [12]

The constraint on Equation 3.4 sets a limit on the maximum transmission speed between satellites. $X^{y,z}_{k,i,j}$ models the traffic from one satellite to another, *y* and *z*, at state *k*, using edges *i* and *J*. This value must be less than $X_{k,i,j}$, which is the maximum transmission speed.

$$\sum_{k=1}^{K}\sum_{i=1}^{N} X^{y,z}_{k,i,z} - \sum_{k=1}^{K}\sum_{j=1}^{N} X^{y,z}_{k,z,j} = \sum_{k=1}^{K} d^{y,z}_{k} \quad \forall y, z$$

Expression 3.5: Models the traffic going from every satellite to the destination, from [12]

The constraint on Equation 3.5 models the traffic from every satellite, to every destination. $X^{y,z}_{k,i,z}$ and $X^{y,z}_{k,z,j}$ model the traffic between two satellites, which in turn balances out with $d^{y,z}_{k}$, modelling the traffic into the buffer of the destination satellite.

$$\sum_{j=1}^{N} Y_{k,i,j} \leq p_i \quad \forall i, k$$
$$\sum_{y=1}^{N}\sum_{z=1}^{N} X_{k,i,j}^{y,z} \leq M \cdot Y_{k,i,j} \quad \forall i, k, j$$

Expression 3.6: Models bounds on the maximum amount of connections a satellite can have at once, from [12]

The constraints on Equation 3.6 sets the bounds for the maximum amount of connections to a single satellite. On the first line, $\mathrm{p_i}$ is the maximum amount of connections, and sum $\mathrm{Y_{k,i,j}}$ is the amount of connections at state *k*. The constraint on the second line provides the same kind of bound, but with regards to the amount of traffic.

$$C_{0,i} = c_{0,1} \quad \forall i$$

Expression 3.7: Models the initial charge of the battery, from [12]

The constraints on Equation 3.7 models the initialization satellite i's battery. $\mathrm{C_{0,i}}$ is the charge of satellite i at time 0, and $\mathrm{c_{0,i}}$ is the initial charge at node i.

$$c_{\text{min}}^{i} \leq C_{k,i} \leq c_{\text{max}}^{i} \quad \forall k, i$$

Expression 3.8: Sets bounds fore the charge as time passes, from [12]

The constraints on Equation 3.8 sets the bounds of the charge for satellite i, with $\mathrm{C_{k,i}}$ being the charge at state *k* at for satellite *i*, and $\mathrm{c^{i}_{min}}$ and $\mathrm{c^{i}_{max}}$ being the lower and upper bound for the satellite.

$$C_{k,i} \leq C_{k-1,i} + (c_{rC}^{i} - c_{rB}^{i} - c_{rT}^{i} \sum_{j=1}^{N} Y_{k,i,j}) \cdot t_k \quad \forall k, i$$

Expression 3.9: Models the change in charge throughout the simulation, using the LiBaM model, from [12]

The constraints on Equation 3.9 is the final equation of the model. It models the battery, using the Linear Battery Model (LiBaM). It has variables for modelling the charging of the battery due to sunlight ($\mathrm{c^{i}_{rC}}$), for the battery consumption due to either transmitting or receiving data($\mathrm{c^{i}_{rT}}$), and for the background load $\mathrm{c^{i}_{rB}}$. The battery constraint is further refined in a post-processing step in [12], using the Kinetic Battery Model (KiBaM).

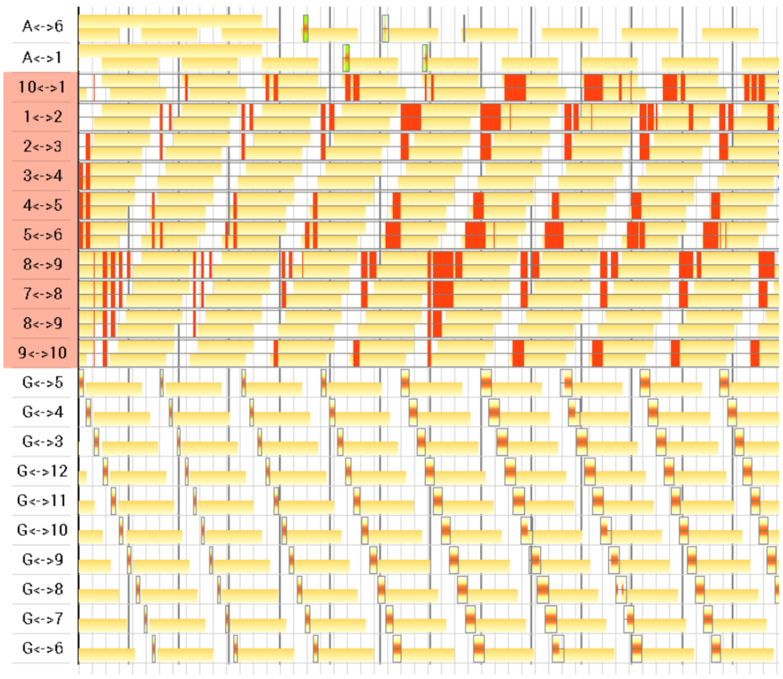


Figure 3.10: Small contact plan as Gantt-chart [12]

3.2.1 Generation of Contact Plans from MILP

The goal of [12] was to generate a contact plan detailing when the satellites should make contacts. For this purpose, they developed a "Contact Plan Designer"[7] plugin for Systems Toolkit(STK)[19]. Systems Toolkit[19] is a simulation program, with facilities for simulating complex physics-based problems. The Contact Plan Designer plugin lets the user generate contact plans based on a MILP model input into STK. An example of such a contract plan can be seen in Figure 3.10.

Each row on the Gantt-chart on Figure 3.10 represents connections between two nodes in the network. The first two rows are connections between the two HSL-enabled satellites and Aalborg. The following ten lines are inter-satellite communication, and the last ten lines are between Greenland and satellites. The yellow background on each row represents when the left-hand satellite of the contact is in sunlight. The red represents the actual contacts between the two nodes.

4 | Models and Toolchain

To solve the problem of scheduling the constellation in the Ulloriaq case, UPPAAL CORA and UPPAAL SMC has been used, as well as a custom tool for generating a contact plan. This was all combined into a toolchain.

UPPAAL CORA and Priced Timed Automata were chosen as the framework for generating contact plans. Using UPPAAL makes it possible to model a problem as a TA through a visual interface, which is easy to work with and communicate about. This made it easy to experiment with and try different approaches to modelling the problem. The ability to find a witness-trace that satisfies a query makes it possible to formulate the end goal for scheduling a time period as a reachability query (E<>), for which a trace is generated. UPPAAL CORA adds to this by adding the notion of cost. This enables querying for a cost-optimal trace, which translates to finding an optimized contact plan. A contact plan can be extracted from a UPPAAL trace by finding all the transitions where the model does something and then record the time at which it happens.

UPPAAL SMC was chosen to make up for a weakness in CORA, namely that it is not possible to model continuous behavior, which made it difficult to model the satellites with using the Kinetic Battery Model. This is however, possible using UPPAAL SMC and modelling the problem as an STA. Furthermore, this allowed for the introduction of uncertainty in the model.

The process for generating a contact plan goes through four phases, as seen in Figure 4.1. The first step is to generate a trace, using a UPPAAL CORA model file (.xml) and a query file (.q), as shown in phase 1 on Figure 4.1. The resulting trace file (.xtr), along with an intermediate file (if), is then parsed in phase 2, which parses these and returns a model-independent trace file (.txt), using a tool called *UPPAAL Timed Automata Parser* (UTAP) [1]. In phase 3, this file is then fed to a custom tool for extracting the contacts, called *Ulloriaq Contact Plan Extractor*. Finally, part 4 on Figure 4.1 is a verification step, that verifies the contact plans battery usage using a model created in UPPAAL SMC. After this step, the contact plan is ready to be sent to the constellation of satellites.

In Section 4.1 the Priced Timed Automata (PTA) model of the Ulloriaq case, henceforth

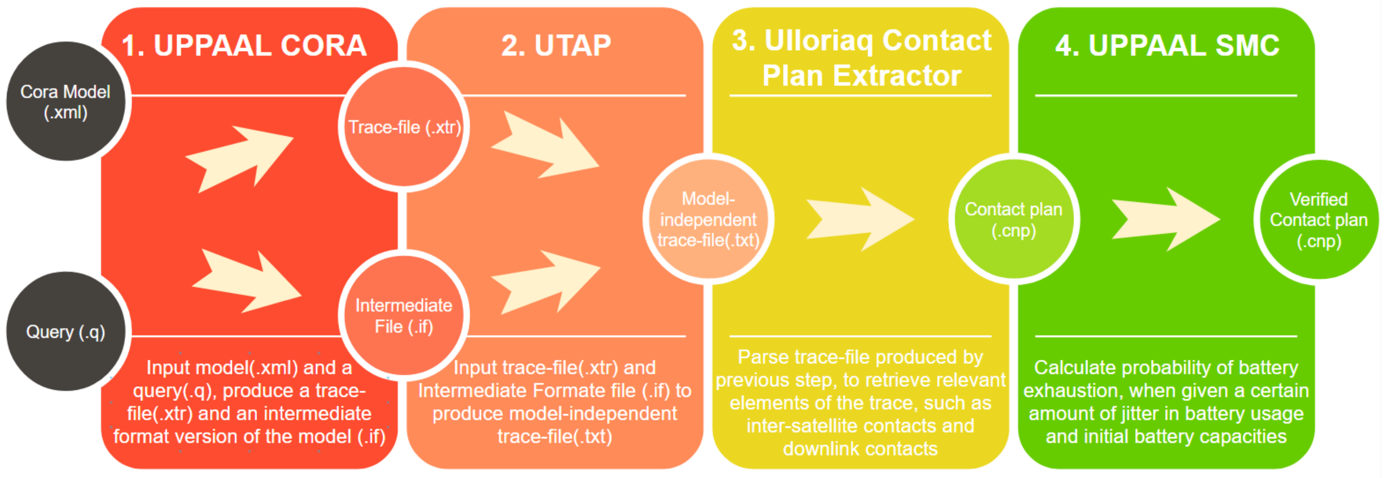

1. UPPAAL CORA
2. UTAP
3. Ulloriaq Contact Plan Extractor
4. UPPAAL SMC
Cora Model (.xml)
Query (.q)
Trace-file (.xtr)
Intermediate File (.if)
Model-independent trace-file(.txt)
Contact plan (.cnp)
Verified Contact plan (.cnp)
Input model(.xml) and a query(.q), produce a trace-file(.xtr) and an intermediate format version of the model (.if)
Input trace-file(.xtr) and Intermediate Formate file (.if) to produce model-independent trace-file(.txt)
Parse trace-file produced by previous step, to retrieve relevant elements of the trace, such as inter-satellite contacts and downlink contacts
Calculate probability of battery exhaustion, when given a certain amount of jitter in battery usage and initial battery capacities

Figure 4.1: The tool-chain enabling contact-plan generation and verification

known as *the Ulloriaq Scheduling Model*, is presented. After this, the Ulloriaq Contact Plan Extractor is explained, as well as an explanation of the tool UTAP. Lastly, a final section explains the Stochastic Timed Automata (STA) model used for contact plan verification is presented.

4.1 *Ulloriaq Scheduling Model*

The goal of this model is to generate a trace file that can be used in part 2 of the tool-chain, as seen in Figure 4.1. The generated trace contains the contact plan for ten satellites in a train-formation, modelled as a delay tolerant network, with each satellite being a node in the network, using a specific cost-function implemented in the model.

Each satellite in the Ulloriaq constellation are represented in the model as an instance of the *Satellite* template presented in section 4.1.4.

The following sections contain a high-level description of the behavior of the model, the different components and the necessary preliminaries are given, followed by an in-depth description of each of the templates that make up the model.

4.1.1 Model Architecture

The architecture of the *Ulloriaq Scheduling Model* consists of 10 satellites, each with their own internal state, as well as a *Synchronizer*, that synchronizes actions between the satellites. An illustration of this is shown on Figure 4.2.

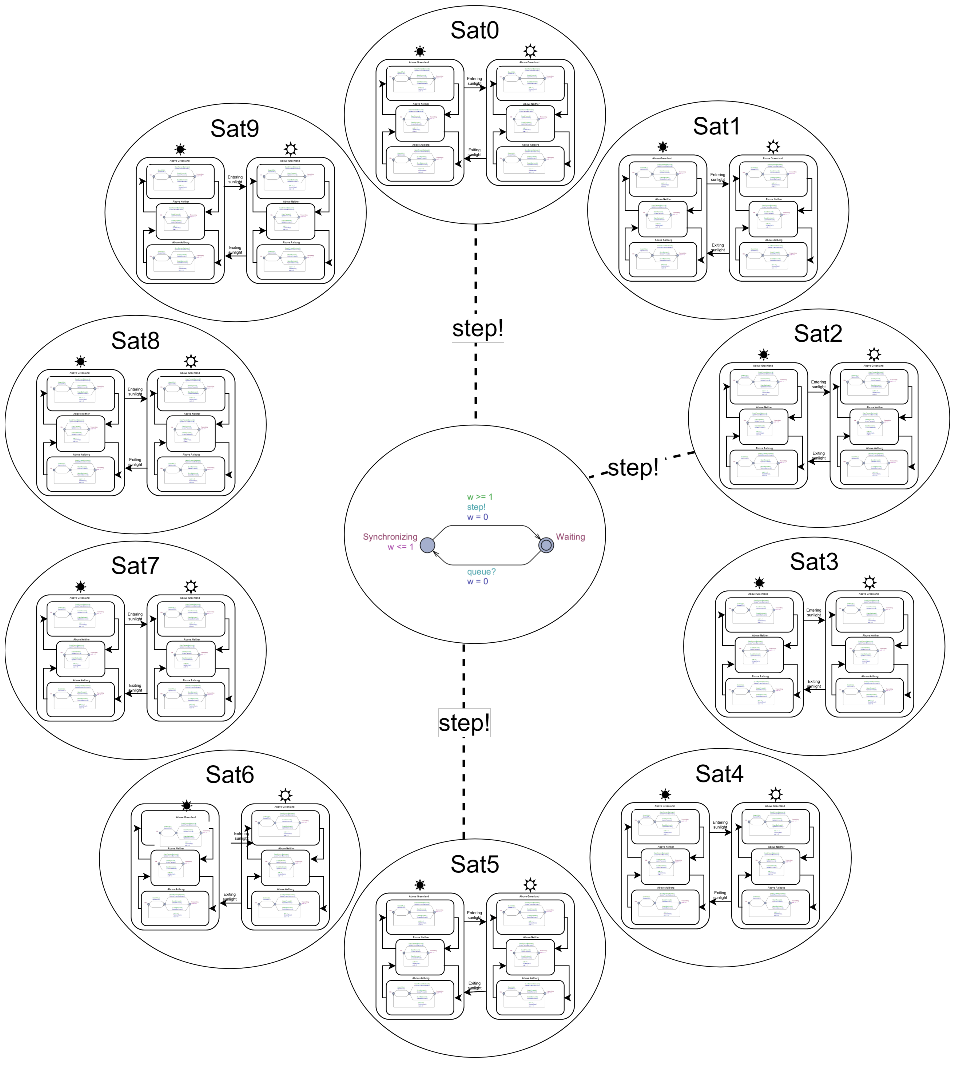


Figure 4.2: Architecture of the model

Each circle surrounding the central circle on Figure 4.2 symbolizes a satellite and its behavior, and the central circle symbolizes the *Synchronizer* part of the model. Whenever a satellite takes an action, as described in Section 4.1 and on Figure 4.3, a connection is made to the *Synchronizer*. Satellites 0, 2 and 5 are at the time depicted performing some action, illustrated by the lines going from these satellites to the *Synchronizer*.

4.1.2 High-level Model Behaviour

While orbiting the earth, each satellite moves through three different types of zones. These zones are Sunlight, Aalborg, and Greenland. The zones Aalborg and Greenland represent at which times the satellites can connect to Aalborg, or collect data from Greenland. Only satellites equipped with an HSL transponder can connect to Aalborg. Any satellite can collect data from Greenland. The Aalborg and Greenland zone-types cannot overlap for a satellite, as they do not inhabit any of the same physical locations. The Sunlight zone-type represents whether a given satellite is exposed to the sun, or in eclipse. This information is used to evaluate whether the battery should be recharging. A Sunlight zone can overlap any of the two other zone-types.

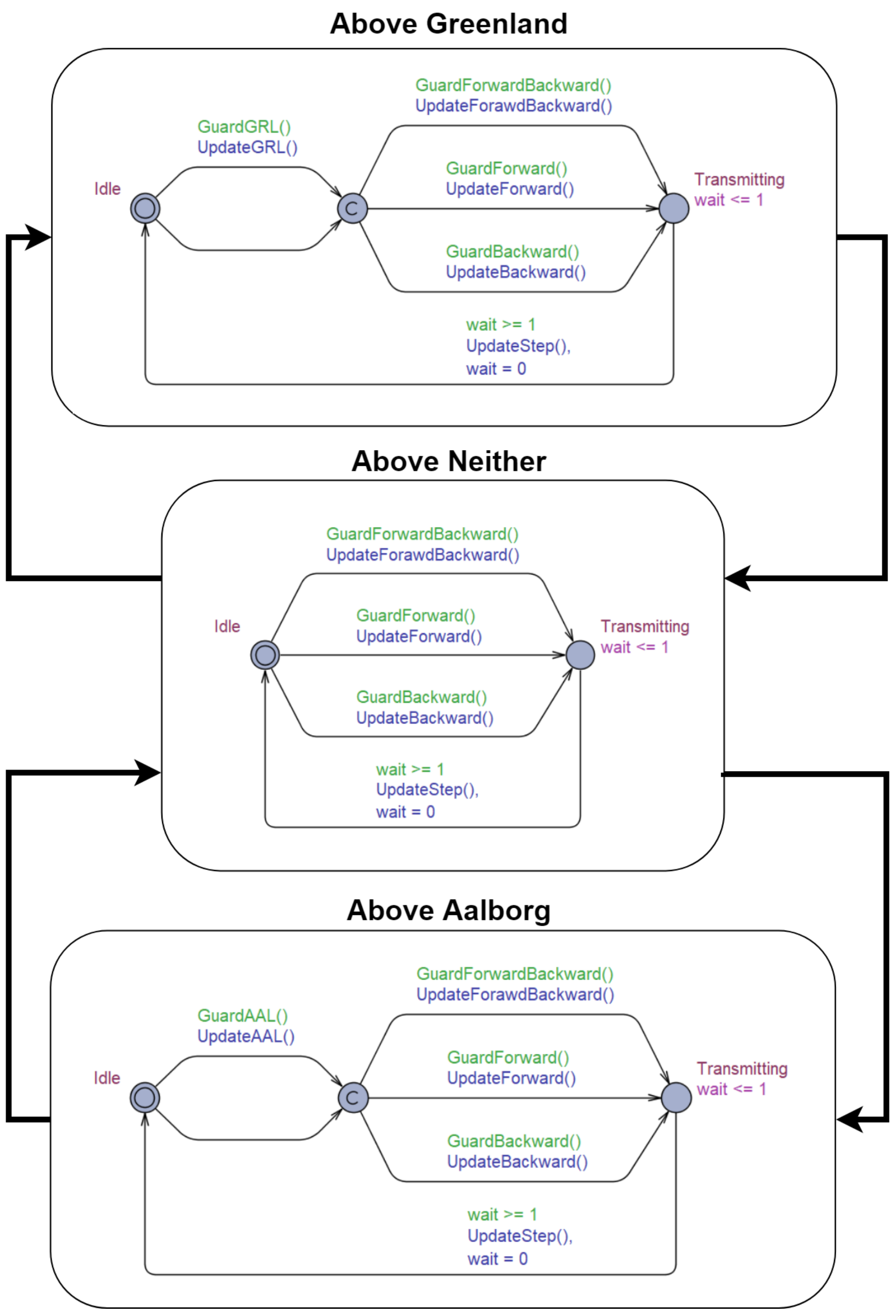


Figure 4.3: Abstract satellite behavior

The zone behavior of Aalborg and Greenland is illustrated in Figure 4.3. Each satellite moves through 3 different states, as depicted on 4.3, with regard to the Aalborg and Greenland zones: Above Greenland, Above Aalborg, and Above Neither. The TA in each box illustrates what the satellite can do when in that state. A satellite can always transmit data to either its forward or backward neighbor or both, and as such, this can happen in any of the three states. When the satellite is in an Aalborg zone, it can transmit data, and when it is in a Greenland zone, it can collect data.

While moving through these zones, two kinds of things can happen in the model: **Actions** and **Events**. An action is everything that happens inside one of the boxes in Figure 4.3. Here the actions available in the current state of the satellite are enabled. Events are represented by the bold arrows going to and from the boxes. These are controlled by the environment and are dependant on the orbit of each satellite. Each event indicates that the satellite enters or leaves an Aalborg or Greenland zones. These zones are not static, meaning that they are different for each orbit and satellite. This is represented by two lists for each zone, one each for start and end times, for each satellite. An example of a satellite begin and end times for the Aalborg and Greenland zones can be found in tables 4.1 and 4.2. These numbers were supplied by the authors of [12].

Begin	315	410	873	966	1063	1746	1841	2305	2397	2494
End	323	416	878	973	1066	1754	1848	2309	2405	2498

Table 4.1: Aalborg zone data example

Begin	19	118	214	309	404	499	594	688	783	877
End	25	121	219	316	414	513	607	700	794	888

Table 4.2: Greenland zone data example

A similar subset of the data for when a satellite is entering and leaving sunlight can be seen in Table 4.3. For this example, the satellite would be in sunlight from time 28 till 96 and then again from 123 till 192. Between these two ranges, the satellite is eclipsed and is not receiving any sunlight.

In general, for all three types of zones (Aalborg, Greenland, and Sunlight), if time progresses into, or out of any of the listed zones, this results in an event being triggered, and makes the satellite change its state accordingly, dependent on which type of zone triggered the event. The time points listed refer to the absolute time since beginning of the contact plan generation. The behavior of a satellite moving in and out of sunlight is illustrated in Figure 4.4. Each side of the figure represents the actions and events from Figure 4.3 happening in either sunlight or eclipse.

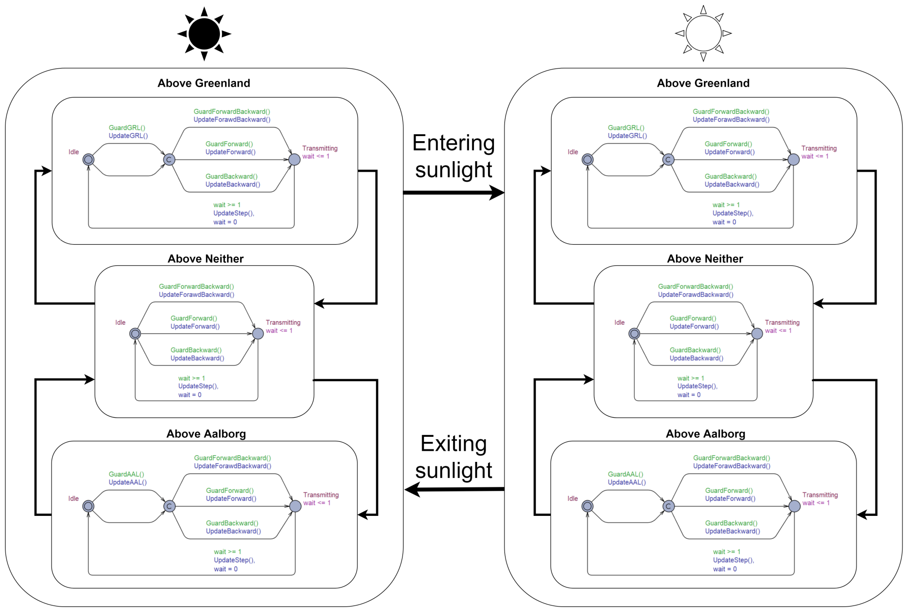


Figure 4.4: Abstract satellite sun behaviour

Begin	28	123	219	314	410	505	600	696	791
End	96	192	287	382	478	573	669	764	859

Table 4.3: Sunlight zone data example

4.1.3 Heuristics

A number of heuristics has been implemented in the model to optimize it in different ways. These work by disallowing or forcing certain kinds of behaviour, that has been determined to be detrimental to the generation of valid contact plans, or the performance of the model in general. An example of this could be allowing the model to perform some looping action, that has no effect on the final result, except wasting battery. This sort of behaviour is not desirable, and as such a heuristic should be implemented to prevent it. A second type of heuristic is for preventing sub-optimal solutions - if these types of solutions are presented in the first place, no computing power is wasted on searching for optimal solutions among them.

Heuristic 1. No ping-pong data transfer Sending data back and forth between two satellites will never be an optimal solution, since it wastes battery. Because of this satellites are not allowed to transfer data to a satellite that it just received data from or that it is currently receiving data from.

Heuristic 2. Data must move towards HSL satellites As a consequence of heuristic 1, if a satellite receives data from both its neighbors it will not be able to do anything and the data will as a result just pool together in that satellite. To avoid this behavior, data must always move towards an HSL-enabled satellite.

Heuristic 3. No batteries can go below the minimum limit If any of the satellite batteries ever go under the defined minimum limit, the generated contact plan must be discarded.

Heuristic 4. HSL satellites must transmit all data to Aalborg when possible The goal of the mission is to transfer data to Aalborg. Because of this and because it is possible for the HSL-enabled satellites to transfer all their data at once, they must always transfer all data to Aalborg when transmitting.

Heuristic 5. HSL satellites cannot send data to other satellites This heuristic might disallow a potential optimal solution. This was however necessary to implement to reduce the size of state space and make it possible to generate a contact plan.

Heuristic 6. A satellite cannot take an action if doing so would deplete the battery at the end of the period Each satellite is only charged or discharged, by the sun and background load, when an event occurs. This results in it being possible for a satellite to perform a number of actions while no Sunlight zone changes happen, possibly depleting the battery when a new zone triggers an update of the battery. Thus, this heuristic prevents this from happening, by queuing up the amount of battery charge each action takes and calculating whether or not an additional action would deplete the battery at the next update to the battery charge.

4.1.4 Model description

As explained in Section 4.1.1, the model consists of two elements, a number of satellites and a *Synchronizer*. The implementation of the model has a template for each of these elements, namely *Satellite*, representing each satellite in the model, and *Synchronizer*, which is responsible for synchronizing satellite communication.

Constants

Recalling Table 2.1 from Section 2.1, containing the constants for the satellites used in the Ulloriaq case. These are defined as floating point numbers, which cannot be used in UP-

PAAL CORA. It was therefore necessary to round these to integers. They have been rounded up or down, according to which direction provides the most pessimistic contact plans. Furthermore, due to technical limitations, the time unit used in the *Ulloriaq Scheduling Model* does not represent seconds, but minutes. This is further explained in Section 4.1.6. Because of this, many of the constants had to be scaled to match this new granularity. The values used in the model can be found on Figure 4.4.

satCount	10	The number of satellites
initCharge	3694	The initial charge of the battery
maxCharge	4617	The maximum charge of the battery
minCharge	2770	The minimum charge of the battery
backgroundLoad	5	The background load on the battery
transmitLoad	14	The load required to do a transmission
rechargeRate	15	The rate at which the battery is recharged
bufferSize	10.000	The size of the satellite buffer
interSatSpeed	1	The speed at which data is transferred between satellites
greenlandSpeed	1	The speed at which data is collected from Greenland
aalborgSpeed	100	The speed at which data is transferred to Aalborg
goalData	313	The total amount of data fed to the satellites
goalTime	2880	The length in time of the contact plan being generated

Table 4.4: Constant values used in the model

There are two types of battery values. The first type indicates the state of the battery (initCharge, maxCharge and minCharge) and the second type indicating a load on the battery (backgroundLoad, transmitLoad and rechargeRate). Instead of scaling the different loads up, we decided to scale the battery capacities down. This was done to keep the value within the range of the built-in integer. The reason for this is also described in Section 4.1.6.

All constants related to data transfer (bufferSize, interSatSpeed, greenlandSpeed and aalborgSpeed) has been divided by 10. This has been done since all the original values were divisible by 10 and the smallest value was 10. It is desirable to use as small numbers as possible to optimize state-space. Aalborg has been scaled down even further to match the satellites buffer size.

The two final constants are associated with the goal state being queried for. Here *goalData* is the amount of data being fed to the model. This number differs from the original value since it had to be scaled in a few different ways to make it fit with the rest of the constants used in the model. Firstly, the original value (187.5) are in megabytes (MB), which had to be converted to kilobytes (KB) to fit with the data transfer speeds, this changed the value to 187,500. As explained above, all data values were divided by 10; this is also the case for this value, which brings it to 18,750. Finally, since the model works in minutes and not seconds, the value had to be divided by 60, which brought it to down to 312.5. This was then rounded to the 313 which is used for querying the model. The goal time is the contact plan length from Table 2.1, in seconds, divided by 60 to get to scale it to minutes.

Satellite template

The *Satellite* template in its entirety, can be found on Figure 4.5. For the sake of explanation, the template has been segmented into three parts, namely: *idle part*, as seen on Figure 4.6 and colored green on 4.5, *event handler*, as seen on Figure 4.7 and colored blue on 4.5 and *action handler*, as seen on Figure 4.9, colored red on 4.5. The functions used in the Satellite template can be seen in Table 4.5 and the variables and constants in Table 4.6.

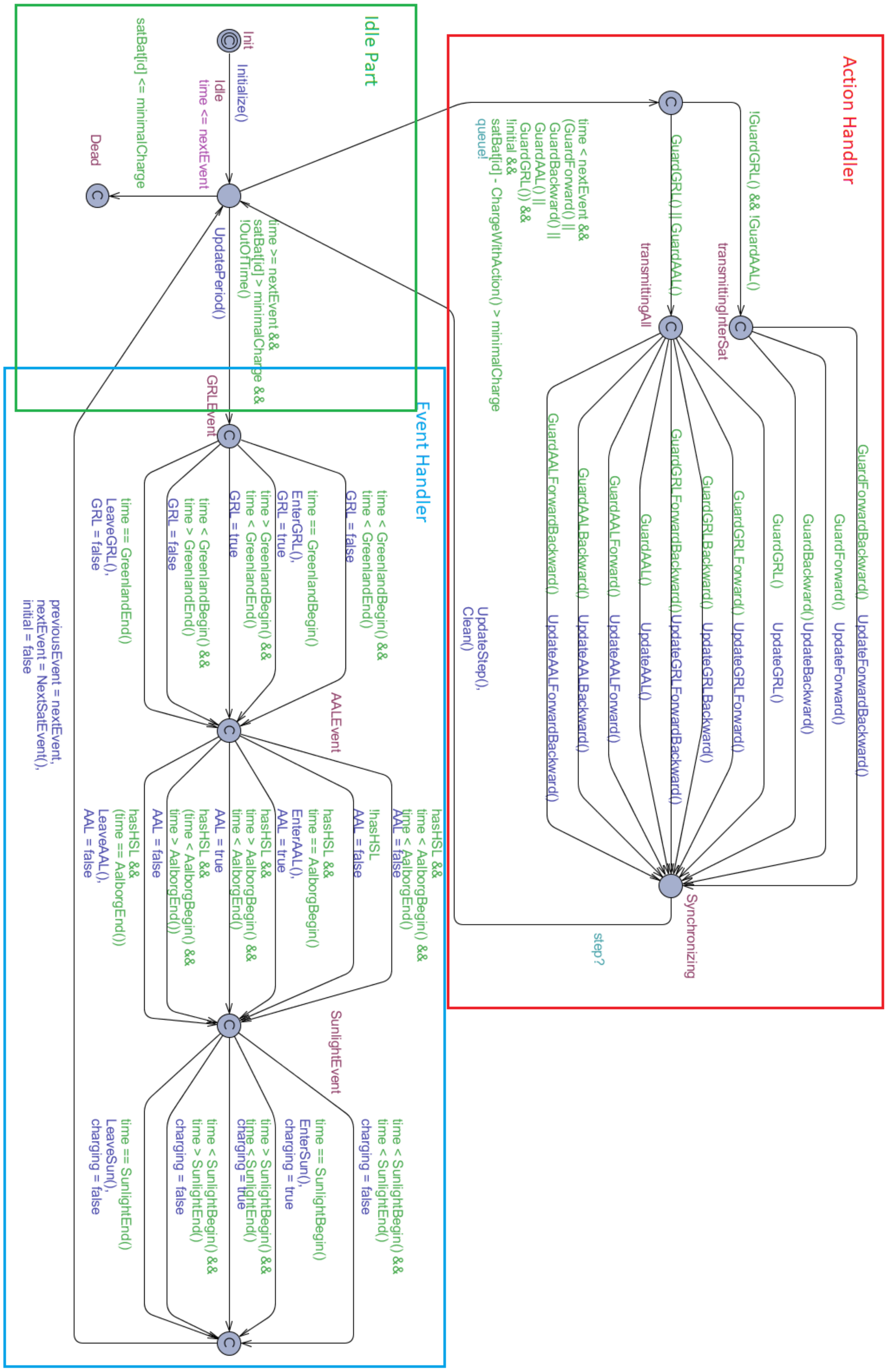

Action Handler
Idle Part
Event Handler
Init
Initialize()
Idle
time <= nextEvent
satBat[id] <= minimalCharge
Dead
time >= nextEvent && satBat[id] > minimalCharge && !OutOfTime()
UpdatePeriod()
time < nextEvent && (GuardForward() || GuardBackward() || GuardAAL() || GuardGRL()) && !initial && satBat[id] - ChargeWithAction() > minimalCharge
queue!
!GuardGRL() && !GuardAAL()
GuardGRL() || GuardAAL()
transmittingInterSat
transmittingAll
GuardForwardBackward() UpdateForwardBackward()
GuardForward() UpdateForward()
GuardBackward() UpdateBackward()
GuardGRL() UpdateGRL()
GuardGRLForward() UpdateGRLForward()
GuardGRLBackward() UpdateGRLBackward()
GuardGRLForwardBackward() UpdateGRLForwardBackward()
GuardAAL() UpdateAAL()
GuardAALForward() UpdateAALForward()
GuardAALBackward() UpdateAALBackward()
GuardAALForwardBackward() UpdateAALForwardBackward()
UpdateStep(), Clean()
Synchronizing
step?
GRLEvent
time < GreenlandBegin() && time < GreenlandEnd() GRL = false
time == GreenlandBegin() EnterGRL(), GRL = true
time > GreenlandBegin() && time < GreenlandEnd() GRL = true
time < GreenlandBegin() && time > GreenlandEnd() GRL = false
time == GreenlandEnd() LeaveGRL(), GRL = false
AALEvent
hasHSL && time < AalborgBegin() && time < AalborgEnd() AAL = false
!hasHSL AAL = false
hasHSL && time == AalborgBegin() EnterAAL(), AAL = true
hasHSL && time > AalborgBegin() && time < AalborgEnd() AAL = true
hasHSL && (time < AalborgBegin() && time > AalborgEnd()) AAL = false
hasHSL && (time == AalborgEnd()) LeaveAAL(), AAL = false
SunlightEvent
time < SunlightBegin() && time < SunlightEnd() charging = false
time == SunlightBegin() EnterSun(), charging = true
time > SunlightBegin() && time < SunlightEnd() charging = true
time < SunlightBegin() && time > SunlightEnd() charging = false
time == SunlightEnd() LeaveSun(), charging = false
previousEvent = nextEvent, nextEvent = NextSatEvent(), initial = false

Figure 4.5: Satellite template - *Ulloriaq Scheduling Model*

void Initialize()	Initializes the battery
bool GuardForward() bool GuardBackward() bool GuardForwardBackward()	Returns whether or not the satellite can connects to a neighbor
bool GuardGRL() bool GuardGRLForward() bool GuardGRLBackward() bool GuardGRLForwardBackward()	Returns whether or nor the satellite can collect data from Greenland, and possibly connect to a neighbor
bool GuardAAL() bool GuardAALForward() bool GuardAALBackward() bool GuardAALForwardBackward()	Returns whether or nor the satellite can transmit data to Aalborg, and possibly connect to a neighbor
void UpdateForward() void UpdateBackward() void UpdateForwardBackward()	Transmits data to neighboring satellite and discharges battery
void UpdateGRL() void UpdateGRLForward() void UpdateGRLBackward() void UpdateGRLForwardBackward()	Collects data from Greenland and discharges battery
void UpdateAAL() void UpdateAALForward() void UpdateAALBackward() void UpdateAALForwardBackward()	Transmits data to Aalborg and discharges battery
void UpdatePeriod()	Charges or discharges the battery after leaving the previous zone
void UpdateStep()	Charges or discharges the battery for one timeunit
int GreenlandBegin() int AalborgBegin() int SunlightBegin()	If in a zone: Returns when the current zone began If not in a zone: Returns when the next zone begins
int GreenlandEnd() int AalborgEnd() int SunlightEnd()	If in a zone: Returns when the current zone ends If not in a zone: Returns when the previous zone ended
void EnterGRL() void EnterAAL() void EnterSun()	Advances the pointer for the zones *begin* array
void LeaveGRL() void LeaveAAL() void LeaveSun()	Advances the pointer for the zones *end* array
int ChargeWithAction()	Returns the amount the battery will be charged or discharged at the next event plus the energy required to take an action

Table 4.5: Functions used in Satellite template

User-defined Types	
s_id	A custom type ranging from 0 up to the total amount of satellites
Int	A custom integer type, which is able to contain a larger span of integers
Constants	
s_id id	The satellites Id
Int minimalCharge	The minimal charge allowed
Variables	
bool initial	Indicates if it is the initial run through the Event Handler
int nextEvent	When the next event will occur
int previousEvent	When the previous event occurred
bool GRL bool AAL bool charging	Indicates if the satellite is in the zone
Int satBat[id]	Current state of charge
Clocks	
clock time	The global clock
Channels	
chan queue!	Used to queue with the synchronizer
chan step?	Used to synchronize with other satellites taking an action

Table 4.6: Attributes used in Satellite template

Each instance of the Satellite template takes two parameters: an ID (*id*) and a Boolean describing whether or not the satellite is equipped with a high-speed downlink (*hasHSL*). Both variables can be found in Table 4.6. The satellite has nine locations, where two are normal locations, and the rest are committed. The committed locations can be thought of as intermediate locations, assisting in updating values, and the two normal locations are the main locations, where actions and events take place. The initial location, `Init`, is one of these committed locations. The singular purpose of this location and its associated edge, is to initialize the satellite battery. After initializing, the satellite proceeds to the `Idle` location. From here the satellite is forced to go to the `Dead` location, if its battery is depleted below the minimum allowed charge, delay until the time of the next event or take a number of actions, each taking one time unit. When the satellite reaches the time for an event, it will then enter the *event handler* part of the template. At any time before the event, the satellite can take an action, by going to the *action handler* part of the template.

The battery has been implemented using the linear battery model, as described in Section 2.6.1. Battery usage is tracked in two different ways in the model, namely during an action and during an event. Taking an action has a cost in terms of battery charge, which is subtracted from the satellites battery level. Furthermore, the charge needed to delay one time unit is spent. This charge is calculated from the background load and whether or not the satellites are eclipsed or not. The function used for the looks as follows:

```
void UpdateStep(){
    recievedForward = connections[GetForwardNeighbor(id)][id];
    recievedBackward= connections[GetBackwardNeighbor(id)][id];

    actionsTaken++;

    if(satBat[id] + CurrentLoad() > maxCharge)
        satBat[id] = maxCharge;
    else
        satBat[id] += CurrentLoad();
}
```

The first two lines of the function above are associated with Heuristic 1 and is not related to the battery. The variable *actionsTaken* tracks the number of actions a satellite has taken. This is used during the battery update for the Event Handler. The if-statement makes sure that the battery charge never is greater than the maximum allowed amount. If adding the current load to the battery does not exceed the maximum, then it is added to the battery (this number can be negative), if not, then the battery is update to its maximum value. The function used to get the current load is defined as follows:

```
Int CurrentLoad(){
    Int total = 0;

    total += -backgroundLoad;
    if(charging)
        total += rechargeRate;
    return total;
}
```

CurrentLoad() returns the current load on the battery. This value is negative if the battery is being charged and positive if it is being recharged. The background load is always added to the total and if the satellite is currently charging, then the recharge rate is also added.

When reaching an event, the time since the previous event is calculated and the battery charge is changed according to this subtracted by the number of actions taken by the satellite since the previous event. This is done to take the charge spent during an action into account. This is done in the function *UpdatePeriod()*, which looks as follows:

```
void UpdatePeriod(){
    int scale = (nextEvent - previousEvent) - actionsTaken;
    actionsTaken = 0;
    if(satBat[id] + CurrentCharge() * scale > maxCharge)
        satBat[id] = maxCharge;
    else
        satBat[id] += CurrentCharge() * scale;
}
```

Starting from the top of the function. On the first line, the scale of the event is calculated by finding the time since the previous event and subtracting the actions taken *actionsTaken*, which corresponds to the time units spent since the last event update where an action did not take place. When this is calculated, the *actionsTaken* variable is reset for the next event period. The final if-statement updates the battery with the charge used in the period. This is done by scaling the current charge, calculated in *CurrentCharge()*, as explained above, by the *scale* variable.

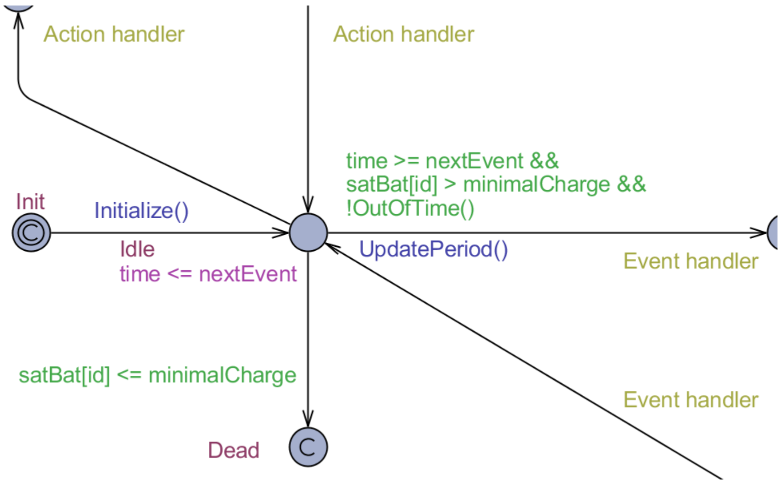


Figure 4.6: Idle part of '*Ulloriaq Scheduling Model*'

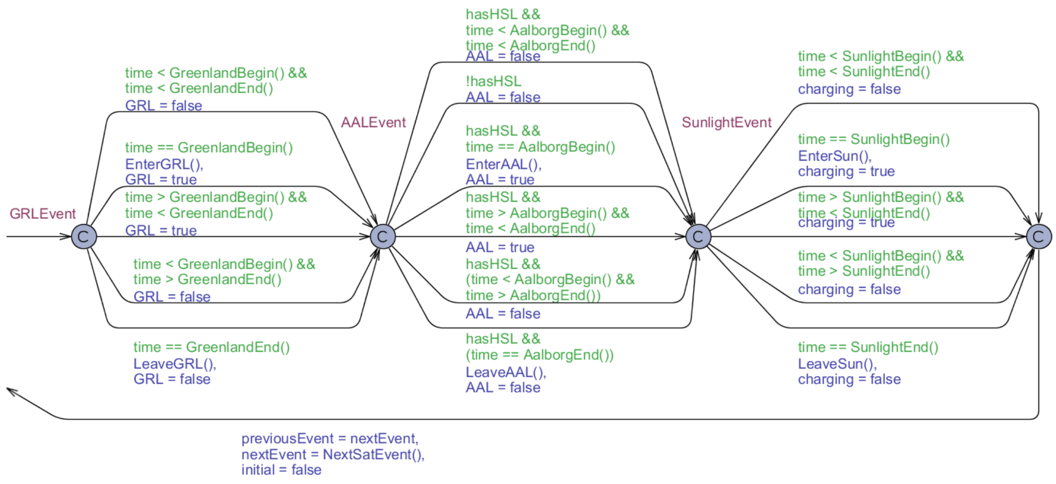

GRLEvent
AALEvent
SunlightEvent
time < GreenlandBegin() && time < GreenlandEnd() GRL = false
time == GreenlandBegin() EnterGRL(), GRL = true
time > GreenlandBegin() && time < GreenlandEnd() GRL = true
time < GreenlandBegin() && time > GreenlandEnd() GRL = false
time == GreenlandEnd() LeaveGRL(), GRL = false
hasHSL && time < AalborgBegin() && time < AalborgEnd() AAL = false
!hasHSL AAL = false
hasHSL && time == AalborgBegin() EnterAAL(), AAL = true
hasHSL && time > AalborgBegin() && time < AalborgEnd() AAL = true
hasHSL && (time < AalborgBegin() && time > AalborgEnd()) AAL = false
hasHSL && (time == AalborgEnd()) LeaveAAL(), AAL = false
time < SunlightBegin() && time < SunlightEnd() charging = false
time == SunlightBegin() EnterSun(), charging = true
time > SunlightBegin() && time < SunlightEnd() charging = true
time < SunlightBegin() && time > SunlightEnd() charging = false
time == SunlightEnd() LeaveSun(), charging = false
previousEvent = nextEvent, nextEvent = NextSatEvent(), initial = false

Figure 4.7: Event handler part of '*Ulloriaq Scheduling Model*'

The *event handler* part, as seen on Figure 4.7, is responsible for updating each satellite whenever it reaches the time for the next event. It is accessed by taking the rightmost transition from the `Idle` location. Due to the invariant `time <= nextEvent` on the `Idle` location and the guard `time >= nextEvent` on the edge going to `GRLEvent`, the satellite is forced to take this transition whenever the clock *time* reaches the next event.

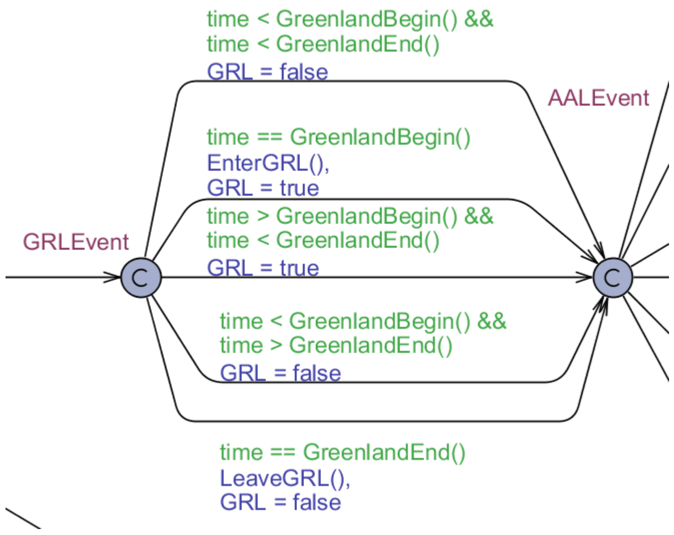

GRLEvent
AALEvent
time < GreenlandBegin() && time < GreenlandEnd() GRL = false
time == GreenlandBegin() EnterGRL(), GRL = true
time > GreenlandBegin() && time < GreenlandEnd() GRL = true
time < GreenlandBegin() && time > GreenlandEnd() GRL = false
time == GreenlandEnd() LeaveGRL(), GRL = false

Figure 4.8: Greenland part of the event handler

When an event is reached, the satellite in question checks whether or not it is entering one of the different types of zones (Greenland, Aalborg and sunlight). This is done by taking one of five edges (six for Aalborg) for each type. An example, using the Greenland part can be seen in Figure 4.8. The two edges with guards `time == GreenlandBegin()` and `time == GreenlandEnd()`, respectively indicates that the satellite has entered or left the zone. The top edge is used when the satellite starts outside of a zone, in which case both the begin and end time for the satellite will be greater then the time, unit it reaches the first zone. The two remaining edges is taken when the satellite is in- or outside of the current zone respectively. This is repeated for each of the three types of zones, with Aalborg having a sixth edge used by satellites that are not equipped with the HSL transponder. The last thing that happens before returning to `Idle` is that the next event for the satellite is found and the time for the event that just occurred is saved to be used to update the battery when the next event occurs.

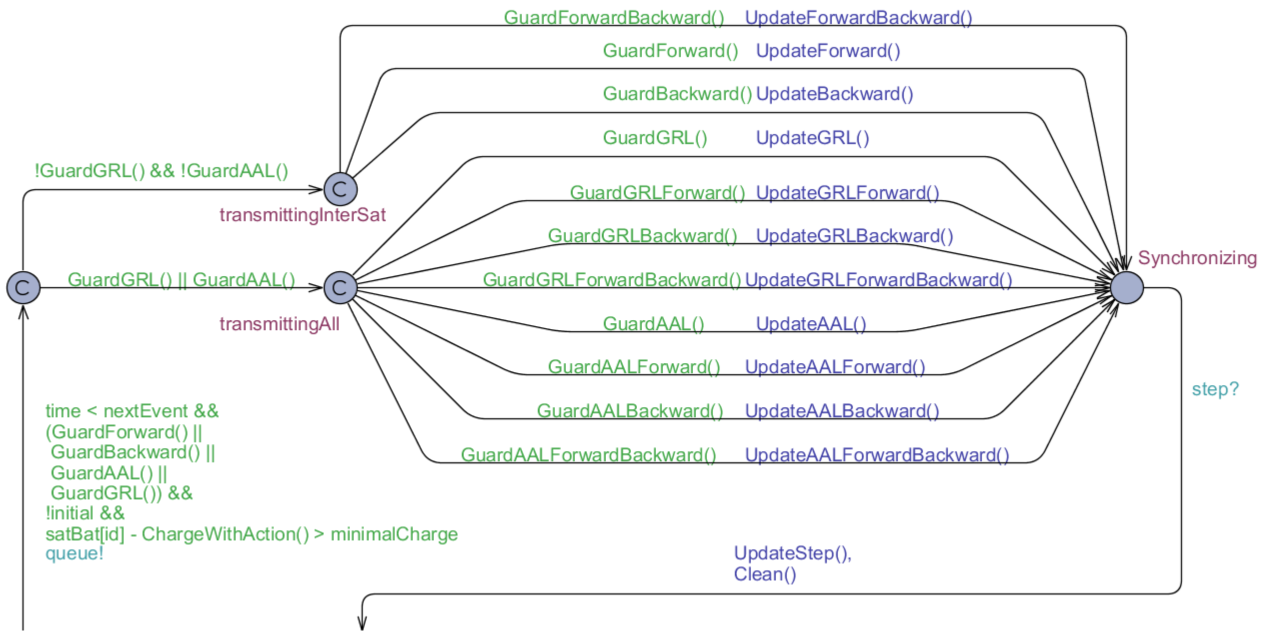

GuardForwardBackward() UpdateForwardBackward()
GuardForward() UpdateForward()
GuardBackward() UpdateBackward()
GuardGRL() UpdateGRL()
!GuardGRL() && !GuardAAL()
C
transmittingInterSat
GuardGRLForward() UpdateGRLForward()
GuardGRLBackward() UpdateGRLBackward()
Synchronizing
C
GuardGRL() || GuardAAL()
C
GuardGRLForwardBackward() UpdateGRLForwardBackward()
transmittingAll
GuardAAL() UpdateAAL()
GuardAALForward() UpdateAALForward()
step?
GuardAALBackward() UpdateAALBackward()
GuardAALForwardBackward() UpdateAALForwardBackward()
time < nextEvent && (GuardForward() || GuardBackward() || GuardAAL() || GuardGRL()) && !initial && satBat[id] - ChargeWithAction() > minimalCharge
queue!
UpdateStep(), Clean()

Figure 4.9: Action handler part of '*Ulloriaq Scheduling Model*'

The *action handler* part, as seen on Figure 4.9, is responsible for taking actions during the satellites orbit. An action can occur at anytime while the satellite is waiting for the next event. The 2nd of the two normal locations are found in this part, named `Synchronizing`, this location is used for synchronizing the actions of all satellites performing an action at a given time and delaying the satellites for one time unit. When taking an action, the battery and buffers of the involved satellites are updated during this synchronization. Each action has two associated functions, a guard function returning a Boolean and an update function. For the edges containing multiple actions, the guard and update functions of the individual actions have been accumulated into a single function each. The edge going from `Idle` to `transmitting` is enabled as long as one of these guard functions evaluates to true. When a satellite takes an action, it broadcasts on the `queue!` channel. This makes the *Synchronizer* ready to handle the delay, this is further described in the following section. When no more satellites can or will take an action, the *Synchronizer* broadcasts in the `step?` channel, which makes all the queued up satellites return to the `Idle` location. Doing so updates each participating satellite through the function called `UpdateStep()`. This discharges the battery for the cost of the action plus the current background load for one time unit.

Synchronizer template

The *Synchronizer* as seen on Figure 4.10 is responsible for synchronizing satellites and their actions. The template consists of three locations: `Waiting`, the initial location, `Synchronizing` and finally `Done`. The template changes from `Waiting` when a satellite broadcasts on the `queue?` channel. The template can stay in the `Synchronizing` location for exactly one time unit, at which point it broadcasts on the `step!` channel, which makes the satellites take one step concurrently. The final location `Done` is used as a goal state. This is further explained in Section 4.1.4. The *Synchronizer* has a clock w. According to the invariant `w <= 1` on `Synchronizing` the *Synchronizer* cannot delay longer than 1 time unit. Furthermore, the guard `w >= 1` only allows the step to be taken after 1 time unit. The combination of this invariant and guard forces the *Synchronizer* to wait exactly 1 time unit.

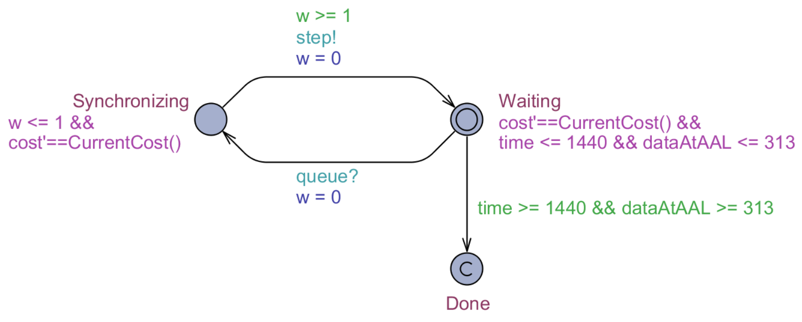


Figure 4.10: Synchronizer template

Heuristic implementation

All of the heuristics, from Section 4.1.3, except *Heuristic 4* are implemented as part of guards on the different edges in the model(GuardForward, GuardBackward, GuardAAL, GuardGRL). This is done to disallow the behavior described in each heuristic. An example of such a guard is the function *GuardForward(), which looks as follows*:

```
bool GuardForward(){ return satData[id] > 0 && !RecievedForward() &&
    AllowedForward() && !hasHSL; }
```

From this example, *!RecievedForward()* is associated with *Heuristic 1* and disallows ping-pong behavior. *AllowedForward()* is associated with *Heuristic 2* and makes sure that all satellites between the current satellite and an HSL-enabled satellite moves in the same direction to avoid pooling of data on non-HSL satellites. *!hasHSL* disallows HSL satellite from sending data to other satellites, in accordance with *Heuristic 5*.

Heuristic 4 is implemented by always emptying the HSL satellites buffer to Aalborg. This is a side effect of the transfer rate from an HSL satellite to Aalborg being greater than the size of the satellites data buffer.

Model goal and query

As described in Chapter 2.1, the goal of the mission is to collect a set amount of data within 48 hours. The authors of [12] were successful in scheduling a constellation of ten satellites for 48 hours. During this period the satellites were fed 187.5 MB of data to be transmitted from Greenland, through the network of satellites and down to the ground station in Aalborg. Using this as the goal for the model, there are two parts to consider, the time window to be scheduled and the amount of data to be transmitted. The amount of data is defined in the model as a global constant called *goal*, and the data available for the satellites to collect is then limited to this goal. The maximum length of a contact plan, generated by the model, is dictated by the arrays used for determining the Aalborg, Greenland, and Sun zones, since the model would not know what to do, further ahead in time than the latest time in said arrays. Within the time possible with the arrays, the time of the contact plan is defined when making the query in UPPAAL CORA. An example of a query used are $E <> time >= 2880 \ \&\& \ dataAtAAL == 313$, or in other words; find a trace where, at the end, the time is equal to 2880 and the amount of data transmitted to Aalborg is 313. The numbers 2880 and 313 are derived from Ulloriaq, and is described further in Chapter 5. In addition to the query explained in this section, an additional location has been added to the *Synchronizer* to function as a sort of goal-state. This is further explained in Section 5.1.2.

Traces

A trace in the *Ulloriaq Scheduling Model* represents a long series of transitions taken by each satellite as well as the transitions taken by the *Synchronizer* during a simulation of the system.

A trace in this system includes a line for each transition taken during the simulation. This is best demonstrated by looking at some lines from the model-independent version of the trace (the product of phase 2 of the tool-chain, shown in Figure 4.1). The version depicted has parts of the state cut off, for readability:

1. State: w.Waiting sat0.Idle sat1.Init sat2.Idle sat3.Idle sat4.Idle sat5.Init
2. Transition: sat5.Init -> sat5.Idle 1; 0; Initialize();
3. State: w.Waiting sat0.Idle sat1.Init sat2.Idle sat3.Idle sat4.Idle sat5.Idle
4. Transition: sat1.Init -> sat1.Idle 1; 0; Initialize();
5. State: w.Waiting sat0.Idle sat1.Idle sat2.Idle sat3.Idle sat4.Idle sat5.Idle
6. Transition: sat2.Idle -> sat2.GrlEvent time >= nextEvent && satBat[id] > minimalCharge
7. State: w.Waiting sat0.Idle sat1.Idle sat2.GrlEvent sat3.Idle sat4.Idle sat5.Idle

In this trace, the first two transitions(lines 2 and 4) move satellites 5 and 1 out of their idle state and runs an initialization function to set up different variables before starting the simulation. The last transition(line 6) is triggered by satellite 2 moving into a Greenland zone and starts the process of updating variables according to which zones the satellite now resides in.

4.1.5 Cost

In the *Ulloriaq Scheduling Model*, the cost was defined to be as close to the cost used in the objective function found in [12], to enable a better comparison of results. The cost update function resides on the *Synchronizer* template, found on Figure 4.10. Every time one or more satellites take an action, and synchronize their steps, the cost is updated. The cost function is as follows:

```
Int CurrentCost(){

    Int result;
    result += TotalMissingCharge();
    result += (goal - dataAtAAL - TotalSatData());
    return (result >= 0 ? result : 0);
}
```

In the above function "TotalMissingCharge()" adds a cost for however much battery is missing at this time, thus accumulating a cost based on how much battery is used. This helps provide traces that optimize the battery usage of the satellites. The second part of the function increases cost based on how much data still needs to be delivered to the ground-station in Aalborg. This helps provide traces that optimize fast delivery of data to said ground station.

4.1.6 Time granularity

The initial versions of the model was created in such a way that 1 time-unit in the model was equal to 1 real second. With this granularity, the goal of scheduling 48 hours, would take

172.800 time units. The standard integer in UPPAAL CORA is 2 bytes, which fits a value of -32.768 to 32.767. Because of this, to do the full 48 hours with a granularity of 1 time unit equals 1 second, it would be necessary to implement a custom integer that can support a bigger range. This however made it impossible to generate any meaningful data, since the way UPPAAL CORA is implemented, using these bigger integers takes up more memory than are proportional to the size difference, which resulted in the command-line version running out of memory before reaching a usable trace. To solve this, the final versions of the model uses a more coarse granularity, equalling 1 time-unit to 1 real minute.

4.2 Contact Plan Extractor

The Ulloriaq Contact Plan Extractor is a custom made tool for parsing a trace generated by UPPAAL CORA, into a contact plan that can be further verified and used as schedules for satellites. Phase 2 on Figure 4.1 is a pre-processing step before the Contact Plan Extractor, parsing the trace produced by UPPAAL CORA into a model-independent trace, after which the Ulloriaq Contact Plan Extractor can then generate a contact plan.

The overall goal of the Contact Plan Extractor is to produce either a textual representation of the contacts available to satellites during operation or a visual one, similar to the ones presented in [12]. The following sections contain a description of phases 2 and 3 of the Ulloriaq toolchain, as described in Figure 4.1.

4.2.1 UTAP

UTAP (UPPAAL Timed Automata Parser Library) [1] is a library for parsing traces produced by different versions of UPPAAL. UPPAAL CORA produces traces in a format called .XTR. This format is intrinsically linked to the model from which it was generated, and thus UTAP is needed to parse it into something independently-readable. UTAP comes with a built-in utility called "tracer," which, using the trace and a converted version of the model it was generated from, generates this model-independent-human-readable version of the trace. The model needs to be converted from the standard .XML format into a .if(intermediate-format) type. The verifier from UPPAAL CORA performs this task. The format of the parsed trace is .TXT.

4.2.2 Ulloriaq Contact Plan Extractor

The Ulloriaq Contact Plan Extractor is depicted on phase 3 on Figure 4.1. The extractor takes a .TXT file generated by UTAP, and translates it into a textual(.cnp) or visual contact plan, represented by a Gantt-chart. It is written in C# [15], using various parts of the .NET framework. A trace being parsed by the Extractor goes through several steps before the final trace is produced. The first step is simply reading every state and the following transition found in the model-independent trace-file. This file is a plain .txt file. An example from such a file can be seen on Listing 4.1.

```
State: w.Waiting sat0.Init sat1.Init sat2.Init sat3.Init ...
Transition: sat5.Init -> sat5.Idle {1; 0; Initialize();}
```

Listing 4.1: Example line of a trace from UTAP

The model-independent trace-file alternates between printing states and transitions. A state of the system being printed includes the location of every template instance, every variable defined in code, the current time, and the current cost. A transition being printed includes what edges are being taken, which functions are being called, and which variables are being updated. When this is all read into the Ulloriaq Contact Plan Extractor, it is all put into corresponding classes, to facilitate different purposes.

The first step after reading the whole trace-file is to stabilize the time(zones). The trace-file produced by UTAP has certain transitions after which the time is either fully reset where it should not be, or is in a zone without a lower bound where it should be included. Both happen at different points during the trace. In order to get a concrete contact plan, these times need to be stabilized into either the right zone or a concrete time-point. Whenever a full reset happens, a number is put back in after a few transitions. In order to handle this, a variable is used to keep track of what the time has been recorded to be as the stabilizing of time moves along. This variable is then assigned as a replacement for a full reset. A similar process is used for whenever a missing zone lower bound happen.

Finally, the contact plan can be produced. The contact plan produced by the Ulloriaq Contact Plan Extractor comes in two file-types: a .cnp file with every contact and their periods, and a .py file that contains the necessary setup to create Gantt-charts of the contact plans using a Python 3 [6] library called Plotly [17], assuming the right python libraries are installed. (Plotly is the main one; it should download any additional libraries automatically) An example of part of a contact plan can be seen in Figure 4.2. The contact plan first lists the number of actions taken for each satellite, in each category, mainly for debugging purposes. This can be seen at the top of the figure. Then, it has a line with periods for each satellite and each type of action listed in the rest of the file.

```
// Intersatellite ations for sat2: 231
// aal ations for sat2: 0
// grl ations for sat2: 71
...
// Intersatellite ations for sat4: 120
// aal ations for sat4: 0
// grl ations for sat4: 54
int n = 184;
...
const int sat3sat2[n][2] = {{373,374}, {383,384}, {385,387}, {393,394},
    ...
```

```
...
const int sat4sat3[n][2] = {{373,374}, {383,384}, {401,402}, {405,406},
    ...

const int sat0AAL[n][2] = {{413,415},{872,873},{965,966},{1062,1064}, ...

const int sat0GRL[n][2] = {{405,406},{409,410},{411,412},{498,499}, ...
...
const int sat4GRL[n][2] = {{276,277},{371,372},{373,374},{467,468}, ...

const int sat0Sun[n][2] = {{0, 1}, {28, 96}, {123, 192}, {219, 287}, ...
...
const int sat4Sun[n][2] = {{0, 58}, {85, 153}, {181, 249}, {276, 344}, ...
```

Listing 4.2: Example textual-format contact plan.

To be able to extract the correct contact plans, it was necessary to add an intermediate location to the *Ulloriaq Scheduling Model* to split up the updates on the edge going from `Synchronizing` to `Idle`. Previously, this part of the model looked as seen on Figure 4.9 in Section 4.1. The same part, but with the intermediate step can be seen in Figure 4.11.

Each of the periods is found by searching for the "Clean" function in the list of transitions found in the trace, and the time is then extracted from the state that belongs to that transition. The "Clean" function is called on the edge between `Idle` and `Intermediate`. Previously, this part of the model looked as seen in Figure 4.5, without an intermediate step.

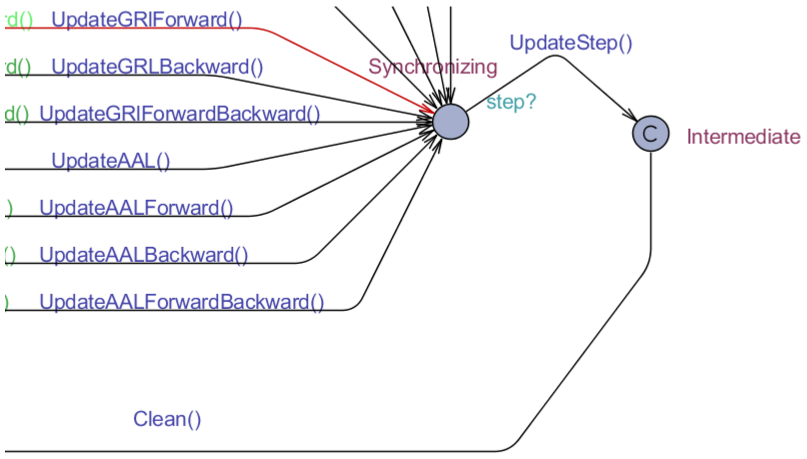


Figure 4.11: Modification of the *Ulloriaq Scheduling Model* to allow for time-extraction.

As these transitions will sometimes produce duplicate periods, some post-processing is also done to clean up the periods before they are added to the contact plan.

4.3 *Ulloriaq Post-Processing Model*

The purpose of the *Ulloriaq Post-Processing Model* is to verify that the contact plan, generated using the *Ulloriaq Scheduling Model*, still adheres to the constraints when using a more realistic battery model and introducing some uncertainties in regards to the battery. To do this, a model has been made using UPPAAL SMC and Stochastic Timed Automata, as described in Section 2.5, which implements the Kinetic Battery Model, as presented in Section 2.6.2. The verification is done by using UPPAAL SMC to calculate what the probability of reaching the end of the contact plan is, without having any satellites battery deplete further than the allowed amount.

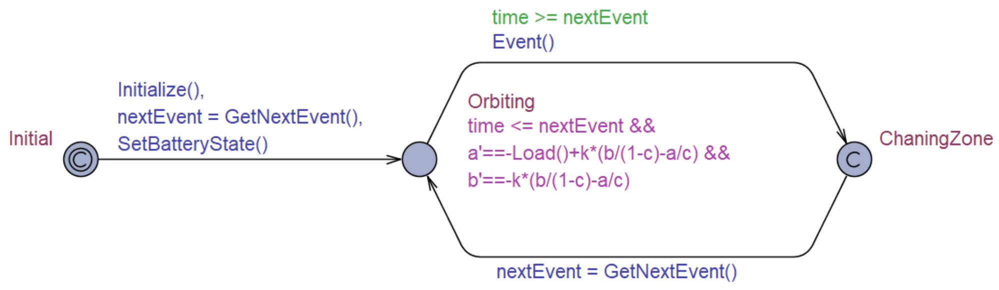


Figure 4.12: *Ulloriaq Post-Processing Model*

The model has an STA for each satellite in the contact plan, the template for which can be seen in Figure 4.12. Each satellite has three locations: `Initial`, `Orbiting` and `ChangingZone` . `Initial` is the initial location, and both it and `ChangingZone` are committed locations. Thus, the only location where time may pass is `Orbiting`. The functions, variables, constants, and clocks used are described in Table 4.7.

Functions	
void Initialize()	Assigns a contact plan to each satellite.
ind GetNextEvent()	Returns when the next event occurs. This can be for either a new event starting or an active one ending.
void Event()	Executed when an event occurs. Sets the current active events for the satellite.
double Load()	Returns the current load on the satellite. Dependant on the current active events.
void SetBatteryState()	Sets the initial values for the two KiBaM wells
Variables	
int nextEvent	The time at which the next event will occur.
Contants	
double k	The k contant for KiBaM
double c	The c contant for KiBaM
Clocks	
clock time	The global clock used to keep time in the model.
clock a	Clock representing the available charge in the satellite battery.
clock b	Clock representing the bound charge in the satellite battery.

Table 4.7: Attributes and functions used in the *Ulloriaq Post-Processing Model*

Each satellite is assigned four or five contact plans, two for the neighboring satellites, one for Greenland, one for when the satellite is in sunlight, and, if it is equipped with an HSL-enabled, one for Aalborg. A contact plan is defined in the model as a struct:

```
typedef struct {
    int from;
    int to; // GRL: -1, AAL: -2, sunlight:  -3
    int periods[n][2];
} ContactPlan;
```

Each *ContactPlan* contains an array of periods, where each period is an integer array of size two. A period defines when communication happen between the two participants, be it two satellites or a satellite and GRL or AAL. For consistency, this is also used to indicate whether or not the satellite is in sunlight. The first and second element of a period indicates the start and end time of said period respectively. An array of size two was chosen, instead of defining a second struct, since it made it easier to input the data into the model. When using an array data in a JSON-like format can be pasted directly into the model. An example of such data three satellites can be seen on the following listing:

```
const int sat1sat0[n][2] = {{10,12}, {14,17}, {87,93}, {100,101}};
const int sat1sat2[n][2] = {{10,11}, {13,14}, {87,88}, {108,109}};
const int sat2sat1[n][2] = {{12,13}, {15,16}, {84,85}, {89,91}};
const int sat2sat3[n][2] = {{82,83}, {99,100}, {102,103}, {197,198}};
const int sat3sat2[n][2] = {{82,85}, {88,91}, {101,104}, {179,182}};
const int sat3sat4[n][2] = {{88,91}, {187,189}, {284,285}, {287,288}};

const int sat0AAL[n][2] = {{316,317},{323,324},{411,412},{413,414}};

const int sat0GRL[n][2] = {{19,25},{118,121},{214,219},{309,315}};
const int sat1GRL[n][2] = {{9,16},{108,111},{204,209},{299,305}};
const int sat2GRL[n][2] = {{98,102},{194,199},{290,295},{385,392}};

const int sat0Sun[n][2] = {{0, 1}, {28, 96}, {123, 192}, {219, 287}};
const int sat1Sun[n][2] = {{18, 87}, {114, 182}, {209, 277}, {305,
    373}};
const int sat2Sun[n][2] = {{9, 77}, {104, 173}, {200, 268}, {295,
    363}};
```

Each satellite must initialize, by following the edge between `Initial` and `Orbiting`, before time may pass. When doing this, three things happen. First the `Initialize()` function is called, which assigns the correct contact plans to each satellite, after which the first event for the satellite is found and assigned to the *nextEvent* variable, and lastly, the initial charge for the two wells of the KiBaM battery is assigned. When each satellite is initialized, they start going through the main loop of the STA until they reach the goal time. While a satellite is in the main loop, it changes location between `Oribiting` and `ChangingZone`. This happens each time the satellite reaches the the time for the next event, as a result of the guard `time >= nextEvent` and the invariant `time <= nextEvent` on `Orbiting`. Each time a satellite changes location to `ChangingZone` the `Event()` function is called, which updates the value of internal booleans used to indicate the different connections the satellite has at a given time. Lastly, the variable *nextEvent* is updated as seen on the edge `nextEvent = GetNExtEvent ()`, and the satellite returns to `Orbiting`.

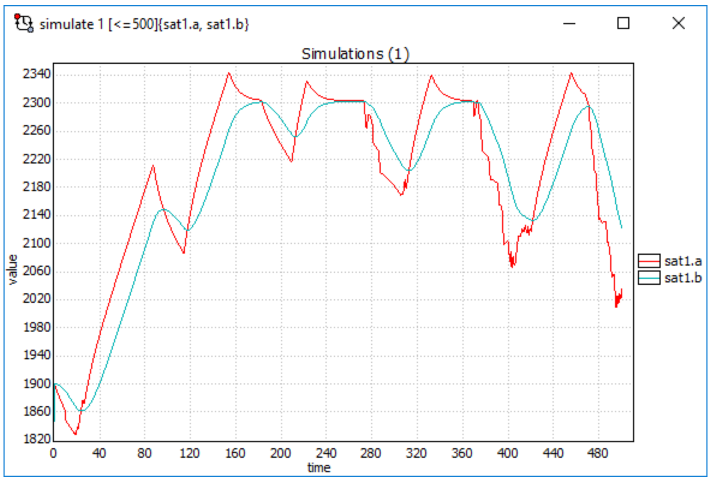


Figure 4.13: Evolution of KiBaM wells *a* and *b* over 500 minutes

The two wells of the kinetic battery are implemented as clocks, namely clock *a* and *b*. The rate of these clocks are set to the formulas given in Section 2.6.2 for KiBaM, where the function `Load()` is the load *I* from the formula. An example of how the charge of the two wells change for the first 500 minutes of a contact plan can be seen in Figure 4.13.

4.3.1 Uncertainty

Since the models represent real-world objects, some uncertainty in the environment is expected. The authors of [12] introduced two areas of uncertainty, in regards to the satellite battery, as explained in Chapter 2.1. Firstly the initial state of charge for the batteries vary over a uniform distribution in range $[-4\%, 4\%]$. Secondly, the charge needed for a satellite to communicate has an uncertainty of +- 0.5 J.

The battery uncertainty is implemented in the `SetBatteryState()` function:

```
void SetBatteryState(){
    double charge = initCharge * (1 - ((random(8.0) - 4) / 100));
    b = charge * (1-c);
    a = charge * c;
}
```

This function assigns a double to each of the two wells of the battery. The constant *initCharge* represents the initial value for the battery. To include the uncertainty, this value is changed by a percentage in the range $[-4\%, 4\%]$. The percentage is generated by the expression on the first line of the function. Here *random(8.0)* returns a random number in the range [0, 8], this is done by SMC making a uniform distribution over the range. This generated value is pulled down to the desired value by subtracting four. In this way, if the number generated is 0, the outcome will be -4. Lastly, the value, now in the wanted range, is divided by 100 and subtracted from one, to get a decimal to scale the initial charge with.

The communication load uncertainty is implemented in the `Load()` function:

```
double Load(){
    double temp = backgroundLoad - (charging * rechargeLoad);

    int i;
    for(i=0;i < (forward + backward + GRL + AAL +
   connections[GetForward()][id] +
   connections[GetBackward()][id]);i++){
        temp = temp + transmitLoad + ((random(1.0) - 0.5) * 60);
    }

    if(temp < 0 && (a+b) - temp >= maxCharge){
        return maxCharge - (a+b);
    }

    return temp;
}
```

The way this uncertainty is implemented is similar to how the battery uncertainty is implemented. This function returns the current load on the satellite's battery. Each satellite has a Boolean for each possible communication, these are used to calculate the amount of simultaneously connections like this: *(forward + backward + GRL + AAL)*. The result of this is used in a for-loop to add the *transmitLoad* plus the uncertainty. The uncertainty for this load represents a jitter in the load. *random(1.0)* is again used to get a uniform distribution in the range $[0, 1]$, subtracting 1 to bring it down to the correct range $[-0.5, 0.5]$. If the satellite is currently in sunlight and charging then the constant *rechargeLoad* is added to the total load.

4.3.2 Query

To do the verification of the contact plans, the following queries were used:

```
// Probability of battery going below min
```

```
Pr[<=endTime] (<> sat0.a + sat0.b < minCharge)
Pr[<=endTime] (<> sat1.a + sat1.b < minCharge)
Pr[<=endTime] (<> sat2.a + sat0.2 < minCharge)
Pr[<=endTime] (<> sat3.a + sat3.b < minCharge)
Pr[<=endTime] (<> sat4.a + sat4.b < minCharge)
Pr[<=endTime] (<> sat5.a + sat5.b < minCharge)
Pr[<=endTime] (<> sat6.a + sat6.b < minCharge)
Pr[<=endTime] (<> sat7.a + sat7.b < minCharge)
Pr[<=endTime] (<> sat8.a + sat8.b < minCharge)
Pr[<=endTime] (<> sat9.a + sat9.b < minCharge)

// Probability of running out of available charge
Pr[<=endTime] (<>sat0.a <= 0)
Pr[<=endTime] (<>sat1.a <= 0)
Pr[<=endTime] (<>sat2.a <= 0)
Pr[<=endTime] (<>sat3.a <= 0)
Pr[<=endTime] (<>sat4.a <= 0)
Pr[<=endTime] (<>sat5.a <= 0)
Pr[<=endTime] (<>sat6.a <= 0)
Pr[<=endTime] (<>sat7.a <= 0)
Pr[<=endTime] (<>sat8.a <= 0)
Pr[<=endTime] (<>sat9.a <= 0)
```

The first set of queries calculates the probability of each satellites battery total charge ever gets lower than the allowed amount. The second set of queries calculates the probability of any of the satellite batteries running out of available charge, in which case the satellite would die.

5 | Experiments

To be able to evaluate the success of the models, a series of experiments were performed. The experiments were chosen to explore the performance of the model and how different constellations and settings affect the generated contact plan. It was decided to adjust and test 5 different areas:

1. Number of satellites
   - Measuring the impact an increased number of satellites has on the performance of the model.
   - Measuring the general performance on different numbers of satellites.
2. Amount of data collected
   - What is the maximum amount of data the model can handle being transmitted within 48 hours?
3. Simulation length
   - Measuring the impact of different simulation lengths
   - Comparing full schedule performance versus the performance of multiple smaller schedules
4. Impact of heuristics
   - Turning each heuristic off one at a time, and measuring the performance impact.
5. Different cost expressions
   - No battery cost
   - No Data cost
   - No cost

After this, the results are compared to the MILP model, as described in Section 3.2.

5.1 Experiment environment

The experiments were run on an Intel-based Ubuntu machine, using a command-line version of Uppaal Cora[20]. The version of Ubuntu was the most recent long-term-support version, which at the time was 18.02. The machine had a clock speed of 4.5 GHz, and 32 gigabytes of RAM. The main factor that had an effect on the experiments was the single-core performance, as the used version of UPPAAL CORA could only utilize a single core at a time. The amount of RAM did not affect the experiments, as the used version of CORA similarly was limited to only using up to 4 GB of RAM, due to being a 32-bit program. In each experiment, the simulation is run four times for ten real-time minutes. The at that moment, best-performing instance, determined via the cost, is then used as the result.

5.1.1 Settings

The following settings were used with the commandline version of UPPAAL CORA to run the verification process:

- *-t3 -o4 -u*
  - *t* is for setting what kind of trace we want to find, which in this case is the best trace.
  - *0* is for setting the search method, in this case, Random Best Depth First Search (RBDFS).
  - *u* is for providing us with additional statistics at the end of the simulation about the state-space size.

RBDFS was choosen over a more traditional search order like *Best First* (BFS) due to the large state-space when running a query that spans such a large timespan and has many synchronizing components. The state space for the *Ulloriaq Scheduling Model*, is big, due to the sheer amount of different actions that can be taken at a given time. If BFS were to be used, UPPAAL CORA would run out of memory before a search could be completed. Every experiment, except the one without any cost, was run with "Random Best Depth First" (RBDFS) search order.

The state-space of the models used in the experiments is extremely large due to the high number of choices available at every point in time. This poses a hindrance to us finding traces that fulfill the requirements, within a reasonable time. It also makes it impossible to find a best-cost trace within the limitations of UPPAAL CORA. The traces needed are usually rare, and in some cases, the verifier has a hard time finding a trace within the current part of the state-space it is searching. This is also why RBDFS is a good choice, as it lets us search very different parts of the state-space, while also using cost. A depth-first search order is to prefer, due to the huge size of the state-space being explored, and the relative few traces where the goal is met. Adding the random element spreads the search out trough state-space, which is necessary since CORA runs out of memory long before it has gone through or pruned the whole space.

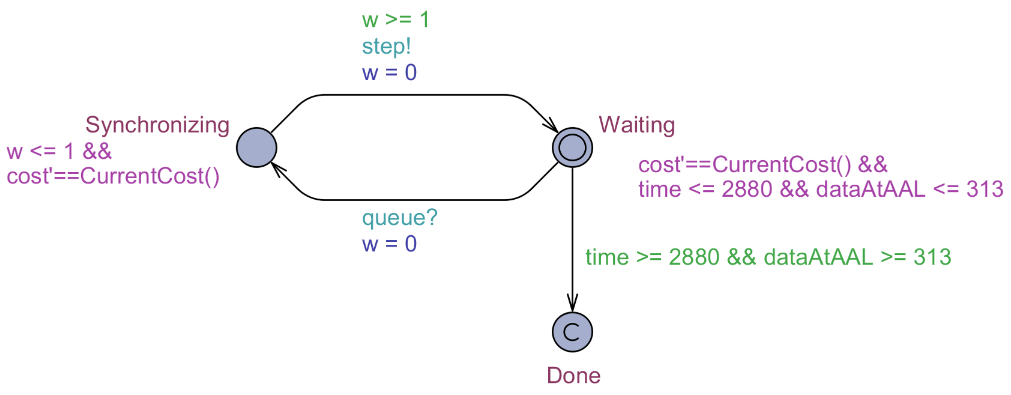


Figure 5.1: The goal location "Done" is found in the lower right part of the template.

5.1.2 Queries

When running an experiment, the query would usually look like the following:

```
E<> time >= 2880 && dataAtAal >= 313
```

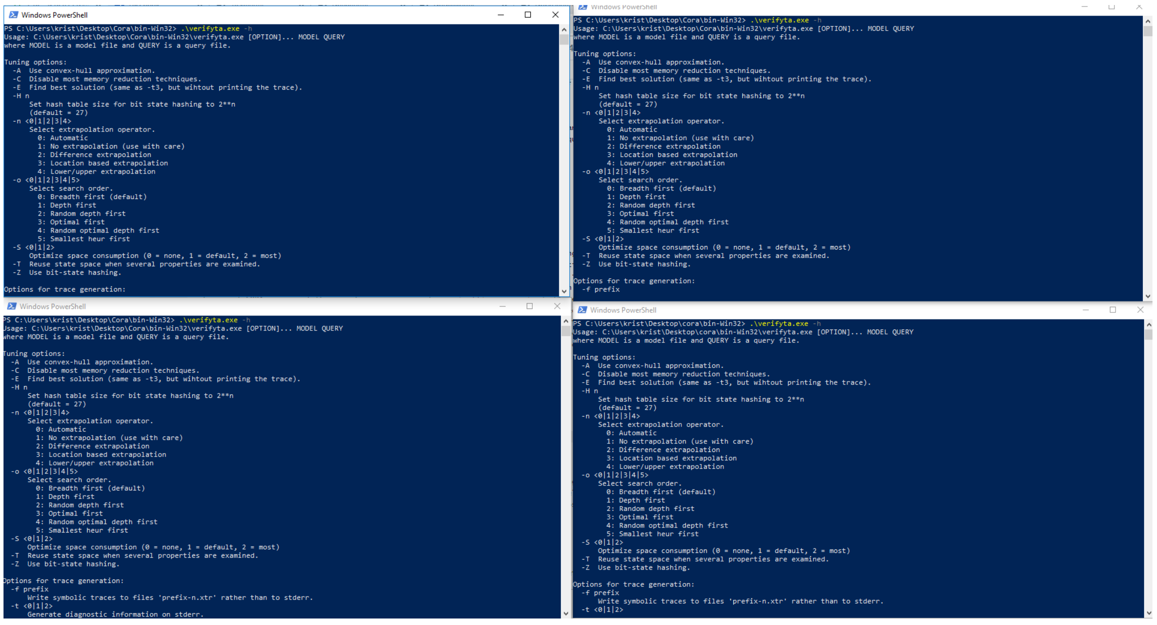
2880 and *313* would then be varied based on the type of experiment. Any experiment that does not rely on either the *2880* or *313* being altered use these as the standard values. As described in Section 4.1.4, these variables are derived from [12], 2880 being the 48 hours they simulated, and 313 being the amount of data they delivered in this time-period.

During experiments, it was discovered that certain simulations were unable to find a better trace than the initial. To combat this, a *goal-location* was implemented in the Synchronizer part of the *Ulloriaq Scheduling Model*, as seen on Figure 5.1, which makes sure that the system deadlocks when the goal data and time is reached.

This *goal-state* achieves the same result as having it in a query, by having guards with the variables in the query, but prevents idling before delivering the final data. The new query, used for every experiment is the following:

```
E<> w.Done
```

The cost used for running the experiments is explained in 4.1.5.



```
PS C:\Users\krist\Desktop\Cora\bin-Win32> .\verifyta.exe -h
Usage: C:\Users\krist\Desktop\Cora\bin-Win32\verifyta.exe [OPTION]... MODEL QUERY
where MODEL is a model file and QUERY is a query file.

Tuning options:
  -A  Use convex-hull approximation.
  -C  Disable most memory reduction techniques.
  -E  Find best solution (same as -t3, but wihtout printing the trace).
  -H n
      Set hash table size for bit state hashing to 2**n
      (default = 27)
  -n <0|1|2|3|4>
      Select extrapolation operator.
        0: Automatic
        1: No extrapolation (use with care)
        2: Difference extrapolation
        3: Location based extrapolation
        4: Lower/upper extrapolation
  -o <0|1|2|3|4|5>
      Select search order.
        0: Breadth first (default)
        1: Depth first
        2: Random depth first
        3: Optimal first
        4: Random optimal depth first
        5: Smallest heur first
  -S <0|1|2>
      Optimize space consumption (0 = none, 1 = default, 2 = most)
  -T  Reuse state space when several properties are examined.
  -Z  Use bit-state hashing.

Options for trace generation:
```

```
PS C:\Users\krist\Desktop\Cora\bin-Win32> .\verifyta.exe -h
Usage: C:\Users\krist\Desktop\Cora\bin-Win32\verifyta.exe [OPTION]... MODEL QUERY
where MODEL is a model file and QUERY is a query file.

Tuning options:
  -A  Use convex-hull approximation.
  -C  Disable most memory reduction techniques.
  -E  Find best solution (same as -t3, but wihtout printing the trace).
  -H n
      Set hash table size for bit state hashing to 2**n
      (default = 27)
  -n <0|1|2|3|4>
      Select extrapolation operator.
        0: Automatic
        1: No extrapolation (use with care)
        2: Difference extrapolation
        3: Location based extrapolation
        4: Lower/upper extrapolation
  -o <0|1|2|3|4|5>
      Select search order.
        0: Breadth first (default)
        1: Depth first
        2: Random depth first
        3: Optimal first
        4: Random optimal depth first
        5: Smallest heur first
  -S <0|1|2>
      Optimize space consumption (0 = none, 1 = default, 2 = most)
  -T  Reuse state space when several properties are examined.
  -Z  Use bit-state hashing.

Options for trace generation:
  -f prefix
```

```
PS C:\Users\krist\Desktop\Cora\bin-Win32> .\verifyta.exe -h
Usage: C:\Users\krist\Desktop\Cora\bin-Win32\verifyta.exe [OPTION]... MODEL QUERY
where MODEL is a model file and QUERY is a query file.

Tuning options:
  -A  Use convex-hull approximation.
  -C  Disable most memory reduction techniques.
  -E  Find best solution (same as -t3, but wihtout printing the trace).
  -H n
      Set hash table size for bit state hashing to 2**n
      (default = 27)
  -n <0|1|2|3|4>
      Select extrapolation operator.
        0: Automatic
        1: No extrapolation (use with care)
        2: Difference extrapolation
        3: Location based extrapolation
        4: Lower/upper extrapolation
  -o <0|1|2|3|4|5>
      Select search order.
        0: Breadth first (default)
        1: Depth first
        2: Random depth first
        3: Optimal first
        4: Random optimal depth first
        5: Smallest heur first
  -S <0|1|2>
      Optimize space consumption (0 = none, 1 = default, 2 = most)
  -T  Reuse state space when several properties are examined.
  -Z  Use bit-state hashing.

Options for trace generation:
  -f prefix
      Write symbolic traces to files 'prefix-n.xtr' rather than to stderr.
  -t <0|1|2>
      Generate diagnostic information on stderr.
```

```
PS C:\Users\krist\Desktop\cora\bin-Win32> .\verifyta.exe -h
Usage: C:\Users\krist\Desktop\cora\bin-Win32\verifyta.exe [OPTION]... MODEL QUERY
where MODEL is a model file and QUERY is a query file.

Tuning options:
  -A  Use convex-hull approximation.
  -C  Disable most memory reduction techniques.
  -E  Find best solution (same as -t3, but wihtout printing the trace).
  -H n
      Set hash table size for bit state hashing to 2**n
      (default = 27)
  -n <0|1|2|3|4>
      Select extrapolation operator.
        0: Automatic
        1: No extrapolation (use with care)
        2: Difference extrapolation
        3: Location based extrapolation
        4: Lower/upper extrapolation
  -o <0|1|2|3|4|5>
      Select search order.
        0: Breadth first (default)
        1: Depth first
        2: Random depth first
        3: Optimal first
        4: Random optimal depth first
        5: Smallest heur first
  -S <0|1|2>
      Optimize space consumption (0 = none, 1 = default, 2 = most)
  -T  Reuse state space when several properties are examined.
  -Z  Use bit-state hashing.

Options for trace generation:
  -f prefix
      Write symbolic traces to files 'prefix-n.xtr' rather than to stderr.
  -t <0|1|2>
```

Figure 5.2: Experiment setup

5.1.3 Running an experiment

Due to the nature of the state-space, finding a trace that returns a near-best cost in reasonable time, and without running out of memory, is practically unfeasible. Furthermore, it is also rare for a trace to provide continuing improvements in costs, also due to how rare a trace that returns a cost is. For this reason, each experiment was run with 4 instances of CORA running at the same time, as seen in figure 5.2. Whenever each instance stabilized on a given cost, the instances with the worst cost would be restarted until a new trace with a better cost was found, thus improving the overall cost found at the end of the experiment time-window. Each experiment was run for 10 real-time minutes, using the process described above. These issues also meant it was impossible for us to find a best-cost trace for any simulation of the model. Instead, we performed the procedure as listed above, to get a "near-best-cost".

5.1.4 Understanding the results

In the following sections, tables are presented with different columns, each representing some type of measurement made with that experiment. Each experiment only contains columns that are relevant to that experiment; An experiment with no cost associated with it will not have a column with cost. Each element is explained in the following list:

1. Time to collect all data
    - How many minutes (time units) did it take to collect the amount of data set by

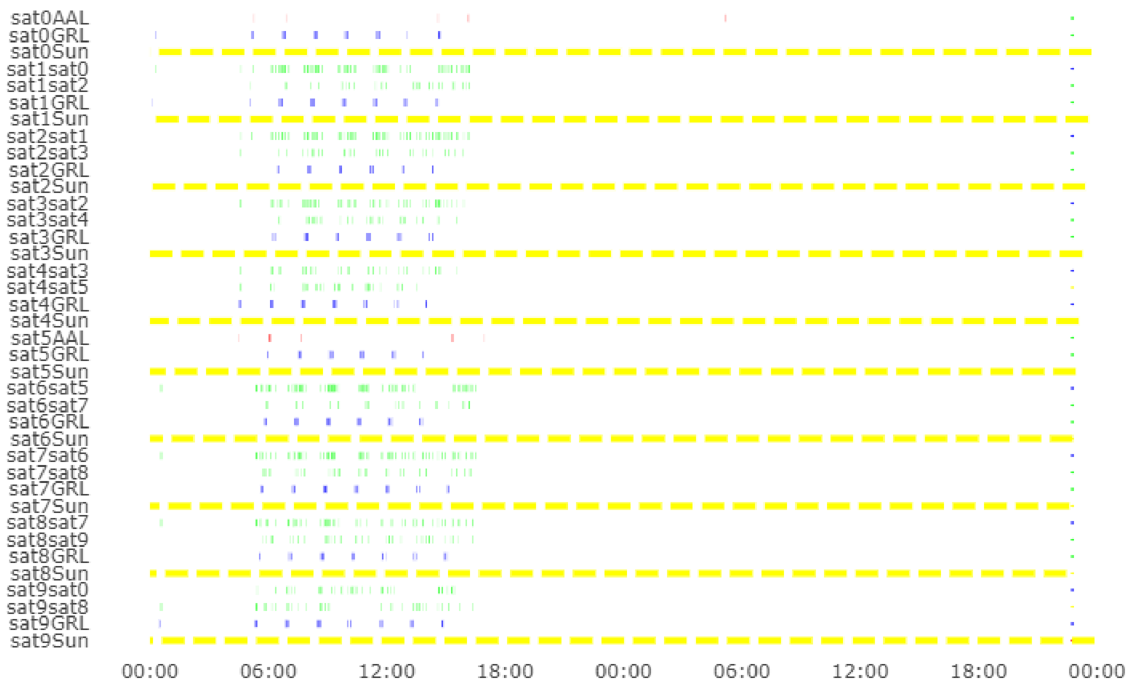


Figure 5.3: An example of a contact plan for 10 satellites.

the goal.

2. Time to deliver all data
   - How many minutes (time units) did it take to deliver the amount of data set by the goal.
3. Time to Trace
   - How long (real-time) did it take to computed a trace.
4. Cost
   - The accumulated cost, defined by missing battery charge and data needed to reach the goal, for the resulting trace.

Time to collect all data and *Time to deliver all data* is determined by looking at the last point in time a collection of data was made over Greenland.

In addition to tables with the aforementioned rows, a contact plan was generated for each experiment. An example of a contact plan can be seen in Figure 5.3.

On Figure 5.3, the X-axis is time, and the Y-axis denotes which satellite and what action is shown on that line. An example could be the first line from the top: "sat0AAL". This title means that on this line, the periods for when satellite 0 connects to Aalborg is listed. Each type of action is also color-coded:

- Red: Connecting to Aalborg.

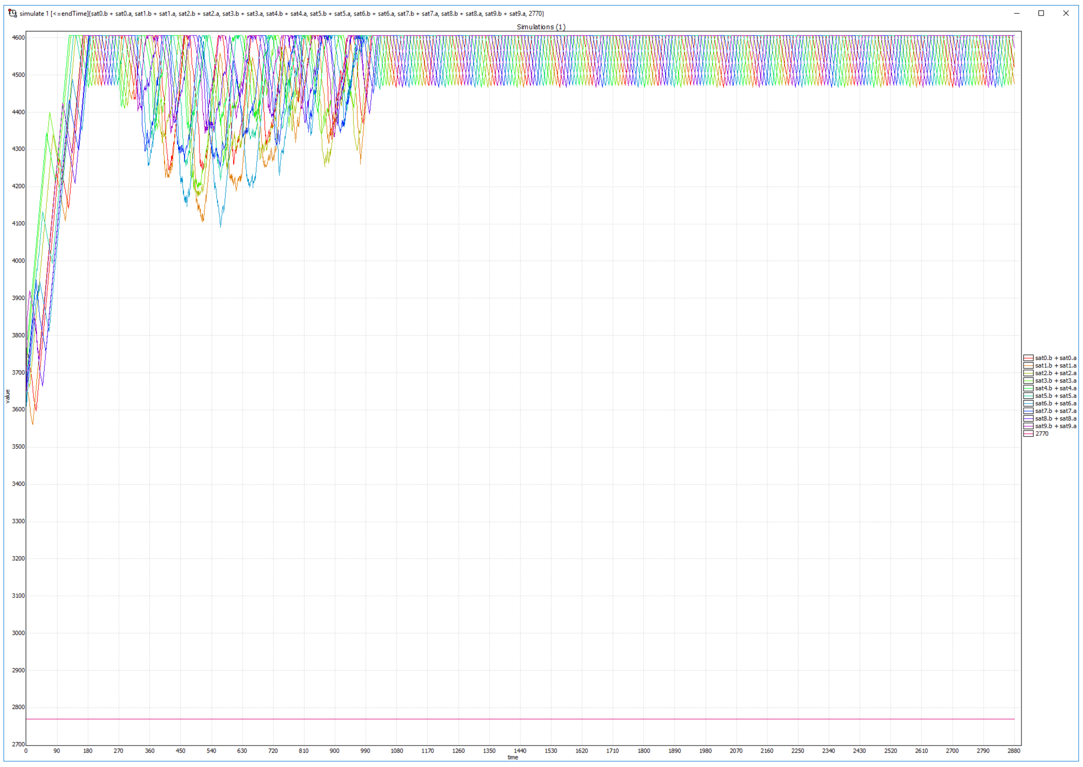


Figure 5.4: 10 satellites battery graph.

- Blue: Collecting data from Greenland.
- Green: Transmitting data between satellites.
- Yellow: The satellite is in sunlight.

Every contact plan generated and not directly used in this section can be found in Appendix A. An example of a battery graph for a contact plan with 10 satellites can be seen on Figure 5.4. Each experiment has its own graph such as this one. Each coloured line represents the evolution of battery charge throughout the experiment. The X-axis is time, and the Y-axis is the battery charge. The *Ulloriaq Post-Processing Model* provides the battery graphs, as explained in 4.3, but also provides a probability for how likely a battery on a given satellite is to deplete, during a given trace. Due to it being impossible to generate a contact plan that drains the battery close the allowed state of charge, resulting in a probability of dying of 0%, it was decided to leave this metric out of the result tables.

5.2 Experiment 1: Number of satellites

The objective of this experiment was to adjust the number of satellites and measure the impact of this on the performance of the model. Three separate models were created, with five, six and ten satellites in the constellation. It was decided to use those specific numbers of satellites for the following reasons:

1. 5 satellites
   - 5 was chosen to see how the model performance changed with only one HSL-enabled satellite.
2. 6 satellites
   - 6 was chosen to see how much enabling one additional HSL satellite changed the performance.
3. 10
   - 10 is the number of satellites proposed for the Ulloriaq case.

The model was modified to include the corresponding amount of satellites for each experiment. This was done my editing the constant *satCount* and by initializing the same amount of the *Satellite* template as were needed. No other changes were made, to ensure comparability of the results. In this experiment, the following elements were measured: *Time to collect all data, Time to deliver all data* and *Time to Trace, Cost*. In addition to these metrics, each column has two additional columns, *Experiment #* and *Satellite Count* which describes the number of satellites used in that experiment as well as how many of those were HSL-enabled satellites.

Results

Every simulation in this experiment completed instantly. Thus no *Time to Trace* row differed between the different versions of the model. The goal of this experiment was to determine how the number of satellites affected the performance of the model. This is measured by *Time to collect all data* and *Time to deliver all data*. Table 5.1 contains the results of the simulation. The corresponding contact plans can be found in Appendix A. As expected, the results show that with the number of satellites, the time to deliver all data decreases. Due to a large gap in the periods in which HSL-enabled satellites can connect to Aalborg, the *Time to Deliver* row in the table has some extremely late delivering of the final part of the data to Aalborg. Thus it is hard to see the benefits of having more satellites when comparing *Time to deliver all data*, and easier to see the comparison between *Time to collect all data* rows. While a straight increase to the number of satellites clearly affect *Time to deliver all data*, as seen by experiment 3 completing in 7 hours and 15 minutes less than experiment 2, it is important to note that the addition of an HSL-enabled satellite is much more important for *Time to deliver all data*. This is seen by experiment 2 completing in 11 and a half hour faster than experiment 1, which has 1 less HSL-enabled satellite. Furthermore, the cost increases with the number of satellites, as this directly affects the number of actions taken, which in turn increases the cost due to the battery drained by these actions.

Exp #	1	2	3
Satellite Count	5 Satellites - 1 HSL	6 Satellites - 2 HSL	10 Satellites - 2 HSL
Time to collect all data	33:22	22:04	14:46
Time to deliver all data	38:26	29:08	29:10
Time to Trace	Instant	Instant	Instant
Cost	3630002	3749283	5199691

Table 5.1: Simulations of various numbers of satellites

Exp #	1	2	3	4
Time to collect all data	28:26	37:52	43:36	Did not finish
Time to deliver all data	29:07	39:11	47:59	Did not finish
Data collected	626	900	1100	Did not finish(Goal: 1300)
Time to trace	Instant	5s	15s	Did not finish
Cost	7105119	8945857	10516451	Did not finish

Table 5.2: Maximum amount of data obtainable in 2880 time-units

5.3 Experiment 2: Maximum data collection

The objective of this experiment was to obtain a maximum on how much data the *Ulloriaq Scheduling Model* can handle within 2880 time-units. This was done by adjusting the *goal* variable used in the *goal-state* found on Figure 5.1 upwards, until it was not possible to find a trace anymore without CORA running out of memory. In this experiment, the following rows were used: *Satellite Count, Time to collect all data, Time to deliver all data, Time to Trace, Cost*. *Time to Trace* is the main way to measure the performance of the model in this experiment.

Results

Several experiments were made while exploring the maximum estimated data collection. The results of these experiments can be found on Table 5.2.

As expected, *Time to collect all data* and *Time to deliver all data* grew as the amount of data being transmitted grew. In addition to this, Cost and *Time to trace* grew as well.

An issue with this experiment was the time it took to find a trace; *Time to Trace* grew fast as we approached values around what was collected in experiment 3 in Table 5.2. As explained in Section 5.1.1, one of the issues found during experimentation, was that finding a trace that fulfilled the goal was rare. When reaching larger goal data amounts, as in experiment 3 in Table 5.2, this trace becomes even more rare and eventually impossible. A second issue is with UPPAAL CORA, when trying to find a trace of this length, with this much collection

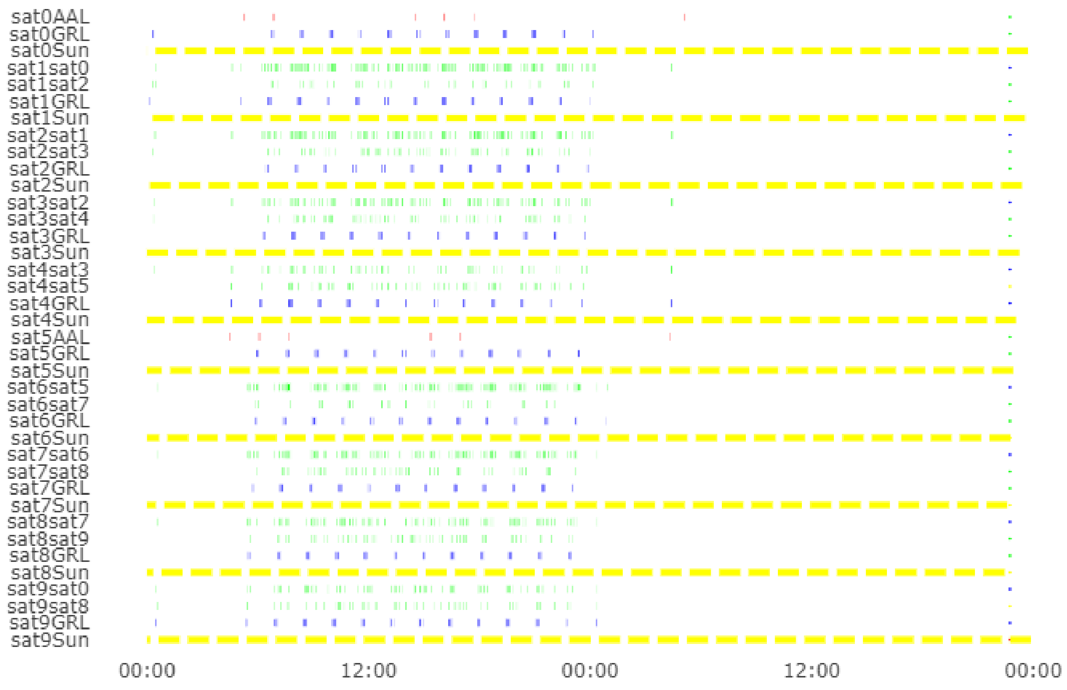


Figure 5.5: 626 data collected

activity, and thus many actions to be stored in the trace, it runs out of memory before a better trace can be found. Therefore it is theoretically possible to find a better result than what we have found, but it would require a 64-bit version of UPPAAL CORA. The following figures each contains a contact plan for the listed amount of data. The first contact plan, found in Figure 5.5, transmitted double the data that was transmitted on Figure A.3. The second and third contact plan, found on figures 5.6 and 5.7, both transmit increasing amounts of data, which is also reflected in the number of actions taken on the contact plan(red blue and green lines).

While it is not likely that the amount of data collected as a maximum in this experiment would deplete any satellites batteries, as the number of actions increases, this would become a problem as the satellites perform more and more actions. The fact that collecting the maximum amount of data in experiment 3 will not deplete the battery, can also be seen by the fact that on Graph 5.8, no battery reaches below the initial charge levels after the initial phase of the contact plan.

A final experiment, with 1300 as the goal amount of data, was made. This experiment never finished, and is listed as such in the 4th column of table 5.2.

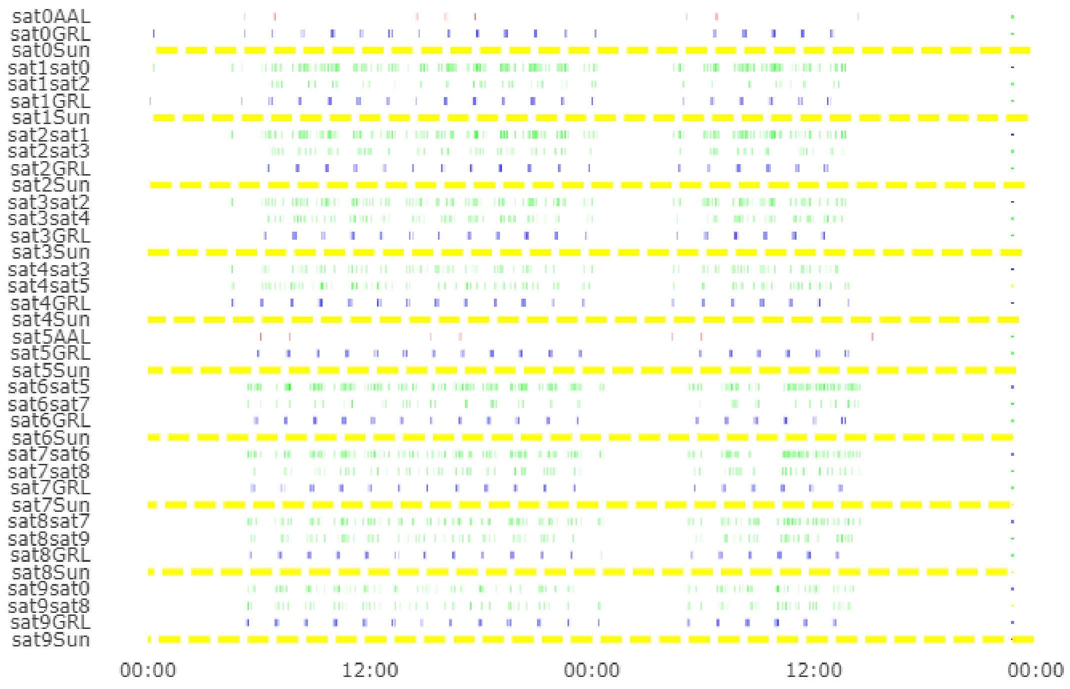


Figure 5.6: 900 data collected

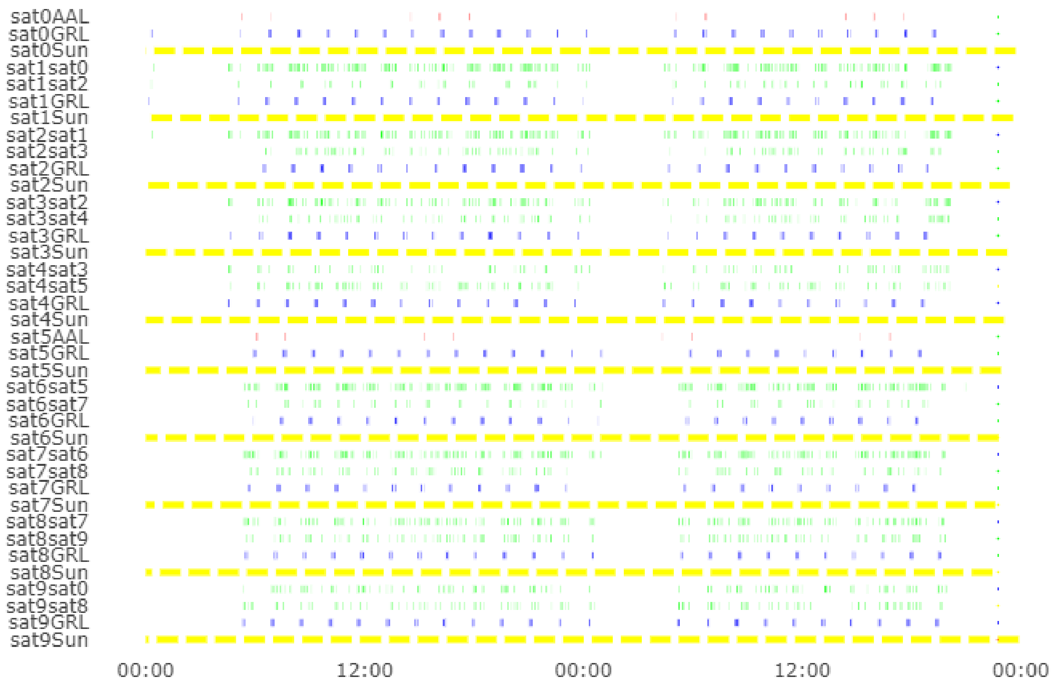


Figure 5.7: 1100 data collected

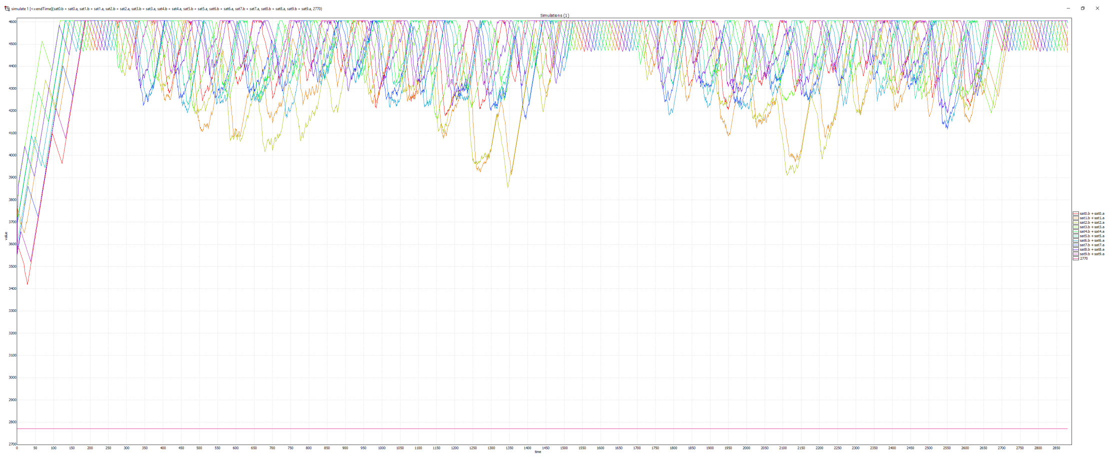


Figure 5.8: Battery graph for experiment 3: Collecting 1100 data

5.4 Experiment 3: Simulation length

The objective of this experiment was to see what the impact of varying simulation lengths had on the model. A side goal was to see whether or not it would be possible to obtain traces for smaller periods of the full time, 2880 time-units, that in combination delivered more data than what a full trace could deliver in the same time. It was chosen to perform this experiment with 2 times 1440 time-units and the full 2880 time-units. Each sub-experiment is rerun until the max amount of data that can be achieved for that sub-experiment is achieved, after which the resulting data and statistics are compared. Two changes were made in the 2x 1440 model as opposed to the 2880 model: The *goal-location* variable concerning time was modified to be 1440 time-units, and each satellite was reconfigured to receive its initial state based on an array of variables containing the result-state of the previous 1440 time-period simulation.

Results

The goal of this experiment was to determine whether or not it was possible to transmit more data via a contact plan computed in two parts, as opposed to computed in one simulation. The results of this experiment are found in Table 5.3.

As seen in Table 5.3, it was possible to schedule slightly more data using two separate 24-hour traces. As these traces are only approximating a best version of the trace, it is not certain that the scheduling in two times 24 hours would perform better. In addition to this, it is also not possible to compare the cost of the two traces, as changes were made to the model to accommodate the second part of the 24-hour model. The contact plan for scheduling two times 24 hours also showcases behaviour similar to the one exhibited by the contact plan on

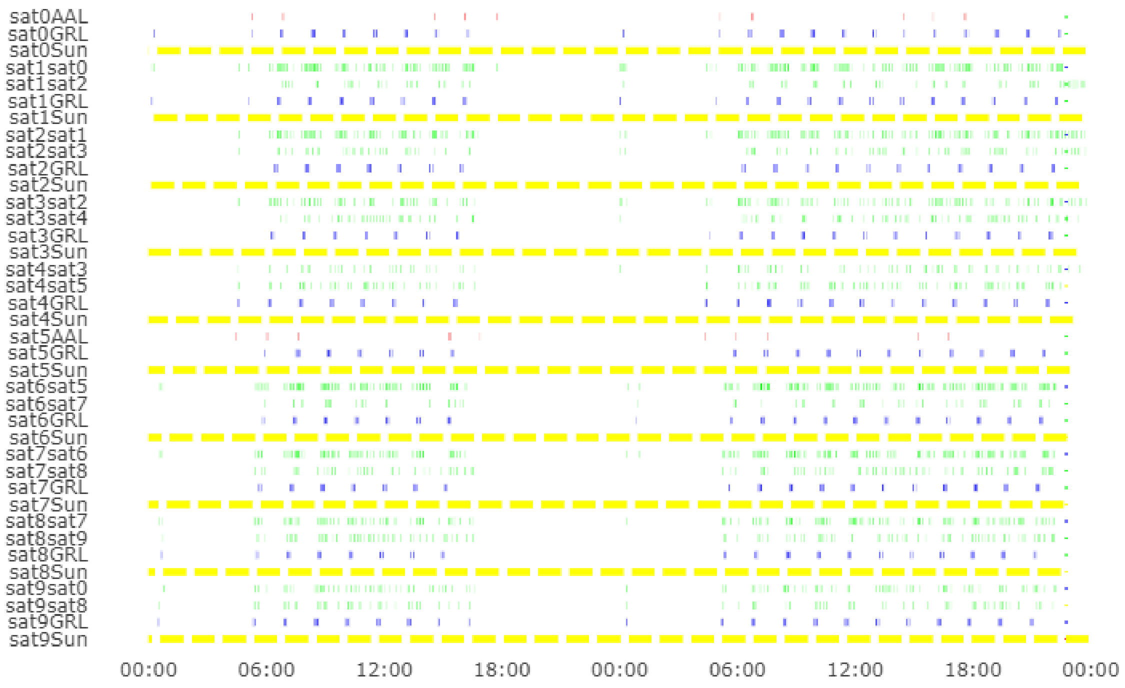


Figure 5.9: Two 24-hour contact plans assembled into one contact plan.

Exp #	24hr pt1&pt2	48
Time to collect all data	46:27	43:36
Time to deliver all data	48	48
Time to Trace	Instant	Instant
Data delivered	1200	1100
Cost	9266076	10516451

Table 5.3: Statistics for simulating 48 hours as two separate traces, and for a full 48 hours.

Exp #	Heur 1	Heur 2	Heur 3	Heur 5	Heur 6	No Heuristic
Time to collect all data	14:02	15:16	14:28	13:06	15:07	Did not finish
Time to deliver all data	28:21	16:58	16:58	29:10	28:22	Did not finish
Time to Trace	Instant	Instant	Instant	Instant	Instant	Did not finish
Cost	7253390	5106368	5200546	13213193	5232765	Did not finish

Table 5.4: The results of turning one heuristic off at a time.

Figure 5.13.

5.5 Experiment 4: Impact of heuristics

The objective of this experiment, was to measure the impact of the heuristics implemented in the *Ulloriaq Scheduling Model*. The heuristics are described in section 4.1.3. In summary, the goal of each heuristic was to either disallow certain un-beneficial types of behavior, or improve the performance of the model, either by reducing the state-space or computation time. Out of the six heuristics listed, five were tested. Heuristic 4, found at 4.1.3, was not tested, due to it being too deeply ingrained in the model itself. Modifying the model to remove this heuristic would make it too difficult to compare the results. In addition to testing with each heuristic on and off, an experiment was made without any heuristics except for heuristic 4. Each heuristic was tested with every other heuristic enabled. No other changes were made to the models. It could be argued that the experiments should have been performed with only one heuristic enabled at a time, but this was not due to CORA running out of memory before being able to produce a trace. The results of these experiments can be found in Table 5.4.

Results

The conclusion for each heuristic is described in the following sections. Table 5.4 contains the results from every heuristic.

Heuristic 1

Heuristics 1 was: *No ping-pong data transfer*. The measure of success for this heuristic is measured mostly in how many actions were taken by the satellites, as part of the contact plan. The first and most easily discernible difference is in the fact that the cost for this experiment is higher than that for an equivalent system with: *7253390* in this experiment, as opposed to the experiment found in 5.2, where the cost for a 10-satellite-2-HSL system is *5199691*. As cost increases with battery usage and data-delivery latency, that is reflected

	10-Satellites - Heur 1 enabled	10-Satellites - Heur 1 disabled	% difference
Sat0 inter-sat actions	0	0	0%
Sat1 inter-sat actions	121	127	5,0%
Sat2 inter-sat actions	119	168	41,2%
Sat3 inter-sat actions	94	166	76,6%
Sat4 inter-sat actions	68	107	57,4%
Sat5 inter-sat actions	0	0	0,0%
Sat6 inter-sat actions	116	155	33,6%
Sat7 inter-sat actions	116	211	81,9%
Sat8 inter-sat actions	95	192	102,1%
Sat9 inter-sat actions	73	104	42,5%

Table 5.5: Amounts of inter-satellite actions performed by each satellite, with Heuristic 1 enabled and disabled.

in this experiment, where satellites are allowed to send data back and forth endlessly. The second measure is in the number of actions taken by this experiment and the one found in Section 5.2, as seen in Table 5.5. The number of actions for the experiment with heuristic one off is on average 44% higher than with it enabled.

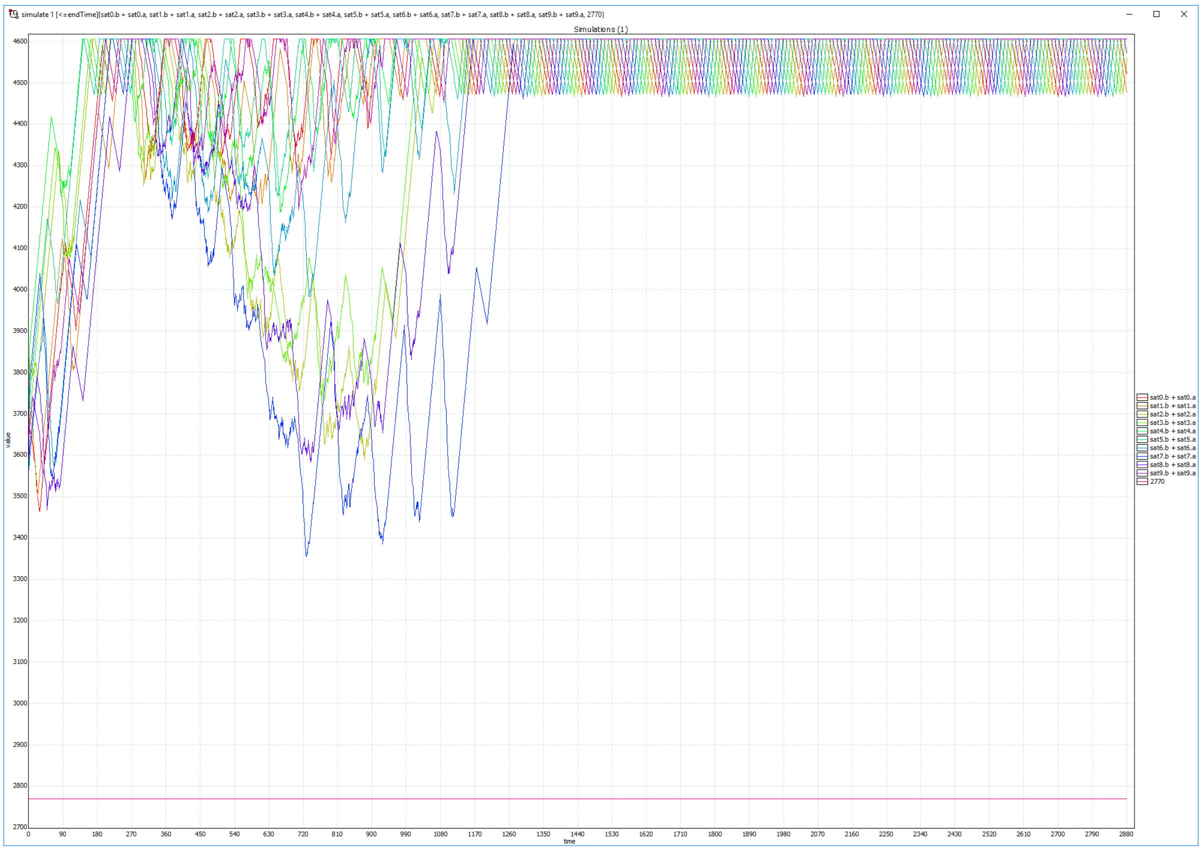
This behaviour of increased inter-satellite actions is also reflected on the battery usage, as seen on Figure 5.10. On this, the battery levels drain much more drastically than what is seen on Figure 5.8.

Heuristic 2

Heuristic 2 was: *Data must move towards HSL satellites*. The results from this heuristic are largely identical to the ones from Section 5.2, for a 10 satellite contact plan. This can be seen in Table 5.4, as the *Cost* and *Time to Deliver* variables for *Heur 2* are almost equal. As far as is measurable, this means that this heuristic does not make a big difference in terms of reducing undesirable actions in the contact plan. This might be due to other parts of the model already performing optimizations that makes it hard for data to move in the way this heuristic prevents, and thus making it redundant. It does however affect the size of the state-space, but whether or not those parts of the state space being cut off by it was already pruned away is unknown.

Heuristic 3

Heuristic 3 was: *No batteries can go below the minimum limit*. This heuristic is slightly redundant in the current version of the model, at least insofar as is computable with the current version of CORA. The contact plan produced for 10 satellites found in Section 5.2 is very similar to the one produced by this experiment, so at least no measurable difference



```
simulate 1 [<=endTime]{sat0.b + sat0.a, sat1.b + sat1.a, sat2.b + sat2.a, sat3.b + sat3.a, sat4.b + sat4.a, sat5.b + sat5.a, sat6.b + sat6.a, sat7.b + sat7.a, sat8.b + sat8.a, sat9.b + sat9.a, 2770}
```

Figure 5.10: Graph of battery usage for a contact plan with heuristic 1 disabled.

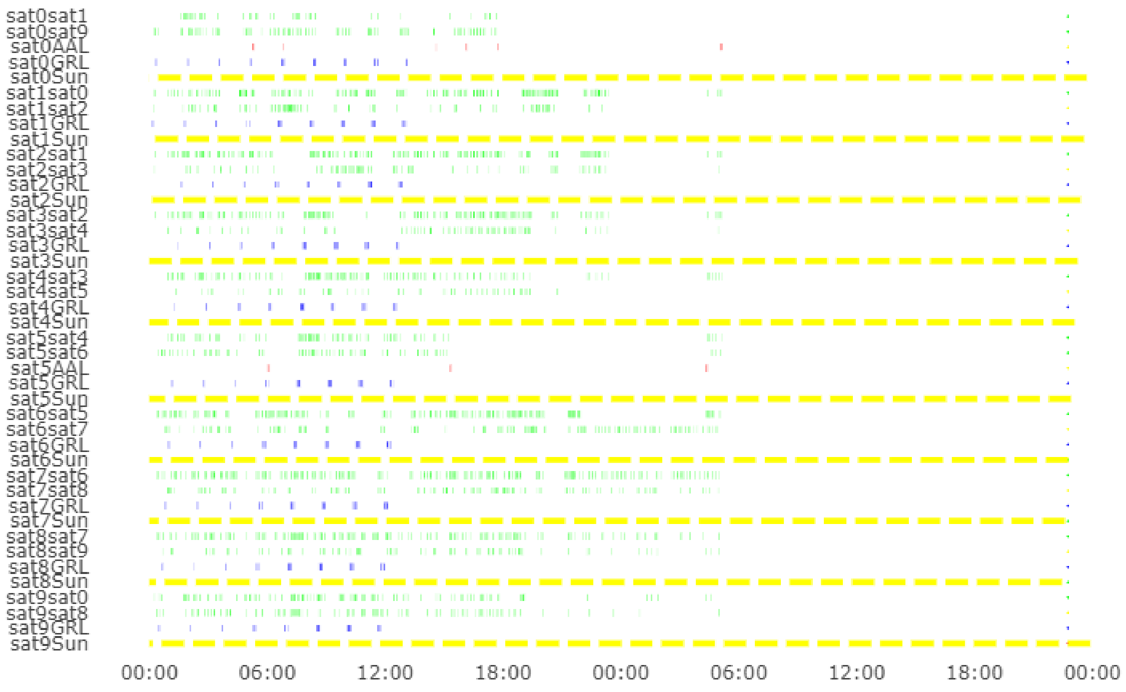


Figure 5.11: Heuristic 5

is present. While this experiment might not produce any battery levels that would have depleted a battery, and thus proving that the heuristic is useful, this is due to it being impossible to generate a contact plan with enough actions to actually deplete the batteries, due to the memory limitations of UPPAAL CORA.

Heuristic 5

Heuristic 5 was: *HSL satellites cannot send data to other satellites*. This heuristic had an impact on both the battery usage curve, and on the contact plan behavior. On the contact plan, seen on Figure 5.11, it is immediately obvious that a lot more actions has been taken per satellite, than on the chart for a contact plan with 10 satellites, found on A.3. Especially inter-satellite transmission actions are spread out over the chart instead of being grouped, as seen by the green lines on Figure A.3.

In addition to this, on Figure 5.12, it can be seen that the battery usage is also much more erratic than on the same amount of satellites with all heuristic enabled (Seen on Figure 5.4. Lastly, *Time to Collect* is 12 hours slower without heuristic 5, than it is with it enabled, and the cost is roughly doubled. These results imply that this heuristic is very important, especially for battery life and data-delivery times.

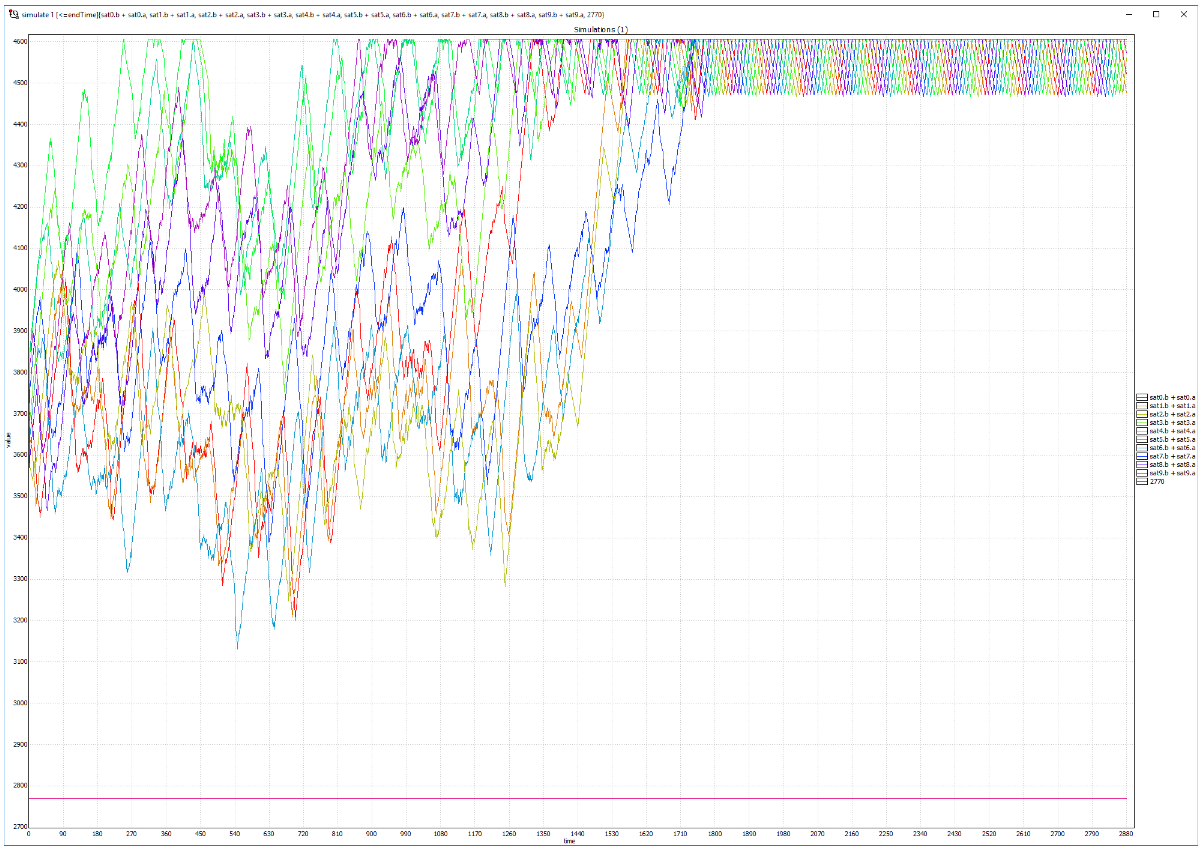


Figure 5.12: Battery graph for 10 satellites with heuristic 5 disabled.

Heuristic 6

Heuristic 6 was: *A satellite cannot take an action, if doing so would deplete the battery at the end of the period.* As with heuristic 3, found in Section 5.5, due to the limitations of CORA when generating traces with many transitions, the traces used in these experiments did not reach a low enough battery charge level for this heuristic to have a benefit. Thus it was not possible to generate a contact plan that actually depleted the battery enough to showcase this heuristic.

No Heuristics

The last experiment with heuristics was one completely without heuristics. In this case, no results were found, as UPPAAL CORA was not able to find any before running out of memory.

To conclude, most of the heuristics are vital for the usefulness of the model. They are successful in disabling unwanted behavior, which results in them helping to minimize state space.

5.6 Experiment 5: Cost

The objective of this experiment was to measure whether or not the cost-function provides a real benefit to the model. An experiment without battery-awareness, one without data-latency-awareness and one completely without cost was performed. Additionally, an experiment was made with a scaling cost, to see whether or not the scale of the different parts of the cost mattered to the end result. This is the cost-function, but without battery-awareness:

```
Int CurrentCost(){
    Int result;
    result += (goal - dataAtAAL);
    return (result >= 0 ? result : 0);
}
```

This is the cost-function, but without data-latency-awareness:

```
Int CurrentCost(){
    Int result;
    result += TotalMissingCharge();
    return (result >= 0 ? result : 0);
}
```

The final version, completely without cost, was performed by removing the cost from the corresponding edges in the Synchronizer, which can be found on Figure 5.1.

The impact of the cost is measured in 3 ways:

- Impact on battery behaviour
- Impact on data behaviour
- Impact on query-times

Table 5.6 contains the results of the four different experiments with different cost-functions, as well as the results from 5.2, in order to compare these results.

Results

The goal of this experiment was to see the effect of having a cost on the performance of the model, and the behaviour of the satellites. The following sections contains descriptions of each version of the experiment with different costs. As these experiments were directly related to the cost-function, the cost varies greatly from the remaining experiments. The experiment with no cost-function, even has no cost at all. The results can be seen on Table 5.6.

Exp #	No Battery	No Data	No Cost	Scaled data-cost
Time to collect all data	10:13	16:14	33:44	9:34
Time to deliver all data	15:20	29:58	38:29	15:20
Time to Trace	Instant	Instant	Instant	Instant
Cost	132066	4930116	0	148917

Table 5.6: The results of Experiment: Cost

No battery cost

In this experiment, the cost function was changed not to take battery usage into account. As seen on Figure 5.13, this allows for a slightly different behaviour in the initial parts of the trace, than what is allowed if battery is part of the cost function. In this version, collection of data starts earlier, which points to battery concerns being a reason as to why the contact plan on Figure A.3 starts the data collection later. The battery usage graph did not change overall, it only started draining the battery a short time earlier, in correspondence with the contact plan. A second difference that can be seen on Figure 5.13 is that in contrast to every

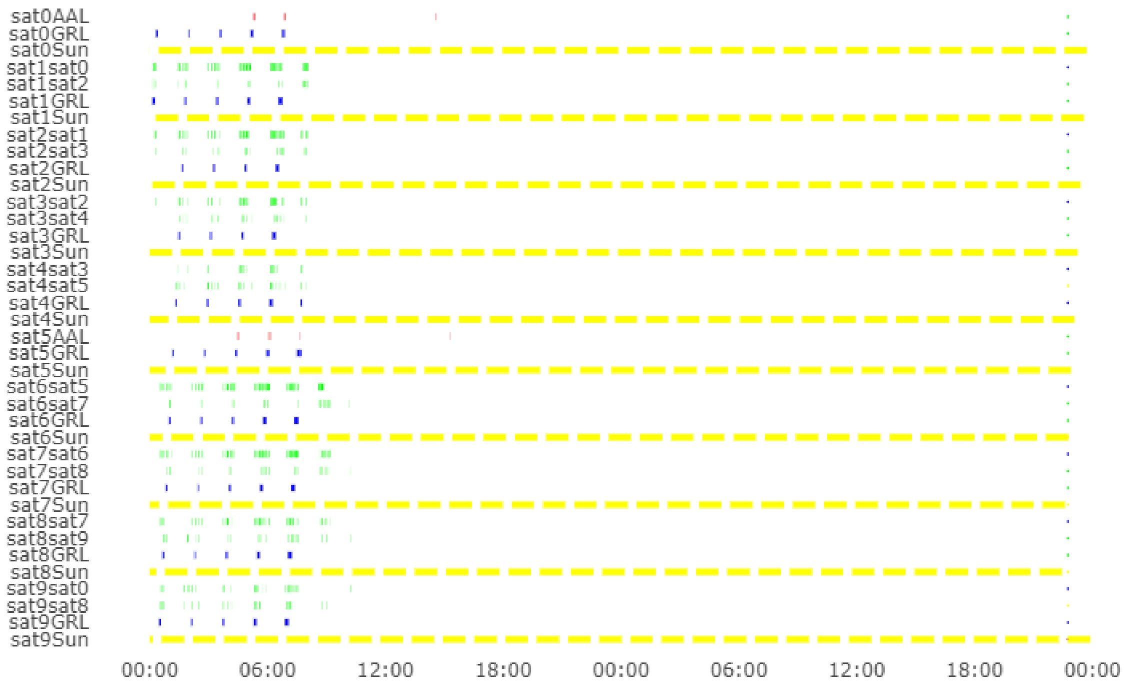


Figure 5.13: Contact plan for experiment using no battery cost in the cost function.

other contact plan, using no battery cost does not force the model to chop every action into smaller bits.

No data cost

In this experiment, the cost function was changed to not take data into account. This experiment did not point to any obvious changes in either the contact plan or the battery utilization. The only data point that did change, was that without data in the cost-function, an additional delivery to Aalborg was made at the end, as opposed to only one delivery at the end of the contact plan with all cost-functions added. This can be seen on figures 5.14 and A.3 respectively. These deliveries are marked by a red line towards the end of the contact plan.

No cost

In this experiment, no cost function was used at all, and no cost was used in the simulation. This, of course, invalidates the whole reason why we would use CORA, but it is a valid experiment nonetheless. The resulting contact plan can be seen on 5.15. The major difference not having a cost makes, outside of not providing the speedups in the verification process, is

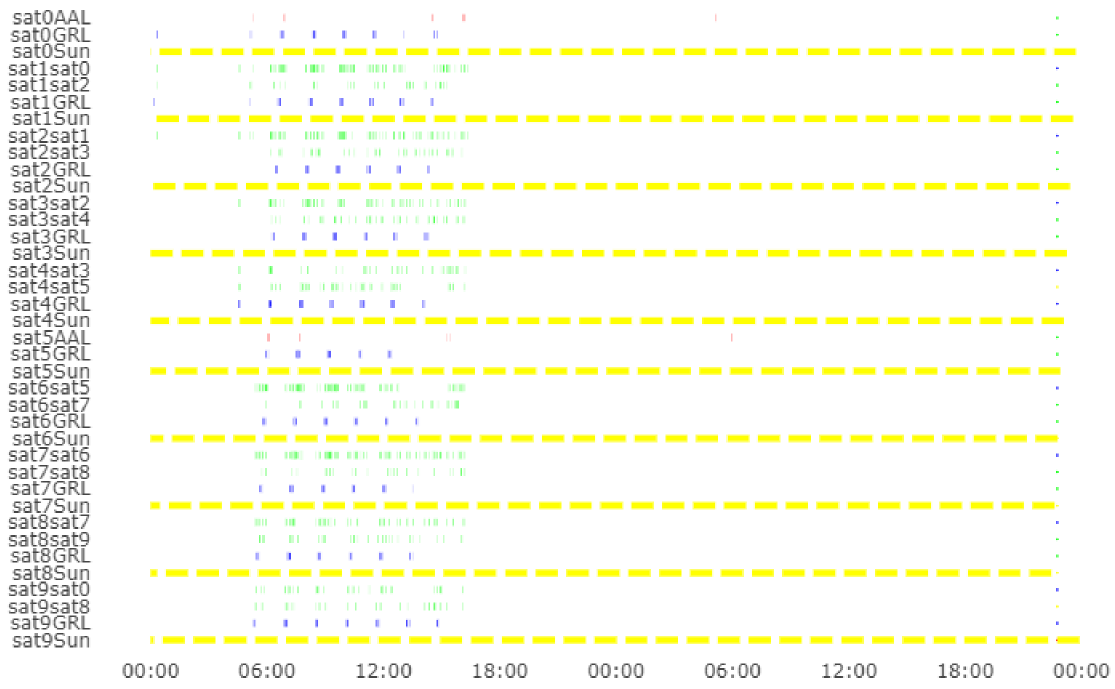


Figure 5.14: No data cost

in when the satellites decide to perform their actions. In addition to this, not having a cost also means that the model is not optimizing for anything - it does not care about delivering data fast, or using small amounts of battery, and will act accordingly. This is visible on Figure 5.15, where every type of action(blue, red and green lines) is spread out over a much larger part of the contact plan. In spite of this, the contact plan is not bad. This is because of the behavior disallowed by the different heuristics.

Cost scaling

In this experiment, the part of the cost function concerning data was scaled up by factors of 10, 100 and 1000. This was to see whether or not increasing this to be on a similar scale to the part of the cost-function about battery usage would help produce more balanced traces. The contact plans and Gantt-charts for scaling the data-cost by 10 and 100, did not produce enough of a difference in the results to determine whether or not it had an effect. When scaling by a 1000 instead, the results did show a difference. In order to run the experiment, every part of the cost had to be scaled back down by 1000 as well, or the cost would simply be too big of a number to work in CORA. The results show that the scaling has a big impact on when and how actions are taken. On the Gantt-chart found on Figure A.3, the actions are more spread out, than they are on the Gantt-chart for this experiment, found on Figure 5.16. On Figure 5.16, the actions are more concentrated towards the beginning, which is desirable

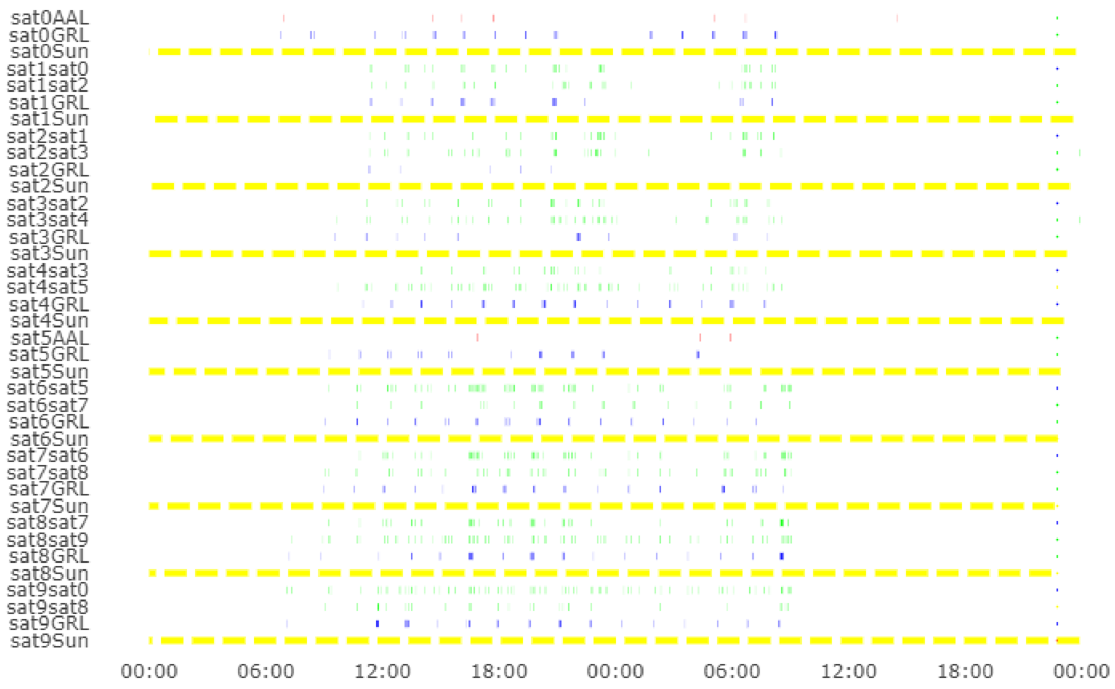


Figure 5.15: No cost

for having lower data-delivery times. This is also reflected in table 5.6, where the *Time to deliver* and *Time to collect* rows are slightly lower for this experiment than for the rest of the cost experiments.

5.7 Comparison with MILP

As the model in this paper is derived from the Ulloriaq case, and not directly from the MILP model in [12], a direct comparison is not possible. However, it is possible to compare the overall satellite behaviour, to see whether or not it behaves similarly. In [12], the satellites have certain periods where they do not perform any actions. A contact plan from [12] can be seen on Figure 5.17.

In the center part of Figure 5.17, there is a gap in communication, for both inter-satellite, and Greenland data pickup. This happens due to a large gap in the zones for connecting to Aalborg, of around 700 minutes. The same behaviour is exhibited by our model, as seen on Figure 5.7. Furthermore, both contact plans exhibit "chopping" behaviour, where activity is chopped into smaller bits, due to battery constraints.

Outside of the behaviour itself, the models rely on the same base data about when and where

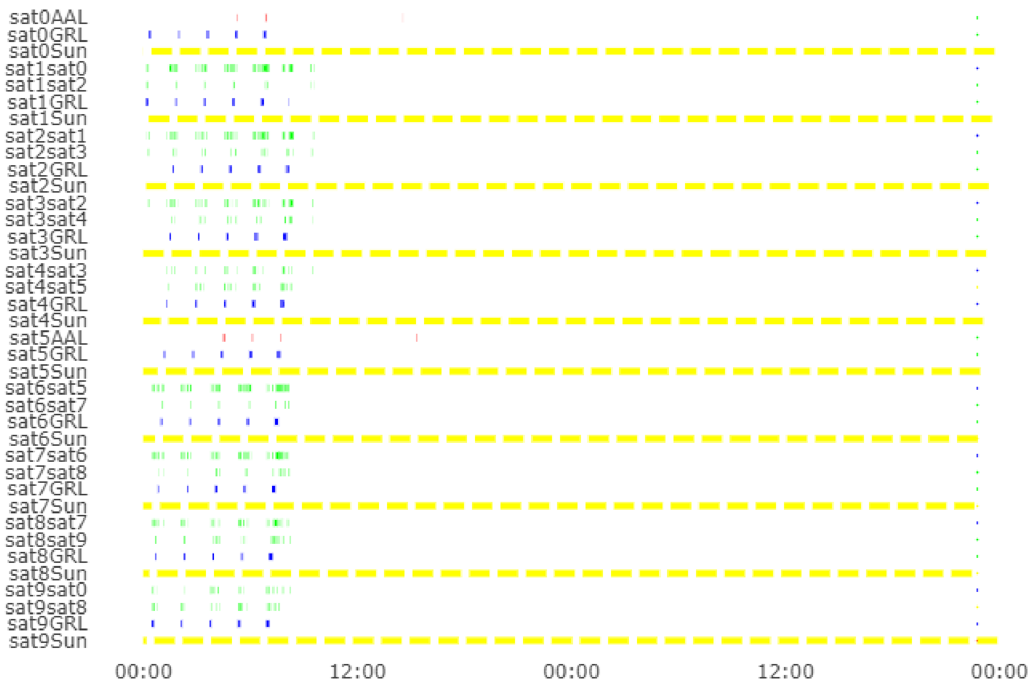


Figure 5.16: Data-cost scaled up by 1000, and both costs scaled down by 1000.

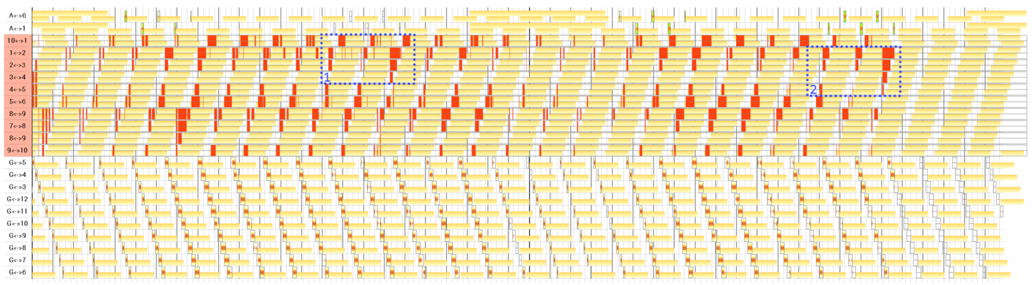


Figure 5.17: A battery aware contact plan from [12]

the satellites can perform actions, about the battery constraints, and about when they can charge. This is also reflected in when Aalborg downlinks are made, as they all happen at similar time points in both contact plans.

6 | Discussion

This chapter contains a discussion of the different parts of the project as a whole. Firstly potential sources of errors are discussed. After this, some suggestions for future work are presented. Lastly, the issues with UPPAAL CORA that the authors encountered during the project is described.

6.1 Sources of error

The following sections contains what we consider might be sources of potential errors in the model.

Many short contacts

In the final Gantt-charts, found throughout Chapter 5, most of the actions are very short (1-2 minutes) with a minutes delay in between.

This might be sub-optimal, but insofar as the authors have been able to determine, this happens due to how the battery part of the cost function is larger by a factor of somewhere between 10 and 100 than the data-cost. This is also corroborated by the results found on Figure 5.16, which has the data-factor scaled up by a factor of 1000, where the actions happen in more concentrated groups than they do on Figure 5.7. It could still present a problem, if this chopping behavior turns out to be sub-optimal. To determine this, it would, however, be necessary to get more domain-specific information to know whether or not this behavior is undesirable. This scale difference in cost, could also very well be a source of errors in the contact plans, as it might induce sub-optimal behavior.

The reason why the big emphasis on battery-cost leads to this behavior is that the contact plan is being optimized to have the least amount of total charge missing from the batteries each minute. Locally this is best achieved by taking an action (and draining the battery) and then delay to recharge right away. This will not give the best result for the complete contact plan, but because of the limitations with using UPPAAL CORA, described in Section 6.3, it is not possible to generate the best trace (cost-optimal).

Rounding errors

A second source of errors could be rounding errors. In the source case material, provided in [12], the battery constants are floating-point numbers. UPPAAL CORA however, only support the use of integers, and as such, it was necessary to round the constants. This is a source of imprecise contact plans. Another source of rounding errors could be that everything was scaled from working on a 1-second scale to a 1-minute scale, including the constraints provided by [12]. The reason why this was done, was due to the limitations of using UPPAAL CORA.

6.2 Future work

The following sections contain our recommendations for what could be done in the future regarding this project.

Optimization

A way of optimizing the model would be to obtain a version of UPPAAL CORA with a proper implementation of the meta-variable *remaining*. This variable, in theory, allows for the model developer to provide a lower bound on the cost of the remaining part of the trace. In the case of the *Ulloriaq Scheduling Model*, this could be constructed from how much data was left to deliver, and the current battery levels.

When provided with such an estimate, UPPAAL CORA can prune state-space more aggressively, according to the bound provided by *remaining*.

Explore other objectives

As it stands, the cost provided in *Ulloriaq Scheduling Model* is calculated from current battery charge levels, and the currently missing amount of data to reach the goal. In [12], the cost-function is an aggregation of battery costs and data traffic, increasing with time. Different cost functions, either closer or in entirely different directions than the ones we have tested, and the one used in [12] should be explored, when trying to apply different goals to this satellite constellation.

Number of satellites

To be able to utilize the *Ulloriaq Scheduling Model* for scheduling bigger constellation, the following two things are required:

- A reliable way to calculate the zones (Greenland, Aalborg, and sunlight) that the satellites move through during the period to be scheduled. The authors of [12] has a custom tool for this.
- A 64-bit implementation of UPPAAL CORA. As it stands, it is not possible to schedule for much longer or for more satellites than what has been described throughout this paper, with the current model, due to CORA running out of memory. Furthermore, there are several improvements that could be made to both CORA and UTAP, to ease the process.

With these things in place, it should be possible to schedule more satellites, and even enable scheduling of constellations such as the one SpaceX recently launched, called Starlink[21] with goals of many thousand networked satellites.

Tool-chain integration

For future use, it would be useful to integrate the tool-chain into a single tool. It would be ideal if it was possible to enter parameters into a simple interface, click a single button, and receive valid contact plans and battery statistics. Currently, the process is split into multiple parts which has to be fed with data manually. This is for multiple reasons, but specifically UTAP cannot work with a trace produced directly from the command-line of CORA. The reason for this is unknown. The fix for it is to open the model in the GUI, and re-save the trace, which then enables the trace to be turned into a model-independent trace by the command-line version of CORA. Furthermore, to the authors knowledge, it is not possible to run a simulation and extract graphs from UPPAAL SMC on the command-line as it stands, this would also have to be enabled for an automatic tool-chain to work.

6.3 UPPAAL CORA

UPPAAL CORA is a very useful tool in many different contexts, and for solving many different problems. There are however some issues, with CORA itself, and the tool UTAP. Some of them are due to age (The latest release is from 2006), but others are simply missing implementation, either in general, or missing implementation with respect to the main version of UPPAAL. The authors encountered the following issues during this project:

- UTAP being a standalone

  - It would make the process of using the tracer utility in UTAP much simpler if it was integrated into a release of UPPAAL CORA directly, either as an option on the command-line, or as an "Export Trace" option in the interface. The trace-files generated directly in CORA are model-dependent, probably due to large file sizes in the model-independent files, but having the option directly in CORA to produce traces model-independent, would be beneficial.
- Memory leaks
  - When generating traces with CORA right now, it seems that it is using up a lot of memory, without freeing any of it up again. This could be because of the large state-space being searched, but it could also very well be due to a memory leak. The memory consumption rises steadily until reaching around what a 32bit program can handle at the most, and then crashes.
- GUI Errors
  - Currently, the errors given by the GUI in CORA are of dubious help. Multiple times during this project, errors have been given that were either completely non-descriptive, or simply not understandable for the user - although the problem later turned out to be a problem the user had to fix in the model.
- Multithreading support
  - Currently, CORA does not utilize the full potential of multi-core systems. On the machine the experiments of this paper was simulated on, CORA used about 10% of the total CPU capacity. A cluster version of CORA, using Simple Priced Timed Automata, already exists, so it should be possible to adapt some of the existing algorithms to utilize these extra cores.
- Update CORA with current UPPAAL functionality
  - There is a number of updates to both the GUI and functionality of CORA that already exist for UPPAAL. These include a 64bit version, and would definitely make the process easier, and enable a model like the one presented in this paper to generate longer traces.
- Mismatch in settings
  - When using the command-line verifier for CORA, the search orders does not match those in the GUI. Secondly, the search order chosen is not the same listed by the verifier and on top of this, the behavior of the tool does not match either. This is a huge problem and makes it difficult to use, since the user is basically working in the dark.
- UTAP .if file size
  - When a custom size integer is defined in a CORA model, it takes up a disproportionally large amount of space in the intermediate format file generated by the command-line version of CORA. UTAP is not able to handle a file of this size(5 GB).

7 | Conclusion

Using a network of microsatellites to facilitate communication between two separate areas on earth, has requirements in the form of constraints which must be taken into account when scheduling their behavior for a mission, such as the Ulloriaq, to be considered a success. Two such constraints found, in particular for the Ulloriaq case was that battery is a limited resource and that real-time communication is rarely possible due to the nature of the constellation.

By creating a tool-chain, consisting of; UPPAAL CORA, UTAP, Ulloriaq Contact Plan Extractor and UPPAAL SMC, we attempted to create a complete solution to solve the scheduling problem related to the Ulloriaq constellation.

We modelled the problem as a Priced Timed Automata (PTA), using a store-carry-forward messaging scheme in the form of a Delay-Tolerant network. In the PTA, the satellite's battery were modelled using a linear battery model, which made it possible to generate a trace which was optimized for both battery usage and delivering data as fast as possible. This trace was then fed to UTAP [1] to generate a model-independent trace which could be fed to our custom tool *Ulloriaq Contact Plan Extractor*. The result of this is a contact plan for each satellite. A shortcoming of the contact plan generated from the previous steps is that the way the battery is modelled is naive compared to a real life battery. To verify whether or not the contact plan would deplete the satellite's batteries in a more realistic environment, an STA and UPPAAL SMC is used. The STA takes the contact plan as input and runs trough it with the more realistic Kinetic Battery Model. Furthermore, it exposes the contact plan to some uncertainties in regards to the battery behavior, by adding jitter to the battery cost of transmission and by varying the initial battery charges.

7.1 Results

The experiments done throughout this project were aimed at exploring the limitations of the model. The following areas were tested:

1. Number of satellites
2. Maximum amount of data collected
3. Simulation length
4. Impact of heuristics
5. Different cost expressions

The effects of having more satellites in the model was clear on the *Time to deliver* and *Time to collect* metrics found in Table 5.1. Additionally, adding HSL satellites provided more of a speedup, than adding a normal satellite did. In terms of model performance, adding more satellites was a trade-off with performance, as each added satellite scales up the complexity of the model and made it harder to generate a contact plan without running out of memory.

The effects of collecting more data overall, or collecting more data in shorter intervals, is mostly on the model performance. While some increases were made in collecting data (1100 data as opposed to 313 in the standard version of the model), further contact plan generation was quickly hindered by the experiment not being able to find a trace that collected more data. This was due to either memory limitations, UPPAAL CORA is 32bit, or due to the kind of trace we were searching for being too rare to practically obtain.

The heuristics implemented in the model all had an effect on the size of the state-space and thus helped when trying to obtain a trace. They work by disallowing undesirable actions, such as sending data back and forth or taking actions that would drain the battery. In addition to experiments with every heuristic, an experiment was made without any of the 5 heuristics enabled. This showcased that they do provide a speedup and that it is not even possible to find a trace without them.

The cost-function also has a big impact on how the final contact plan turns out. Experiments were made with different versions of the cost-function, as well as one completely without cost. This showcased the benefits of optimizing for a cost, as there was no real direction to the actions the model took without one.

Finally, a comparison was made with the MILP model presented in [12]. The comparison was somewhat tenuous, as the implementation here is more of an example of how to implement the Ulloriaq using Timed Automata and UPPAAL, rather than a reimplementation of the MILP model itself. The conclusion was that both models exhibit many of the same behavioral patterns, and the results for contact plans generated as closely as possible to the ones presented at the end of [12], are fairly similar. Due to limitations in UPPAAL CORA however, the model has not been tested to its full extent.

Overall, we were successful in modelling the constellation proposed for the Ulloriaq case using UPPAAL CORA and UPPAAL SMC. The Tool-chain designed for the purpose can take raw data concerning the involved satellites orbits and generate verified contact plans. The process is however not completely automated and still need some involvement. The results from our many experiments were successful and helped to increase our confidence in the model, especially when comparing to the results of [12].

Bibliography

[1] Marius Mikucionis Alexander David. Utap. `http://people.cs.aau.dk/~adavid/utap/`, "2007".

[2] Gerd Behrmann, Kim G Larsen, and Jacob I Rasmussen. Priced timed automata: Algorithms and applications. In *International Symposium on Formal Methods for Components and Objects*, pages 162–182. Springer, 2004.

[3] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. 32:34–40, 2005. ISSN 0163-5999. doi: 10.1145/1059816.1059823.

[4] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal 4.0 (2006). *URL http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial. pdf*, 2014.

[5] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[6] Python Software Foundation. Python 3. `https://www.python.org/`, "2019".

[7] J. A. Fraire. Introducing contact plan designer: A planning tool for dtn-based space-terrestrial networks. In *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 124–127, Sep. 2017. doi: 10.1109/SMC-IT.2017.28.

[8] Juan A Fraire and Jorge M Finochietto. Design challenges in contact plans for disruption-tolerant satellite networks. *IEEE Communications Magazine*, 53(5):163–169, 2015.

[9] GOMspace. Gomspace homepage. `https://gomspace.com/home.aspx`, "2019".

[10] GOMspace. gomx4. `https://gomspace.com/gomx-4.aspx`, "2019".

[11] Marijn R Jongerden and Boudewijn R Haverkort. Which battery model to use? *IET software*, 3(6):445–457, 2009.

[12] Holger Hermanns Kristian Bay Morten Bisgaard Juan A. Fraire, Gilles Nies. Battery-aware contact plan design for leo satellite constellations: The ulloriaq case study, "2018".

[13] August Kørvell and Kristoffer. Degn. Modelling cubesat constellations using timed automata and uppaal. 2019.

[14] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[15] Microsoft. C#. `https://docs.microsoft.com/en-us/dotnet/csharp/`, "2019".

[16] Gilles Nies, Marvin Stenger, Jan Krčál, Holger Hermanns, Morten Bisgaard, David Gerhardt, Boudewijn Haverkort, Marijn Jongerden, Kim G. Larsen, and Erik R. Wognsen. Mastering operational limitations of leo satellites - the gomx3 approach. 2016.

[17] Plotly. Plotly. `https://plot.ly/`, "2019".

[18] Alexander Schrijver. *Theory of linear and integer programming*. Wiley, 2011.

[19] AGI Software. stk. `https://agi.com/products`, "2019".

[20] AGI Software. stk. `http://people.cs.aau.dk/~adavid/cora/introduction.html`, "2019".

[21] SpaceX. Starlink mission. `https://www.spacex.com/sites/spacex/files/starlink_press_kit.pdf`, "2007".

[22] Uppsala University and Aalborg University. Uppaal.org. `http://www.uppal.org`, "2018".

[23] Erik Ramsgaard Wognsen, Rene Rydhof Hansen, and Kim Guldstrand Larsen. Battery-aware scheduling of mixed criticality systems. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 208–222. Springer, 2014.

A | Contact Plans

This section contains a visual gantt-chart version of every contact plan file generated as a result of the experiments ran in Chapter 5.

A.0.1 Number of satellites

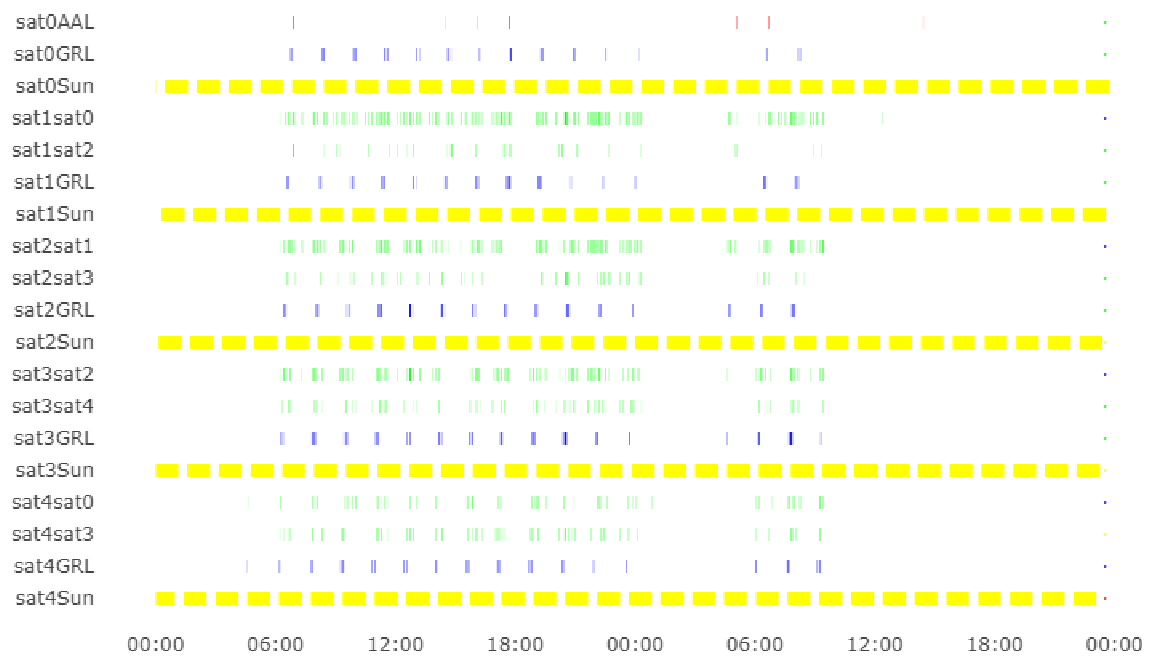


Figure A.1: 5 satellites, 1 HSL

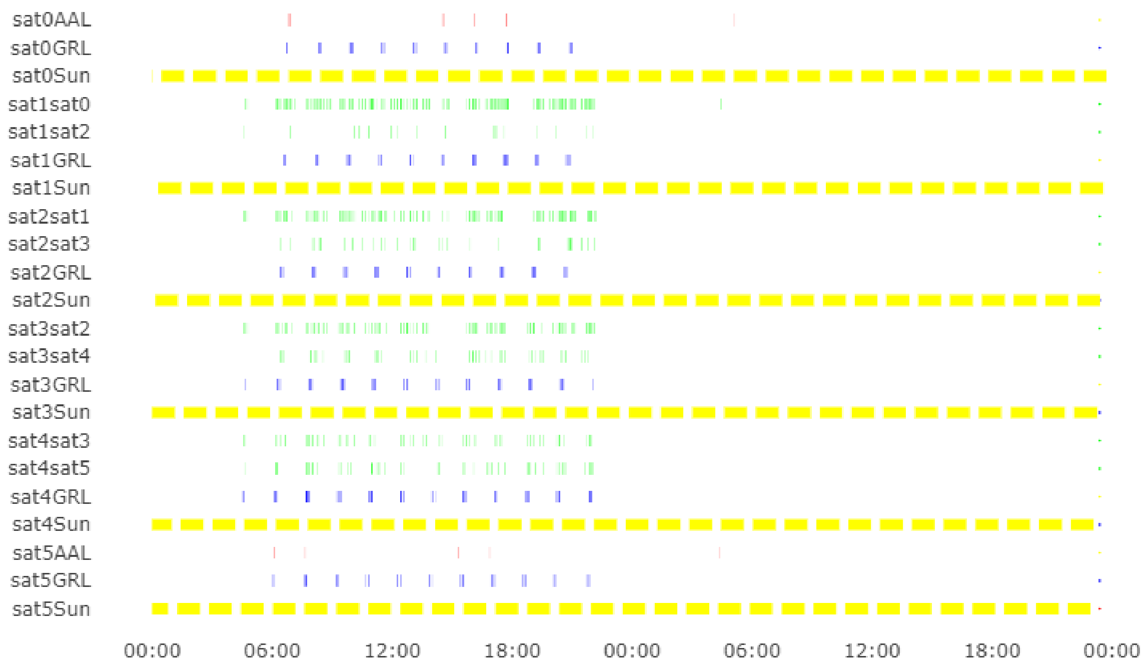


Figure A.2: 6 satellites, 2 HSL

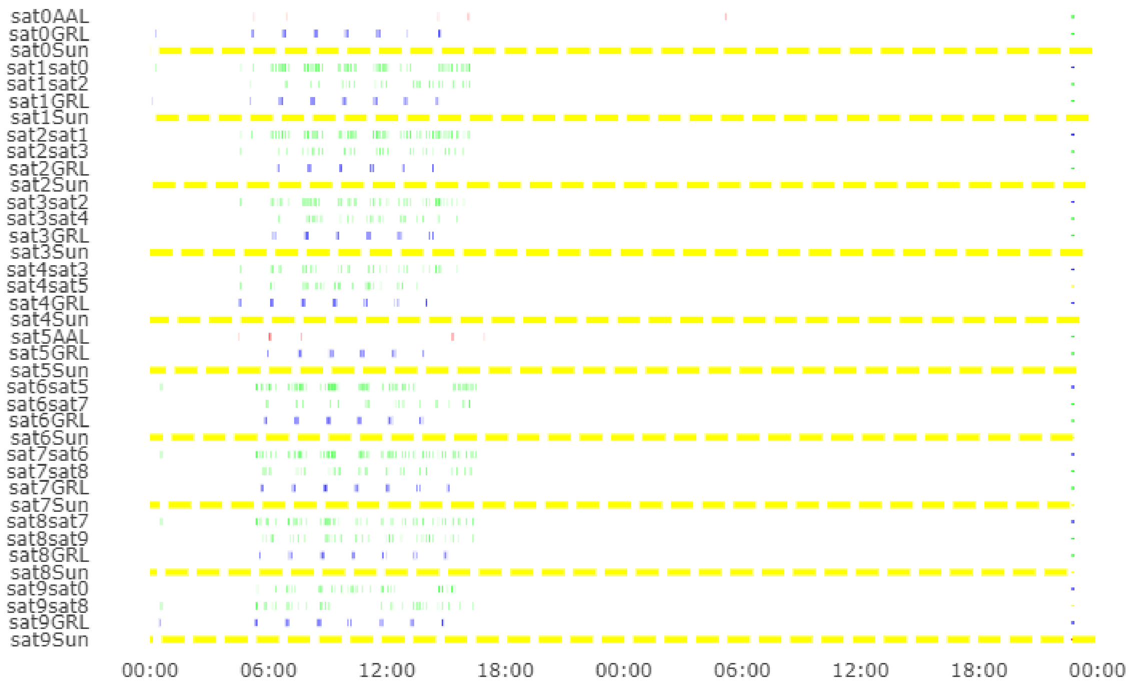


Figure A.3: 10 satellites, 2 HSL

A.0.2 Simulation Length

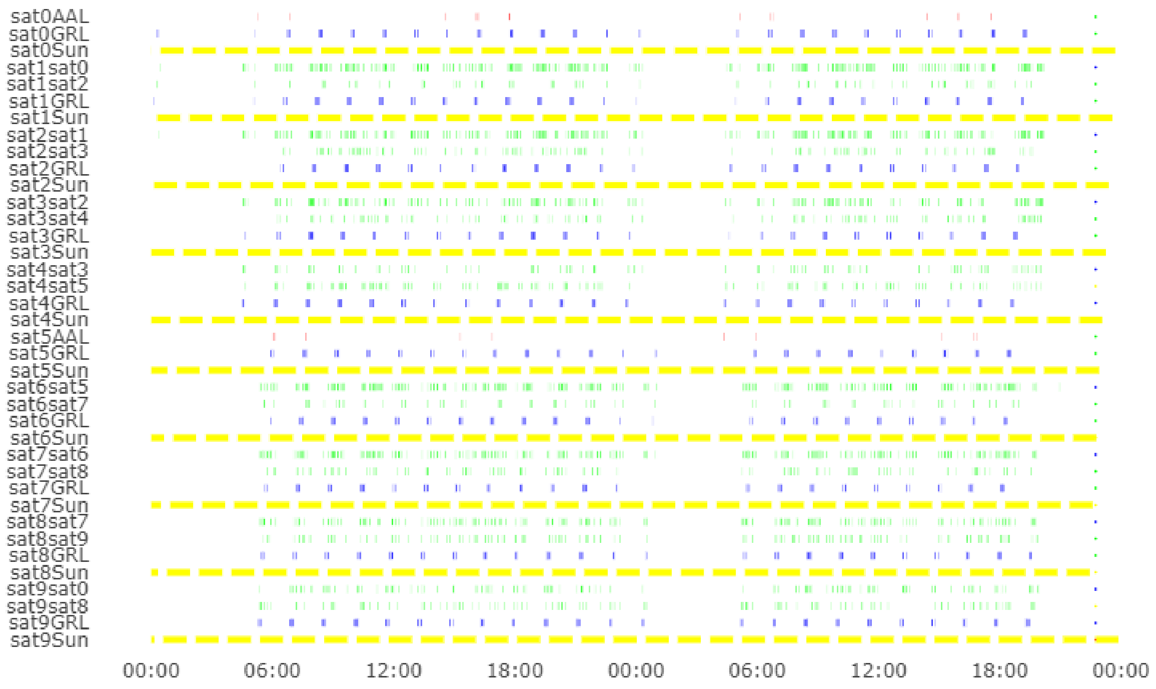


Figure A.4: 48 hour simulation

A.0.3 Heuristics

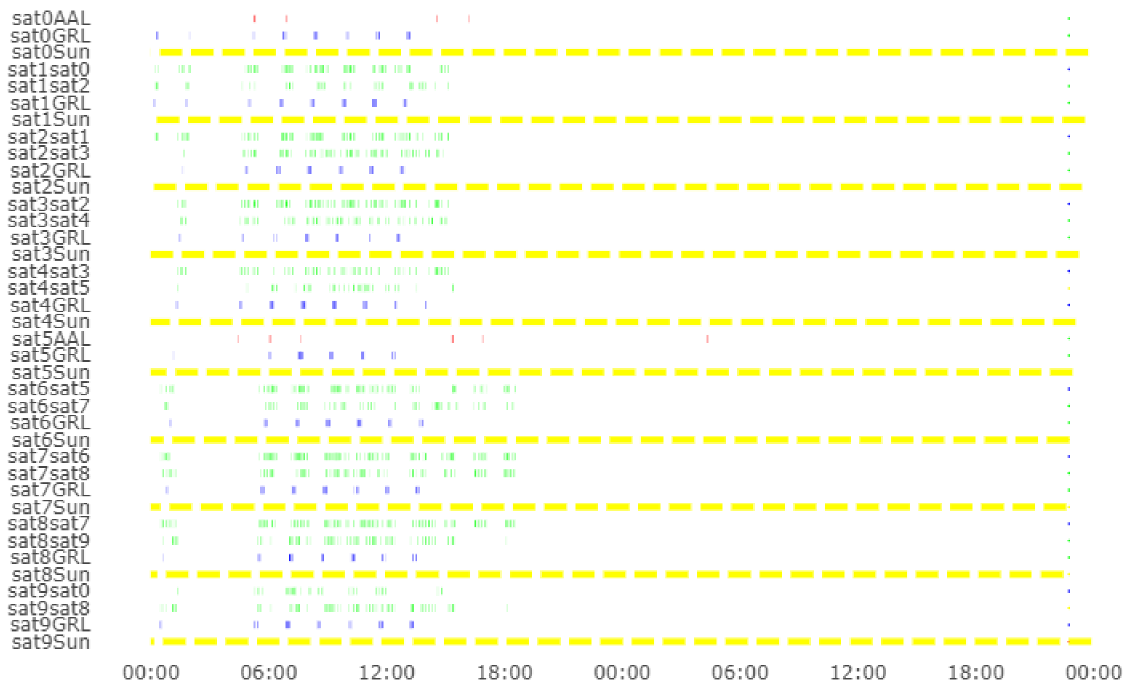


Figure A.5: Heuristic 1

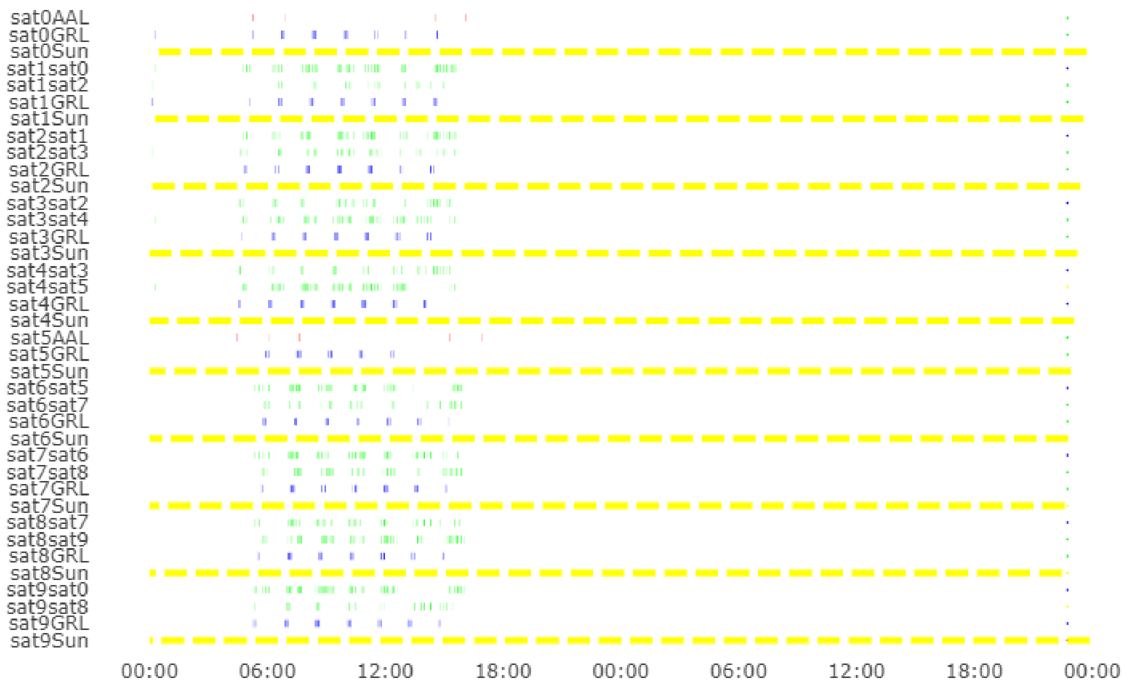


Figure A.6: Heuristic 2

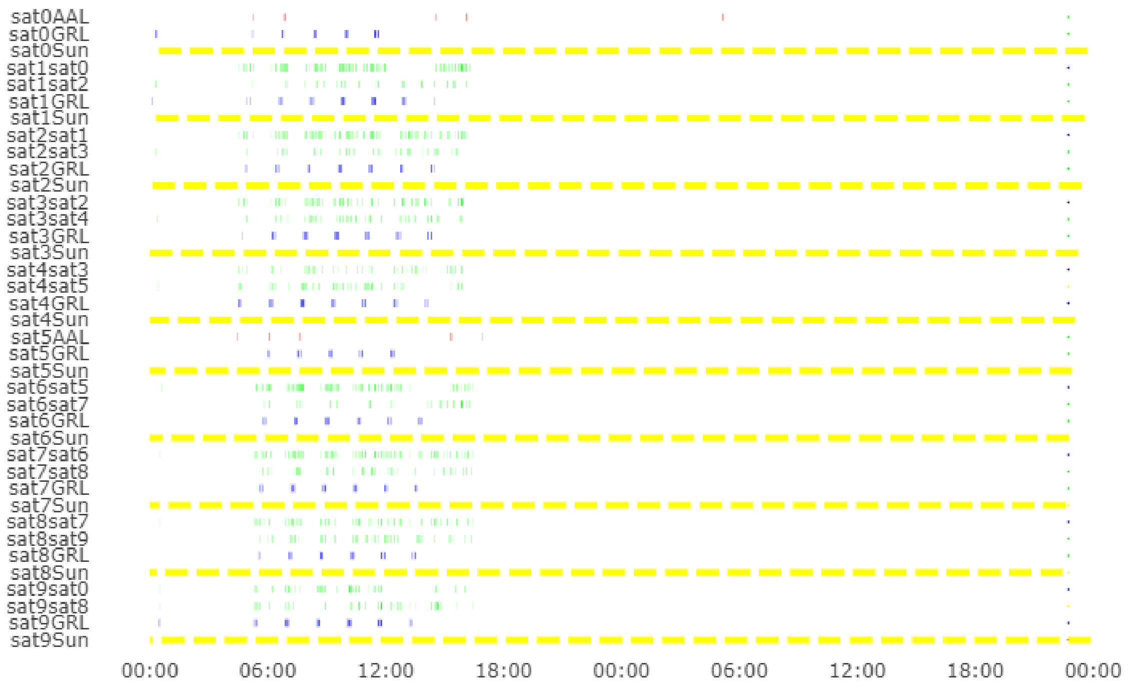


Figure A.7: Heuristic 3

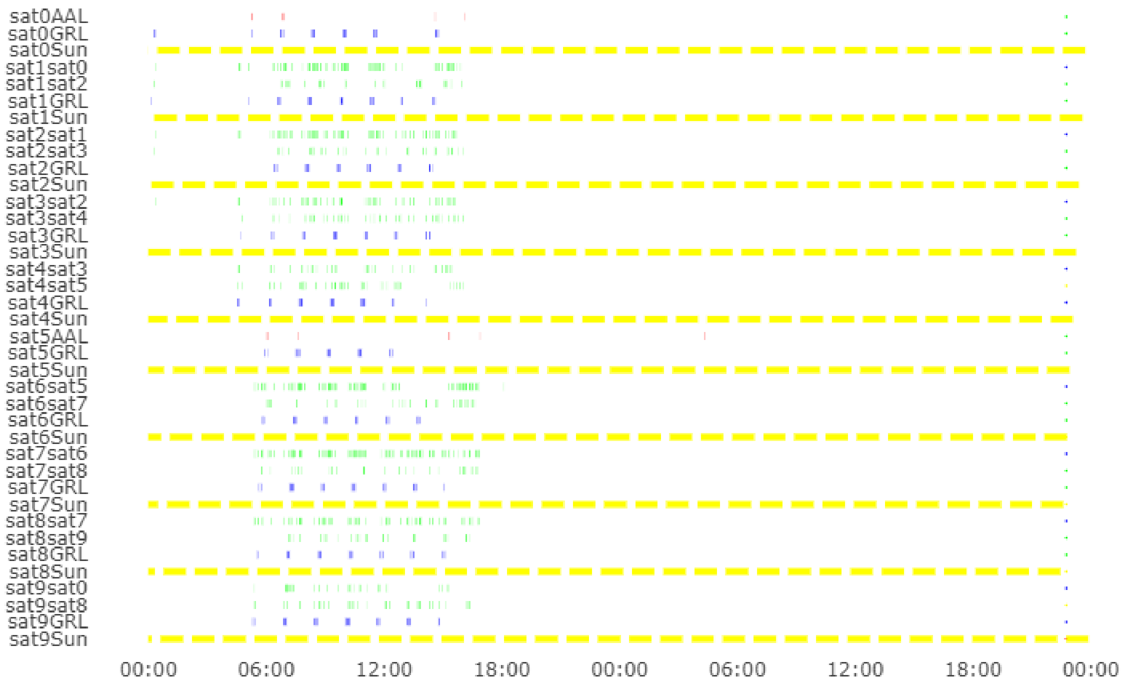


Figure A.8: Heuristic 6