Batteries Included 2.0

-Defining Semantics and implementing a Schedule Generator -

Master Thesis

Kristoffer Brodersen & Mads Nielsen/des104f17

Aalborg University
Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
http://www.aau.dk

Title:
Batteries Included 2.0

Theme:
Distributed, Embedded and Intelligent Systems

Project Period:
Spring Semester 2017

Project Group:
des104f17

Participant(s):
Kristoffer Brodersen
Mads Broen Nielsen

Supervisor(s):
Kim Guldstrand Larsen
René Rydhof Hansen

Copies: 1

Page Numbers: 101

Date of Completion:
June 2, 2017

Abstract:

This report expands on the Batteries Included [2] report, in which we describe a formalism (BATTCIO) for automatic generation of schedules for satellites. We further develop the BATTCIO formalism with semantics. A specification of a query language is presented, which could be used to place further restrictions on the generated schedules. We present an implementation of a schedule generator for BATTCIO, as well as examples of schedules generated using the tool. In addition, we will consider how such a tool might be appropriate for use-cases other than satellites. Finally we present the limitations of the tooling, as compared to more mature solutions.

0.1 Summary

In this project we have worked with implementation of a schedule generator for the formalism BATTCIO. We did so by first re-visiting the work done in the previous semester where we states the findings and work in general. The findings where regarding how a GomX-3 satellite works and what constraints that are applied to how it performs tasks. We also investigated battery models in general and chose to go with a discrete representation and a kinetic battery model. Lastly, regarding the batteries we also found a proposed wear score function that could help determine the wear on the battery based on the schedule generated. Then we presented the syntax and the conceptual GomX-3 model and how the model consists of a set of tasks, actions, components, opportunities, intervals, a battery and lastly start and end time. Then the Uppaal implementation was presented, as that would act as a basis for both the semantics and the implementation. After re-visiting the work in the previous report we gave the semantics for BATTCIO that would clearly define the behaviour of the model. The semantics entailed the rules for the four state transitions that happen in the model: Start a task, preempt a task, drop a task and transition in time. Where for start task we check component availability, if there is enough power, is the task within its opportunity, is the task locked and can it lock if need be, and lastly if the defined dependencies are resolved. For dropping a task the task simply needs to be defined as droppable in order to do so. And if it does it unlocks the tasks. For preempting a task we need to make sure that if preempted it can still finish if time passes one time unit. When time transitions we need to make sure that the battery does not deplete. That preempted tasks does not miss their opportunity, and if there is any component overlap.

With the semantics defined we defined a small concept query language to be used to generate more custom schedules. Here we can put constraints on the resulting schedule such as that a certain task must only complete a certain amount of times. Other constraints can be on the battery determining the lower bound of charge throughout the schedule or of we want the schedule to charge as much as possible or as little as possible. Thereafter we present the implementation of the parser, model, schedule generator in the language RUST. The gantt chart generator and code generator were developed in Python. The model implementation were aimed to entail the same behaviour as the semantics. When generating schedules we have applied two approaches. A greedy approach that simply aims to start as many tasks as possible and a branch and bound approach, where the upper bound was decremented every time a task was not started. The greedy approach quickly results in schedules however with no guarantees. The branch and bound is more time consuming but also produces schedules however the defined heuristic results in sub-optimal schedules for some systems. We further explored the which kind of systems the BATTCIO formalism could model and generated schedules for all of these small concept systems.

When generating schedules for these small systems the possibilities, limitations and behaviour of the model became more apparent.

Contents

Preface

Here is the preface. You should put your signatures at the end of the preface.

Aalborg University, June 2, 2017

Kristoffer Brodersen
kbrode11@student.aau.dk

Mads Broen Nielsen
madnie12@student.aau.dk

Chapter 1

Introduction

The art of ensuring that a certain plan or schedule is the best fitting for a given set of parameters can be a difficult task. It all depends on the amount of possible schedules that can be created. Which means that the higher amount of possible schedules would require for the person deriving these schedules to be exhaustive in his or her manner of generating these solutions in order to guarantee that the chosen solution is the best of all possibilities.

This has the risk of becoming a time consuming task if it were to be done manually. Dependent on the system that the given schedule is being generated for, the importance of achieving the best possible schedule may vary. In a system that is easily replaceable and relatively cheap one can argue that schedules that are *good enough* can be acceptable. However, in systems where the quality of the schedule is determining whether or not the system is worth using; the schedules must be of a certain standard. Therefore, if the ability to automate the process of deriving optimal schedules based on the specific system that are to perform the schedule was possible, it could greatly reduce the time used to create the schedule. If we look at satellites in general it could be argued that the better schedules that are derived and submitted to the satellite, the more value the satellite has. Not only does the construction of a satellite require large founds but placing the satellite in orbit is also an expensive part of the process. And once the satellite is in orbit, hands-on maintenance is very unlikely to occur. Some satellites, like the GomX-3 satellite has schedules uploaded to them periodically, that determines when it should do which tasks. Furthermore the satellite is a small satellite with a relatively small amount of power available and therefore it is vital for the satellite to have the right balance between performing tasks and recharging to avoid depleting the battery.

The modeling formalism BATTCIO tries to address the issue of generating possible schedules for a system that relies on a resource such as a battery. An attempt to implement a model of GomX-3 satellite based on the BATTCIO formalism has been done in Uppaal, however it exhausted the memory of a consumer laptop rather quickly resulting in the inability to generate schedules that are usable. However, a specialized schedule generator might be able to generate usable schedules by addressing the memory exhaustion and develop a model checker that takes it into consideration.

Therefore the problem statement is as follows:

- How can we implement a model and a schedule generator that can generate extended schedules for systems defined in the BATTCIO formalism on consumer hardware?

Chapter 2

Preliminaries

This chapter will, in an overview, go through the work done in the previous report. The previous report focused on designing the syntax of the formalism which was done based on the how the GomX-3 satellite was constructed and described in Batteries Included [2]. Furthermore, there was done some research as to which battery models would be best applicable to use in the formalism. Lastly we implemented the behavior of the model in the tool Uppaal in order to generate a trace that could be used to define a schedule with. A more detailed presentation of the findings in the previous report will be presented in the following sections.

2.1 GomX-3 Satellite

A starting point was taken from the GomX-3 satellite which is shown in figure 2.1. The GomX-3 satellite is a small satellite that measures $10 \times 10 \times 30$ centimeters. It is deployed in low earth orbit and has a set of components that it uses to perform certain tasks. The tasks that it has performed has varied over time [3], but at some point it was used to track airplanes over the Pacific and Atlantic sea. It tracked them, and if the planes were diverting from its original course, it was an indication of a storm. This is but one of the many different kind of missions that it has undertaken during its time. However, it is no longer in orbit as it entered the earths atmosphere the 4Th November 2016 [3].

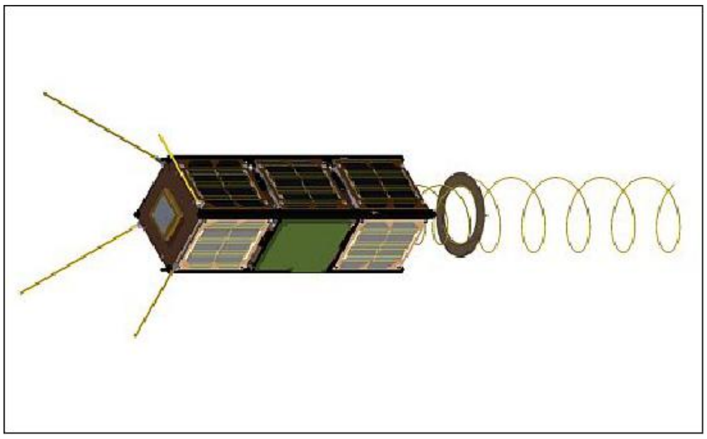


Figure 2.1: The GomX-3 satellite

We modelled the satellite in the BATTCIO formalism based on the article [1], where a thorough assessment of the satellite was performed. The satellite had:

- Three antennas
  - A Patch antenna
  - ADS-B antenna
  - UHF antenna
- A Gyroscope
- Solar Panels
- Battery

This is a coarse breakdown of the satellite as there are many more components in the satellite, however these components are the ones used in its tasks and defined in the article [1]. A task can be that the satellite must use one of its antennas to track an airplane. And in order to do so it must change its attitude both before and after. The changing of attitude, tracking, and then return to its original attitude all constitutes a task.

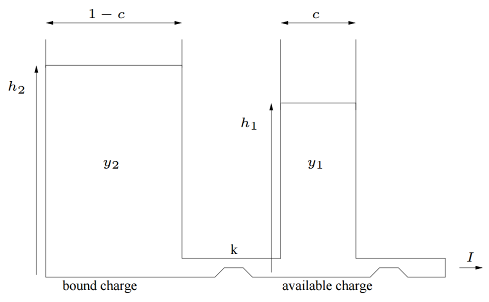
The current workflow of how schedules are derived for the satellite is that firstly an engineer performs the task of manually deriving a schedule. The engineer has a plan of when each task can be executed and then chooses which opportunities will be taken when. This results in a schedule. This schedule is then verified through a piece of software that determines the chance of depleting the battery based on the derived schedule. If the schedule is considered good enough it is deployed to the satellite for execution [1].

This was where we presuppose that a model encompassing the behavior of the satellite and its environment could improve the workflow for the engineer by generating schedules that could then be verified through their verification software.

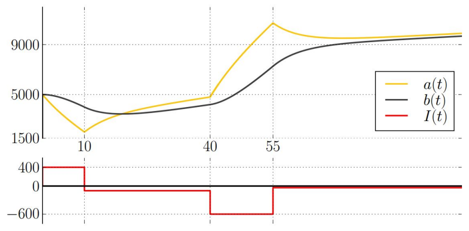
2.2 Battery Inquiry

A satellite such as a GomX-3 satellite has a limited power resource, meaning that it can deplete it if the tasks that it is performing are not managed correctly. Therefore, we chose to investigate battery models in general in order to be able to have an as accurate representation of the battery as possible. Among the investigated battery models we chose to go forward with a discrete representation of the battery, which can be meaningful in well understood systems where if you have very precise readings of how the battery is effected, it can be sufficient. However, the problem with a discrete model is that is does not encompass the recovery effect [4]. In short, a battery recovers some power as time passes, meaning that in an actual battery there is more power over time, compared to a discrete representation.

The other model that we chose to to forward with was the Kinetic Battery Model(KiBaM) which does encompasses the recovery effect[4]. The drawback of this model is that it needs more parameters to define its behaviour compared to the discrete representation. The Kinetic Battery model is viewed as the battery consisting of two wells, where there is power in each of them. One well encompasses the *available* charge and the other has the *bound* charge as shown in figure 2.2a.



(a) The Kinetic Battery Model visualized



(b) A graph showing the recovery effect

Figure 2.2: The Kinetic Battery Model and a graph showing the recovery effect[1][4]

The power that is directly available will be drawn from the available charge. As described the KiBaM encompasses the recovery effect by the notion that the two wells always goes toward equilibrium. So when power is drawn from the available charge, power runs from the bound to the available charge. This is shown in figure 2.2b where the available charge *a(t)* reacts to the load, *I(t)* in a direct manner, whereas the bound charge *b(t)* reacts to the charge in *a(t)* more slowly. The rate in which this happens entirely depends on the specifications of the battery.

Furthermore, we aimed to be able to determine from similar schedules which of the schedules has the least impact on the longevity of the battery. In order to do this we would make use of a proposed wear score function(wsf) that makes use of a

Fast Fourier transformation on the state of charge(soc) over time to get a score that determines the wear of the battery [8]. This function is useful in the situation where a set of schedules are generated that are similar in the amount of tasks it completes. Then the score of these schedules can be compared and then the schedule with the lowest score function can be chosen.

$$wsf(S,f) = \frac{2f}{n} \sum_{i=0}^{\lfloor {}^{n}/_{1} \rfloor} i|F(S)_i|^2 \tag{2.1}$$

Equation 2.1 shows the wear score function where S is the set of state of charge samples, taken at frequency f and n is the number of samples taken. *F(S)* is the Fast Fourier transform function on the set *S*. This equation results in a number that in itself does not indicate much. However, compared with other numbers generated from the same system it can allow for comparison and hereby considerations about the impact of the longevity of the battery. The lower the score, the lower the wear on the battery is.

2.3 BATTCIO Formalism

With the gained knowledge of the GomX-3 satellite we derived a formalism wherein the GomX-3 satellite and its environment can be modelled. We named the formalism BATTCIO, based on the fact that such a model consists of:

- A **B**attery
- A set of **A**ctions
- A set of **T**asks
- A set **T**imeframe
- A set of **C**omponents
- A set of **I**ntervals
- A set of **O**pportunities

With this segregation of systems we designed the syntax, that can be seen in appendix A.2. With the defined syntax we modelled a conceptual GomX-3 satellite as can be seen in Listing 2.1.

Listing 2.1 : The conceptual model of a GomX-3 satellite

```
Component ProcessorOne(2);
Component ProcessorTwo(3);
Component Gyroscope(10);
Component LBand(5);
Component SolarPanel(50);
Component XBand(3);

Action Slew(Components: {Gyroscope} Duration: 2);
Action Track(Components: {LBand, ProcessorOne} Duration: 2);
Action Communicate(Components: {XBand, ProcessorOne} Duration: 1);
Action Charge(Components:{SolarPanel} Duration: 5);
Action Calculate(Components:{ProcessorOne} Duration: 5
                | Components:{ProcessorTwo} Duration: 5);


Task Send(Actions: [Slew, Communicate, Slew]);
Task Receive(Actions: [Slew, Communicate, Slew]);
Task Track(Actions: [Slew, Track, Slew]);
Task Charge(Actions: [Charge]
            Locks:[Send, Receive, Track, Calculate]
            Droppable: True
            Preemptable: True);
Task Calculate(Actions: [Calculate]);

Interval SendReceive(10, 15);
Interval Track(30,35);
Interval Charge(15,40);
Interval Calculate(10,15);

Opportunity(Intervals: SendReceive
            Task: Send
            Dependencies: Track: 1);
Opportunity(Intervals: SendReceive
            Task: Receive);
Opportunity(Intervals: Track
            Task: Track
            Dependencies: Receive: 1);
Opportunity(Intervals: Charge
            Task: Charge);
Opportunity(Intervals: Calculate
            Task: Calculate);


Battery(Capacity: 400
        InitialCharge: 400
        Type: Discrete);

Start(0);
Termination(40);
```

A model consists of a set of components that makes use of the battery i.e. they drain power from a battery. In our concept model it is for instance the component *LBand*. These components are used in one or more actions. An action is the use of one or more components simultaneously in a specified amount of time. In the listing it can be seen from the action *communicate* that uses processor one and the lband antenna components respectively.

This action can be a part of task where more actions are executed sequentially. For instance in the task *track* we can derive that it first slews, then tracks, then slews back.

This is how we define the internal behaviour of the system, however a BATTCIO model is also dependent on some external factors that determines when tasks can be executed. In order to accommodate this we have defined opportunities and intervals.

These opportunities are defined as well in the model by a given interval. There can be one or more opportunities defined for a task. The interval must be within the time frame of the "mission".

In the series of opportunity declarations as can be seen in the listing we see that we can add intervals to it, define which task it is bound to and define dependencies between tasks. Second to last we define the battery along with its maximum capacity and starting state of charge. And lastly the time for the entire period of the system is defined in the *Start* and *End* constructs.

This formalism can describe a GomX-3 satellite along with its environment because we are able to describe the components, their usages in tasks, along with when they can perform certain tasks. So, to say that the formalism can model the satellite is not fulfilling as it also models its environment in regards of when it can perform its tasks.

2.4 Uppaal Implementation

With the knowledge of the satellite gained and a model of the satellite derived we implemented the system in Uppaal. This implementation encompassed the behaviour of the model and therefore we could use to find out if schedules where able to be generated based on the configuration of the model. The models will be presented, however not fully detailed. The in-depth presentation of these can be found in [2].

Overall we defined an automaton for each of the language constructs i.e. components, actions, tasks, opportunities. Intervals we did not make an automaton for as it encompassed no real behaviour, only data in the form of a two numbers. Then we added some helper automata i.e. the Timer in which we also defined the battery, the scheduler and an orbiter that is very specific to this system.

Figure 2.3 shows the model of a component where it can be either active or idle. When activated it adds the cost to the current load of the battery and when deactivated it subtracts the cost.

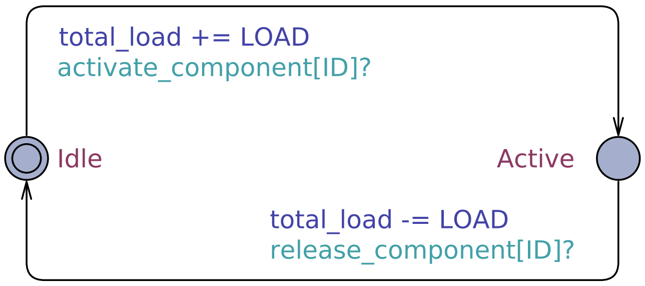


Figure 2.3: Simplified template for a component

Figure 2.4 shows how an action is defined. It starts it in the waiting location and from there can be prompted to transition to the committed location where it can activate its components. If there is an alternate set of components that can be activated, those can be done as well. This ensures that if a an action wants to start a component that is already running, it can't hereby preventing the task from starting. From there it can transition to the running location where it resides until it reaches the end of its duration. It can also be both preempted and dropped from this location.

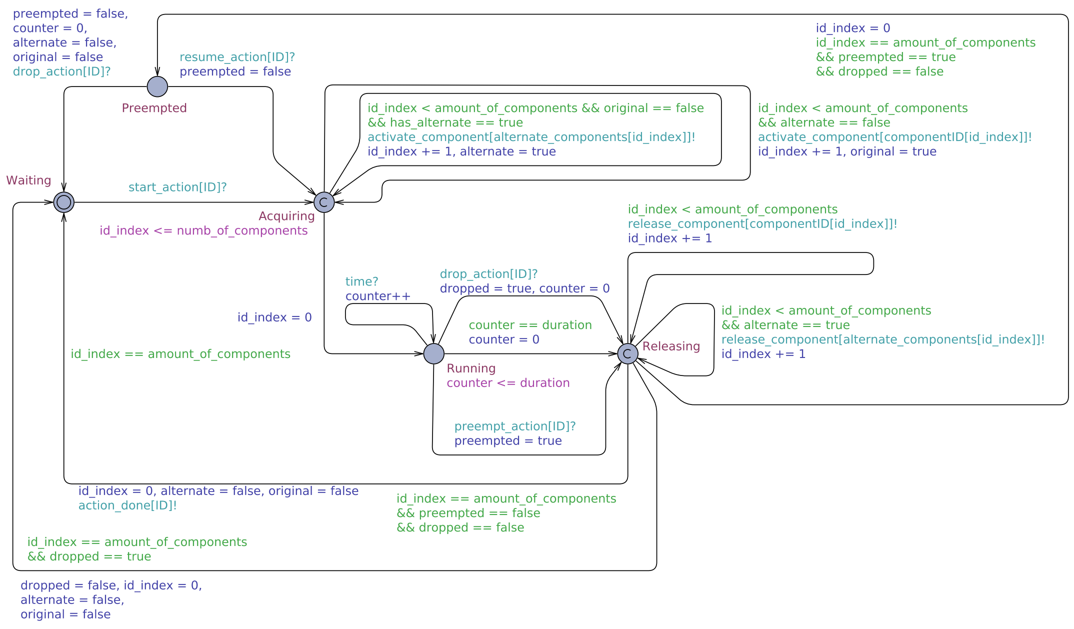

preempted = false,
counter = 0,
alternate = false,
original = false
drop_action[ID]?
resume_action[ID]?
preempted = false
Preempted
id_index = 0
id_index == amount_of_components
&& preempted == true
&& dropped == false
id_index < amount_of_components && original == false
&& has_alternate == true
activate_component[alternate_components[id_index]]!
id_index += 1, alternate = true
id_index < amount_of_components
&& alternate == false
activate_component[componentID[id_index]]!
id_index += 1, original = true
Waiting
start_action[ID]?
Acquiring
id_index <= numb_of_components
id_index < amount_of_components
release_component[componentID[id_index]]!
id_index += 1
time?
counter++
drop_action[ID]?
dropped = true, counter = 0
id_index = 0
counter == duration
counter = 0
id_index < amount_of_components
&& alternate == true
release_component[alternate_components[id_index]]!
id_index += 1
Releasing
id_index == amount_of_components
Running
counter <= duration
preempt_action[ID]?
preempted = true
id_index = 0, alternate = false, original = false
action_done[ID]!
id_index == amount_of_components
&& preempted == false
&& dropped == false
id_index == amount_of_components
&& dropped == true
dropped = false, id_index = 0,
alternate = false,
original = false

Figure 2.4: Template for an action

Figure 2.5 shows the template for a task. It starts in its initial location idle. A task can be locked when waiting; preventing it from being prompted to start. Once the task is within its opportunity it can be started. If there are no opportunities bound to the task it can be prompted to start at any given point. If prompted to start it first locks the tasks that needs to be locked and then activates the first action defined for it and from there resides in the running location. In the running location it can either be preempted, dropped, finish or start the next action in the sequence.

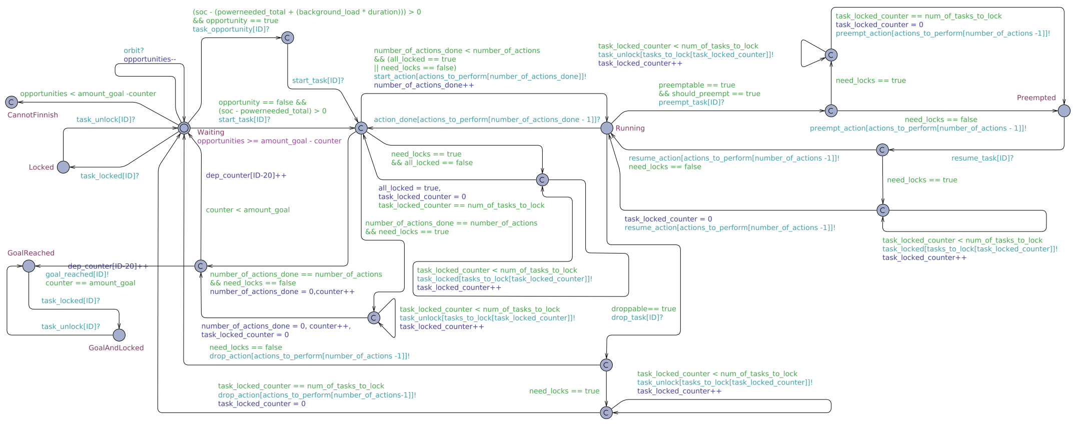

(soc - (powerneeded_total + (background_load * duration))) > 0
&& opportunity == true
task_opportunity[ID]?
orbit?
opportunities--
opportunities < amount_goal -counter
CannotFinnish
task_unlock[ID]?
Waiting
opportunities >= amount_goal - counter
Locked
task_locked[ID]?
start_task[ID]?
opportunity == false &&
(soc - powerneeded_total) > 0
start_task[ID]?
dep_counter[ID-20]++
counter < amount_goal
number_of_actions_done < number_of_actions
&& (all_locked == true
|| need_locks == false)
start_action[actions_to_perform[number_of_actions_done]]!
number_of_actions_done++
action_done[actions_to_perform[number_of_actions_done - 1]]?
need_locks == true
&& all_locked == false
all_locked = true,
task_locked_counter = 0
task_locked_counter == num_of_tasks_to_lock
number_of_actions_done == number_of_actions
&& need_locks == true
task_locked_counter < num_of_tasks_to_lock
task_locked[tasks_to_lock[task_locked_counter]]!
task_locked_counter++
Running
task_locked_counter < num_of_tasks_to_lock
task_unlock[tasks_to_lock[task_locked_counter]]!
task_locked_counter++
preemptable == true
&& should_preempt == true
preempt_task[ID]?
task_locked_counter == num_of_tasks_to_lock
task_locked_counter = 0
preempt_action[actions_to_perform[number_of_actions -1]]!
need_locks == true
Preempted
need_locks == false
preempt_action[actions_to_perform[number_of_actions - 1]]!
resume_action[actions_to_perform[number_of_actions -1]]!
need_locks == false
resume_task[ID]?
need_locks == true
task_locked_counter = 0
resume_action[actions_to_perform[number_of_actions -1]]!
task_locked_counter < num_of_tasks_to_lock
task_locked[tasks_to_lock[task_locked_counter]]!
task_locked_counter++
GoalReached
dep_counter[ID-20]++
goal_reached[ID]!
counter == amount_goal
task_locked[ID]?
task_unlock[ID]?
GoalAndLocked
number_of_actions_done == number_of_actions
&& need_locks == false
number_of_actions_done = 0,counter++
task_locked_counter < num_of_tasks_to_lock
task_unlock[tasks_to_lock[task_locked_counter]]!
task_locked_counter++
number_of_actions_done = 0, counter++,
task_locked_counter = 0
droppable== true
drop_task[ID]?
need_locks == false
drop_action[actions_to_perform[number_of_actions -1]]!
task_locked_counter == num_of_tasks_to_lock
drop_action[actions_to_perform[number_of_actions-1]]!
task_locked_counter = 0
need_locks == true
task_locked_counter < num_of_tasks_to_lock
task_unlock[tasks_to_lock[task_locked_counter]]!
task_locked_counter++

Figure 2.5: Template for a task

Figure 2.6 shows the opportunity template. From its initial location it can synchronize over the channel task_opportunity if it is within the opportunity. An implementation specific construct can be seen where it synchronizes on the goal_reached channel if the amount of specified orbits has been completed. When there is a new orbit, it transitions back to the the initial location.

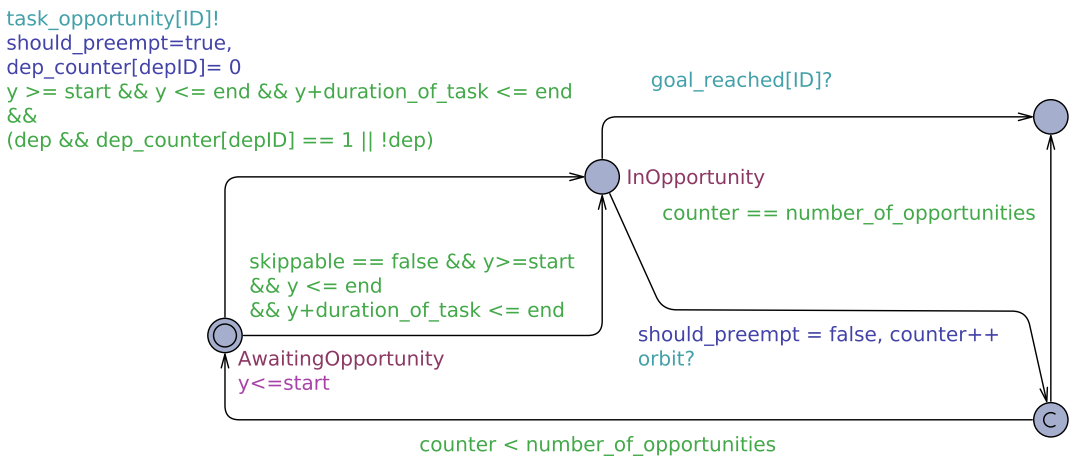

task_opportunity[ID]!
should_preempt=true,
dep_counter[depID]= 0
y >= start && y <= end && y+duration_of_task <= end
&&
(dep && dep_counter[depID] == 1 || !dep)
goal_reached[ID]?
InOpportunity
counter == number_of_opportunities
skippable == false && y>=start
&& y <= end
&& y+duration_of_task <= end
AwaitingOpportunity
y<=start
should_preempt = false, counter++
orbit?
counter < number_of_opportunities

Figure 2.6: The template for an opportunity

The template responsible for the time transitions and the battery is defined is shown in the figure 2.7.

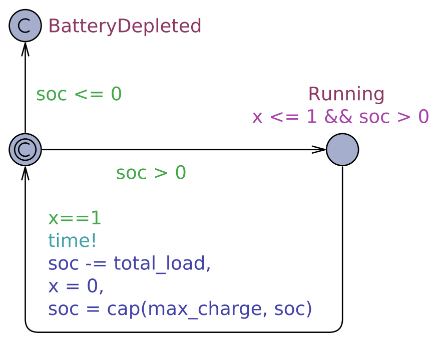


Figure 2.7: The template for the timer

The second to last figure 2.8 shows the scheduler, which again, is implementation specific to the GomX-3 system. It makes a non-deterministic choice between starting, preempting and dropping tasks, and time transition. It has been altered from the previous report, however maintains the same behaviour.

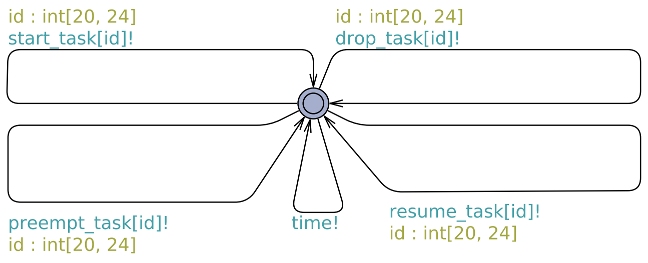


Figure 2.8: Template for the scheduler

The last figure 2.9 shows the orbiter which is responsible for resetting timers once an orbit has passed.

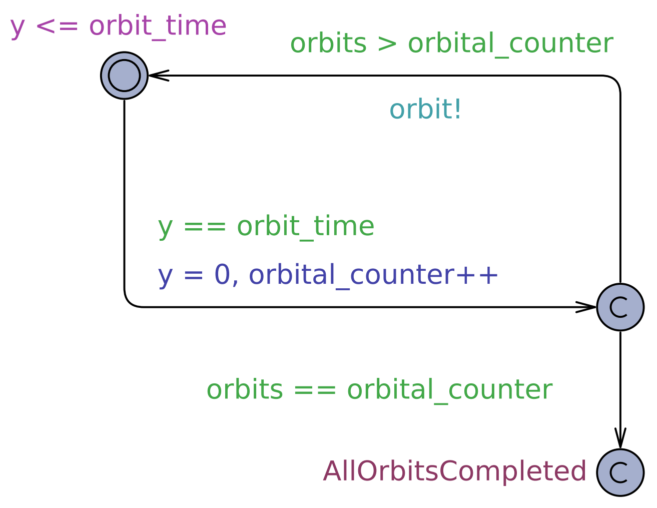


Figure 2.9: The template for the orbiter

With the conceptual GomX-3 model we made the following query:

$$\begin{aligned} &E <> TaskReceive.GoalReached \quad \&\& \quad TaskTrack.GoalReached \\ &\&\& \quad TaskSend.GoalReached \quad \&\& \quad TaskCalculate.GoalReached \end{aligned} \tag{2.2}$$

In the system definition within Uppaal we specified both the amount of completions needed to complete and the duration of the mission.

To begin with we started with a short time frame for the mission and incrementally expanded the time until we reached a maximum threshold where the query would exhausted the memory available to us.

We were able to schedule for 11 orbits, where each orbit takes approximately 40 minutes. The computer that the system was being verified on ran out of memory if we tried to verify a schedule for more than 11 orbits ahead in time. The results in the end where as shown in table 2.1.

Table 2.1: Results, from the previous report [2]

Orbits	Nr. of completions each	Time	Memory
3	1	1.023 s	25.900 KB
5	2	24.292 s	406.316 KB
7	3	81.494 s	1.266.984 KB
9	4	112.210 s	3.107.544 KB
11	5	205.620 s	6.146.136 KB
13	6	Out of memory	Out of memory

The results led us to the conclusion that it is indeed possible to generate schedules, however measures needed to be taken in order to extent the time frame for which we can generate them.

Chapter 3

BATTCIO Semantics

This chapter will present the semantics that will define a clear behaviour of the model. Furthermore, the semantics will aid us in the implementation of the schedule generator as we will have a clear understanding of how the model should behave.

A BATTCIO system consists of a series of sets:

- $C \in Component = CompName \times cost$
- $A \in Action = ActionName \times P(Comp) \times Duration$
- $T \in Task = TaskName \times Actions^* \times Bool \times Bool \times P(\mathbb{N} \times \mathbb{N}) \times P(Task \times \mathbb{N}) \times P(Task)$
- $Battery = Cap \times InitialCharge$
- $End = \mathbb{N}$

Component is the set of components in the system, which consists of an identifier and a cost. An action consists of an identifier along with a set of components and a time, that determines the duration of the action. Task consists of a series of actions that are to be executed in sequence, a Boolean to determine whether or not the task is preempt-able, a second Boolean to determine whether or not the task is droppable and a set of intervals, a set of dependencies and lastly a set of tasks to lock.

So, in short a task is determined by: a series of actions, a Boolean to determine preempt-ability, a Boolean to determine drop-ability, a set of intervals, a set of dependencies and lastly locks. Battery is the resource from which components draw power from.

So the system then has a dynamic state consisting of:

$$S \in State = Time \times TaskStatus \times SoC$$

Where TaskStatus is the following:

$$\begin{aligned}&TaskStatus = Task \rightarrow Bool \times Counter \times Counter \times P(Task \times Counter)\\&\times Bool \times Bool \times Bool\end{aligned} \tag{3.1}$$

Where the first Bool is a Boolean indicating whether or not a task is running, the Counter is a counter that decrements as time passes while the task is running. The second Counter is a completion counter that increments every time a task has completed. The P(Task x Counter) is the dependencies of the task, where every time the task completes it saves the current completion counter for the tasks that it is dependent to be compared with later. The last three Boolean's indicates if the task status is locked, just preempted and just dropped respectively.

An overview of a taskstatus can be shown as:

$$\begin{aligned}&TS(T) = (Running, TimeCounter, CompletionCounter, DependencyCounters,\\&\quad Locked, JustPreempted, JustDropped)\end{aligned} \tag{3.2}$$

In the semantics we will present every part of the system as tuples so for instance a Task will be the tuple

$$Task \in Name \times Actions \times Bool \times Bool \times Intervals \times Dependencies \times Locks$$

We will use a construct for sorting out Tasks based on the value of an element in the tuple:

$$\text{Where } T \text{ Where } T = (\_, \_, true, \_, \_, \_, \_) \tag{3.3}$$

Equation 3.3 shows that the task T is a task where the third element in the tuple is true. This construct is used when generating sets of tasks based on the values of specified variables in the tuple.

$$\begin{aligned}&SomeFunction : Task \rightarrow P(Task)\\&SomeFunction(T) = \bigcup_{T \in Task} T \text{ where } T = (\_, \_, true, \_, \_, \_, \_)\end{aligned} \tag{3.4}$$

Where Equation 3.4 returns a set of tasks where the tasks are preempt-able.

Furthermore, we will also use a similar construct to retrieve a variable from one of these tuples:

$$SomeFunction(T) = SomeFunctionNext(Depends) \text{ where } T = (\_, \_, \_, \_, depends, \_, \_) \tag{3.5}$$

Where this function calls a subsequent function that takes the specified variable in as a parameter.

So for each state there are four possible things that can happen:

- Start a task that is available for start. We will call this rule for *StartTask*
- Drop a task that is running. Named *DropTask*
- Preempt a task that is running. Names *PreemptTask*
- Let the time progress by one increment. Named *Time*

Where we will for each of the following sections present the semantics for each of the rules.

3.1 Starting a Task

For a task to start we check for:

- Is the task within opportunity and can it complete within its opportunity
- Is the dependencies resolved
- Is the required components available
- Is the task locked and can it lock the tasks it needs to
- Will the battery be emptied if it is started

Furthermore, there might be more than one task available, and therefore the system can start any which of the available tasks. There must also be enough time within the opportunity for the task to complete. Therefore we define a function *AvailTasks* that calculates which tasks are available. Five functions determines this, called CheckOpp, CheckDep, CheckComponents, CheckLocks, and CheckPower. And lastly there must be enough power on the battery for the task to be able to complete given the situation.

3.1.1 CheckOpp

In order to check for if the opportunity is upheld we need to make sure that:

- The global time is within an opportunity interval defined for a given task
- That the task will not end passed the end of the interval

In order to figure this out we need to know the duration of the task, which is shown in Equation 3.6

$$\begin{aligned}
&DurationTask : Task \rightarrow \mathbb{N} \\
&DurationTask(T) = DurationActions(acts) \text{ for } T = (\_, acts, \_, \_, \_, \_) \in Task \\
&\quad \text{where } acts \in Action \\
&DurationActions(acts) = \sum_{i=0}^{i=n} DurationAction(act) \text{ For } acts = (act_0, ..., act_n) \\
&DurationAction(act) = d \text{ for } act = (\_, \_, d)
\end{aligned} \tag{3.6}$$

Now that we know the duration of the task, as shown in Equation 3.6 we can determine whether or not the task can start in its opportunity.

The function $CheckOpp$ is defined:

$$\begin{aligned}
&CheckOpp : Task \times Time \rightarrow Bool \\
&CheckOpp(Task, Gt) = CheckOppInt(intervals, Gt) \text{ for } T = (\_, \_, \_, \_, intervals, \_) \\
&CheckOppInt(intervals, Gt) = \\
&\begin{cases} true, \text{if } \exists(t_1, t_2) \in \text{intervals, where } t_1 \leq Gt \leq t_2 \land duration(T) + Gt \leq t_2 \\ false, otherwise \end{cases}
\end{aligned} \tag{3.7}$$

This function as shown in Equation 3.7 return true if the it can start within its opportunity, false if not.

3.1.2 CheckDep

As stated some tasks can have dependencies declared on other tasks meaning that they need to be resolved in order for the task to complete. The dependencies are monitored by saving the completion counter for each of the dependent tasks the last time the task completed. So, when we check whether or not the task is ready we subtract the dependency counter with the corresponding completion counter and compare it with the dependency declared in the system and determine whether or not it is resolved.

To begin with we define a function that returns the specified dependencies for a specific task as shown in Equation 3.8.

$$\begin{aligned}
&GetDependencies : Task \rightarrow P(Task \times \mathbb{N}) \\
&GetDependencies(T) = Deps \text{ for } T = (\_, \_, \_, \_, \_, deps, \_)
\end{aligned} \tag{3.8}$$

Thereafter we define a function that returns the counter for the dependency which will be used to determine whether or not the dependencies has been resolved as shown in Equation 3.9.

$$\begin{aligned}&GetDepCount : Task \times Task \to \mathbb{N}\\&GetDepCount(T, T') = \begin{cases} counter', \text{ if } \exists (T', counter') \in GetDependencies(T) \\ 0 \text{ otherwise}\end{cases}\end{aligned} \tag{3.9}$$

With the ability to extract the number of times a task must complete as specified in the system we need to be able to get the current completions for a task as shown in Equation 3.10

$$\begin{aligned}&GetCompletions : Task \times TaskStatus \to \mathbb{N}\\&GetCompletions(T, TS) = CCounter \text{ for } TS(T) = (\_, \_, CCounter, \_, \_, \_, \_)\end{aligned} \tag{3.10}$$

Thereafter we need to retrieve the dependencies as they were seen the last time the given task was run as shown in Equation 3.11.

$$\begin{aligned}&GetLastCounts : Task \times TaskStatus \to P(Task \times \mathbb{N})\\&GetLastCounts(T, TS) = lastcount \text{ for } TS(T) = (\_, \_, \_, lastcount, \_, \_, \_)\end{aligned} \tag{3.11}$$

And now, with all of these defined we can check whether or not the dependencies has been resolved as shown in equation 3.12.

$$\begin{aligned}&CheckDep : Task \times TaskStatus \to Bool\\&CheckDep(T, TS) =\\&\begin{cases} true, \text{ if } \forall (T', counter') \in GetLastCounts(T, TS) : \\ \quad GetCompletions(T', TS) - counter' \geq GetDepCount(T, T') \\ false, \text{ otherwise}\end{cases}\end{aligned} \tag{3.12}$$

If the dependency is resolved the function returns true. The functions checks all the dependency counters for the taskstatus of a task and subtracts the counter from the amount of current completions and compares with the count for the task specification.

3.1.3 CheckComponents

Another check we need to do is that the components in the task must not be in use beforehand, so we need to check whether or not the task's components are in use. We call this function CheckComponents. In order to check if the components are in use, we need to check which tasks that are active and from there calculate which of the actions that are running. When we know which actions are running we can determine which components that are in use and from there check whether or not the needed components are available.

We need a helper function that returns the set of running tasks as shown in Equation 3.13.

$$\begin{aligned} &GetRTask : TaskStatus \times Task \rightarrow P(Task) \\ &GetRTask(TS) = \bigcup_{T \in dom(TS)} T \text{ where } TS(T) = (true, \_, \_, \_, \_, \_, \_) \end{aligned} \tag{3.13}$$

Thereafter we get the action that is currently the active one within the task as shown in Equation 3.14.

$$\begin{aligned} &GetActionR : Task \times Counter \rightarrow Action \\ &GetActionR(acts, d) = \text{Where T} = (\_, acts, \_, \_, \_, \_) \text{ where } acts = acts_0 : ... : acts_n \\ &\begin{cases} acts_0, \text{ if d} \geq Duration(Task) - Ad \text{ for } acts_0 = (\_, \_, Ad) \\ GetActionR(acts_1...acts_n, d) \text{if d} < Duration(Task) - Ad \end{cases} \end{aligned} \tag{3.14}$$

Once we have the action we can retrieve the components that the action is occupying as shown in Equation 3.15

$$\begin{aligned} &GetComponents : Action \rightarrow P(Component) \\ &GetComponents(A) = C for A = (\_, C, \_) \end{aligned} \tag{3.15}$$

Now that we can retrieve a single active action and its components, we can with the following function retrieve all the active actions in the system as shown in Equation 3.16.

$$\begin{aligned} &GAA : TaskStatus \rightarrow P(Action) \\ &GAA(TS) = \bigcup\nolimits_{\text{T} \in \text{GetRTasks(TS)}} \text{GetActionR(T,Counter)} \ \text{ for TS(T)} = (\_, Counter, \_, \_, \_, \_, \_) \end{aligned} \tag{3.16}$$

Once we are able to retrieve all the active actions we can retrieve all the active components with the function defined in Equation 3.17.

$$
\begin{aligned}
&GetActiveComponents : TaskStatus \to P(Component)\\
&GetActiveComponents(TS) = \bigcup_{A \in GAA(TS)} GetComponents(A)
\end{aligned}
\tag{3.17}
$$

Now that we have retrieved all the active components, we can compare them to the needed components to determine if they are available.

We make a helper function that retrieves the set of needed components for the task to start, which is the components needed by the first action in the sequence shown in equation 3.18.

$$
\begin{aligned}
&RetreiveComponents : Task \to Component\\
&RetreiveComponents(T) = RetreiveComponentsActs(acts) \text{ for T} = (\_, acts, \_, \_, \_, \_)\\
&RetreiveComponentsActs(acts) = GetComponents(acts_0) \text{ for } acts = acts_0, ..., acts_n
\end{aligned}
\tag{3.18}
$$

This function can now be used to define the function where we determine whether or not the components are available as shown in equation 3.19.

$$
\begin{aligned}
&CheckComponents : Task \times TaskStatus \to Bool\\
&CheckComponents(T, TS) =\\
&\begin{cases} true \text{ if } RetreiveComponents(T) \bigcap GetActiveComponents(TS) = \{\} \\ false\text{, otherwise} \end{cases}
\end{aligned}
\tag{3.19}
$$

If none of the components needed are in the set of active components the function true, and false if it does exist in the set of active components.

3.1.4 Check Locks

The task must not be locked, so we define a function in Equation 3.20 that returns true if the task is unlocked and false if it is locked.

$$
\begin{aligned}
&IsUnlocked : Task \times TaskStatus \to Bool\\
&IsUnlocked(T, TS) =\\
&\begin{cases} true\text{, if } locked = false \text{ where } TS(T) = (\_, \_, \_, \_, \_, \_, locked) \\ false \text{ otherwise} \end{cases}
\end{aligned}
\tag{3.20}
$$

Thereafter we have to check that the tasks to be locked are not running, as shown in equation 3.21.

$$
\begin{aligned}
&CanLock : Task \times TaskStatus \rightarrow Bool\\
&CanLock(T, TS) =\\
&\begin{cases} true, \text{ if } GetRTask(TS) \cap Locks = \{\} \text{ where } T = (\_, \_, \_, \_, \_, \_, Locks)\\ false \text{ otherwise}\end{cases}
\end{aligned}
\tag{3.21}
$$

With these two functions 3.20 and 3.21 we will make a function that makes both of these checks as shown in equation 3.22

$$
\begin{aligned}
&CheckLocks : Task \times TaskStatus \rightarrow Bool\\
&CheckLocks(T, TS) = \begin{cases} true, \text{ if } IsUnlocked(T, TS) \land CanLock(T, TS)\\ false, \text{ otherwise}\end{cases}
\end{aligned}
\tag{3.22}
$$

Now we have defined the function that determines if we are locked and able to lock the required tasks.

3.1.5 CheckPower

Lastly we define the function that checks whether or not the task can complete given the current situation, as shown in equation 3.48. The function is recursive where it calls itself again with changed parameters until either the task duration has passed or the state of charge has gone below zero.

$$
\begin{aligned}
&CheckPower : TaskStatus \times Battery \times \mathbb{N} \times \mathbb{N} \rightarrow Bool\\
&CheckPower(TS, B, SoC, TaskCounter) =\\
&\begin{cases} true, \text{ if } TaskCounter == 0 \text{ and } SoC > 0\\ false, \text{ if } SoC < 0\\ CheckPower(Time(TS), B, ApplyLoad(B, SoC, TS), TaskCounter - 1), \text{ otherwise}\end{cases}
\end{aligned}
\tag{3.23}
$$

The functions *Time*, 3.55, and *ApplyLoad*,3.56, are defined in Section 3.4 regarding time transitions. The goal of this function is to determine if there is enough power on the battery for the task to complete given the current situation.

3.1.6 Available Tasks

With all the functions created that determines whether or not a task is available for start we can define the set of tasks that can be started.

Firstly we make a function that determines whether or not a specific task can start as shown in Equation 3.24.

$$
\begin{aligned}
&IsTaskAvailable : Task \times Time \times TaskStatus \times Battery \rightarrow Bool\\
&IsTaskAvailable(T, Gt, TS, B) =\\
&\begin{cases} true, \text{ if } CheckDep(T) \wedge CheckComponents(TS) \wedge CheckOpp(T, Gt)\\ \wedge CheckLocks(T, TS) \wedge CheckPower(StartTask(TS, T), B, SoC, Duration(T))\\ false, \text{ otherwise} \end{cases}
\end{aligned} \tag{3.24}
$$

The function *StartTask* that is applied when using the function CheckPower is defined in Equation 3.33. We use this function to gather the set of available tasks as shown in Equation 3.25.

$$
\begin{aligned}
&AvailableTasks : TaskStatus \times Task \times Time \times Battery \rightarrow P(Task)\\
&AvailableTasks(TS, T, Gt, B) = \bigcup_{T \in Tasks} T \text{ where } IsTaskAvailable(T, Gt, TS, B) = true
\end{aligned} \tag{3.25}
$$

Now we have the set of all the available tasks. However, we cannot get the preempted tasks with this function meaning that we need to define a function that provides the tasks that are preempted but available for resuming. We only need to check whether or not the components are available as it would not be in a preempted state if either opportunities or dependencies were not resolved.

First we need a function to retrieve set of preempted tasks and in order to do so we need a function that determines if a task is preempted, as shown in Equation 3.26.

$$
\begin{aligned}
&IsPreempted : Taskstatus \times Task \rightarrow Bool\\
&IsPreempted(T, TS) =\\
&\begin{cases} true, \text{ if } Running = false \text{ and } Counter > 0 \text{ for } TS(T) = (Running, Counter, \_, \_, \_, \_, \_)\\ false, \text{ otherwise} \end{cases}
\end{aligned} \tag{3.26}
$$

Hereafter we can define the function that retrieves all preempted tasks as defined in Equation 3.27.

$$
\begin{aligned}
&GetPTasks : TaskStatus \rightarrow P(Task)\\
&GetPTasks(TS) =\\
&\bigcup_{T \in dom(TS)} T \text{ where } IsPreempted(T, TS) = true
\end{aligned} \tag{3.27}
$$

Now that we have retrieved the set of preempted tasks we need to define the set of them that can be resumed as shown in Equation 3.28

$$\begin{aligned}
&AvailablePTasks : TaskStatus \rightarrow P(Task)\\
&AvailablePTasks(TS) =\\
&\bigcup_{T \in GetPTasks(TS)} T \text{ where } CheckComponents(T) \wedge CheckLocks(T, TS)
\end{aligned} \tag{3.28}$$

Now that we have defined how we retrieve the set of the tasks that can be started we can continue to what happens when a task is stared.

3.1.7 Start a Task

When a non-preempted task is started the Boolean flag indicating if it is running is set to true and the counter is set to the duration of the task. When a preempted task is started only the Boolean indicating if the task is running is set to true. This is done by the function defined in equation 3.29.

$$\begin{aligned}
&ActTask : TaskStatus \times Task \rightarrow TaskStatus\\
&ActTask(TS, T) =\\
&\begin{cases}
TS[T \mapsto (true, Cn, CCounter, DCounter, false, JPbool, JDBool)]\\
\text{ for } TS(T) = (\_, Cn, CCounter, DCounter, \_, JPBool, JDBool) \text{ if } IsPreempted(T, TS) = true\\
TS[T \mapsto (true, Duration(T), CCounter, DCounter, false, JPBool, JDBool)]\\
\text{ for } TS(T) = (\_, \_, CCounter, DCounter, \_, JPBool, JDBool), \text{ otherwise}
\end{cases}
\end{aligned} \tag{3.29}$$

Furthermore, when some tasks are started they need to lock all the tasks, so we need a function that locks a task, which is shown in equation 3.30.

$$\begin{aligned}
&LockTask : Task \times TaskStatus \rightarrow TaskStatus\\
&LockTask(T, TS) =\\
&TS[T \mapsto (false, Duration, CCounter, DCounter, true, JPBool, JDBool)]\\
&\text{where } TS(T) = (\_, Duration, CCounter, DCounter, \_, \_JPBool, JDBool)
\end{aligned} \tag{3.30}$$

Hereafter we define a function that locks all the tasks that must be locked for a given task to start as shown in equation 3.31.

$$\begin{aligned}
&LockTasks : Task \times TaskStatus \to TaskStatus\\
&LockTasks(T, TS)(T') =\\
&\begin{cases} LockTask(T', TS)(T') \text{ if } T = (\_, \_, \_, \_, \_, \_, locks) \land T' \in locks\\ TS(T')\text{otherwise}\end{cases}
\end{aligned} \tag{3.31}$$

So, when a task is started it starts itself and locks the tasks that needs to be locked with the function defined in equation 3.32.

$$\begin{aligned}
&StartTask : TaskStatus \times Task \to TaskStatus\\
&StartTask(TS, T) = ActTask(LockTasks(T, TS), T)
\end{aligned} \tag{3.32}$$

With this task defined we can now define the rule for starting a task as shown in equation 3.33.

$$StartTask\frac{t \in AvailableTasks(TS, T, Gt, B) \cup AvailablePTasks(TS)}{(Gt, TS, SoC) \to (Gt, StartTask(TS, t), SoC)} \tag{3.33}$$

3.2 DropTask

A task can be dropped if it is possible to do so by the definition of the task in the system. Meaning that in order to drop a task it must both be running and defined as droppable.

So in order to drop a task we need to get the tasks that are running, droppable and not just dropped, as shown in equation 3.34

$$\begin{aligned}
&GetRunningDTasks : Task \times TaskStatus \to P(Task)\\
&GetRunningDTasks(T, TS) = \bigcup_{T \in GetRTask(TS)} T \text{ where } T = (\_, \_, true, \_, \_, \_)\\
&\text{ and TS(T) = (\_,\_,\_,\_,\_,\_,false)}
\end{aligned} \tag{3.34}$$

The reason for having the Boolean just dropped is in order to avoid cyclic state transitions. If it was not there the model would be able to, in some instances, to drop the task, start it, and drop it again etc.

When a task is dropped it should also release all the locks that it has on other tasks. Therefore we define a function that unlocks a single task, as shown in equation 3.35

$$\begin{aligned}
&UnlockTask : Task \times TaskStatus \rightarrow TaskStatus\\
&UnlockTask(T, TS) =\\
&TS[T \mapsto (false, Duration, CCounter, DCounter, false, JPBool, JDBool)]\\
&\text{where } TS(T) = (\_, Duration, CCounter, Dcounter, \_, JPBool, JDBool)
\end{aligned} \tag{3.35}$$

Then we define the function that unlocks all the tasks that were locked by a given task as shown in equation 3.36

$$\begin{aligned}
&UnlockTasks : Task \times TaskStatus \rightarrow TaskStatus\\
&UnlockTasks(T, TS)(T') =\\
&\begin{cases} UnlockTask(T', TS)(T') \text{ if } T = (\_, \_, \_, \_, \_, \_, locks) \wedge T' \in locks\\ TS(T') \text{ otherwise} \end{cases}
\end{aligned} \tag{3.36}$$

And here is the function that drops a specified task as shown in 3.37:

$$\begin{aligned}
&DropTask : TaskStatus \times Task \rightarrow TaskStatus\\
&DropTask(TS, T) =\\
&TS[T \mapsto [false, 0, CCounter, DCounter, locked, JPBool, true]\\
&\text{ for } TS(T) = (\_, \_, CCounter, DCounter, locked, JPBool, \_)
\end{aligned} \tag{3.37}$$

So the function that drops and unlocks the tasks is shown in equation 3.38;

$$\begin{aligned}
&Drop : TaskStatus \times Task \rightarrow TaskStatus\\
&Drop(TS, T) = DropTask(UnlockTasks(T, TS), T)
\end{aligned} \tag{3.38}$$

With this we can define the rule for dropping a task as shown in 3.39.

$$Drop\frac{t \in GetRunningDTasks(T, TS)}{(Gt, TS, SoC) \rightarrow (Gt, Drop(TS, t), SoC)} \tag{3.39}$$

3.3 Preempt Task

Preemption is only allowed if the task itself is preempt-able and running but also if there is at least enough time for the task to complete within its opportunity for the current time plus one. So, what we do first is get the Running tasks that are preempt-able as shown in Equation 3.40.

$$\begin{aligned}
&GetRunningPTasks : TaskStatus \rightarrow P(Task)\\
&GetRunningPTasks(T, TS) = \bigcup_{T \in GetRTask(TS)} T \text{ where } T = (\_, \_, true, \_, \_, \_)\\
&\text{ and } TS(T) = (\_, \_, \_, \_, \_, false, \_)
\end{aligned} \tag{3.40}$$

We need a function that determines if a task can complete if it is preempted for one time unit. We do this by first making a function that determines that, based on a time, if there is enough for the task to finish as shown in Equation 3.41. It is very similar to the CheckOpp function except that instead of calculating the duration of the task we make use of the counter that contains the remaining time until the task finishes.

$$\begin{aligned}
&Preemptable : Task \times Counter \times Time \rightarrow Bool\\
&Preemptable(T, Counter, Gt) = PreemptableInt(intervals, Counter, Gt)\\
&\quad \text{for } T = (\_, \_, \_, \_, intervals, \_, \_)\\
&PreemptableInt(intervals, Counter, t) = \begin{cases} true, \text{ if } \exists (t_1, t_2) \in intervals \text{ where } t_1 \leq Gt \leq t_2 \text{ and } Gt + Count \\ false, \text{ otherwise} \end{cases}
\end{aligned} \tag{3.41}$$

With this function we can now determine if it can complete if time passes and returns the preempt-able tasks. We make use of the function in 3.41 where we add a one to the third parameter i.e. the global time as shown in Equation 3.42.

$$\begin{aligned}
&PreemptableTasks : TaskStatus \times Time \rightarrow P(Task)\\
&PreemptableTasks(TS, Gt) = PreemptableTasksC(TS, Counter, Gt)\\
&\quad \text{for } TS(T) = (\_, Counter, \_, \_, \_, \_, \_)\\
&PreemptableTasksC(TS, Counter, Gt) =\\
&\qquad \bigcup_{T \in GetRunningPTasks(TS)} T \text{ where } Preemptable(TS, C, Gt + 1) = true
\end{aligned} \tag{3.42}$$

So, what we do when preempting a task is simply setting the running Boolean to false and setting the Boolean indicating if the task has just been preempted to true, while keeping values of the other elements as shown in Equation 3.43.

$$\begin{aligned}
&PreemptTask : TaskStatus \times Task \rightarrow TaskStatus\\
&PreemptTask(TS, T) = TS[T \mapsto (false, duration, CCounter, DCounter, locked, true, JDBool)\\
&\text{ for } TS(T) = (\_, duration, CCounter, DCounter, locked, \_, JDBool)
\end{aligned} \tag{3.43}$$

It must also unlock all the tasks that it has locked as well, so the function for preempting a task is given Equation 3.44

$$\begin{aligned} &Preempt : TaskStatus \times Task \rightarrow TaskStatus \\ &Preempt(TS, T) = PreemptTask(UnlockTasks(T, TS), T) \end{aligned} \tag{3.44}$$

And hereby we can define the rule in Equation 3.45

$$PreemptTask \frac{t \in PreeptableTasks(GetRunningPTasks(T, TS))}{(Gt, TS, SoC) \rightarrow (Gt, Preempt(TS, t), SoC)} \tag{3.45}$$

3.4 Time

There are a series of conditions that must be upheld in order for time to pass in the system and these are:

- SoC must not go below zero
- A preempted task must be able to be finish even if time passes
- Global time must not exceed end time
- If a task transitions from one action to another, the components must be available for it to be used.
- If a task finishes its completion counter must increment and unlock the tasks it has locked.
- The battery must have the load applied to it

This means that we need functions that check for each of these conditions.

3.4.1 Check Future SoC

We make a function that calculates the load on the battery as shown in Equation 3.46.

$$\begin{aligned} &Load : TaskStatus \rightarrow \mathbb{N} \\ &Load(TS) = \sum_{C \in GetActiveComponents(TS)} cost \text{ for } C = (\_, cost) \end{aligned} \tag{3.46}$$

With this we can define the function that checks if the soc is depleted below zero as shown in 3.47.

$$
\begin{aligned}
&CheckSoC : TaskStatus \times \mathbb{N} \rightarrow Bool\\
&CheckSoC(TS, SoC) =\\
&\begin{cases} true, \text{ if SoC} - Load(TS) \geq 0\\ false, \text{ otherwise}\end{cases}
\end{aligned}
\tag{3.47}
$$

3.4.2 Check Preempted Tasks

We need to make sure that the preempted tasks can still complete once we transition in time. This is done with the function CheckPreempt as shown in Equation 3.48.

$$
\begin{aligned}
&CheckPreempt : TaskStatus \times Time \rightarrow Bool\\
&CheckPreempt(TS, Gt) =\\
&\begin{cases} true, \text{ if } \forall T \in GetPTasks(TS)\ Preemptable(T, Counter, Gt + 1) = \text{true}\\ \text{for } TS(T) = (\_, Counter, \_, \_, \_, \_, \_)\\ false, \text{ otherwise}\end{cases}
\end{aligned}
\tag{3.48}
$$

3.4.3 Check Future Component Availability

In order to check for if there are any component usage overlapping we have to check which action each task will be performing one time step ahead. Meaning that for all active tasks we find the actions that will be active in the next time frame and hereby also their respective components and check if there are any overlaps. This is shown in Equation 3.49.

$$
\begin{aligned}
&CheckFComponents : TaskStatus \times Task \rightarrow Bool\\
&CheckFComponents(TS, T) =\\
&\begin{cases} true, if \forall T \in GetRTask(TS) \bigcap_{A \in GAA(Time(TS))} GetComponents(A) = \{\}\\ false, \text{ otherwise}\end{cases}
\end{aligned}
\tag{3.49}
$$

Now we have made the functions that determines whether or not time is eligible to pass.

3.4.4 When Time Passes

When the time can pass:

- Running tasks counters are decremented by one
- The battery is affected
- Global Time increases by one

A function determines if time can pass is shown in Equation 3.50

$$\begin{aligned}
&CanPass : TaskStatus \times \mathbb{N} \times Time \rightarrow Bool \\
&CanPass(TS, SoC, Gt) = \\
&\begin{cases} true \text{ if } CheckSoc(TS, SoC) \wedge CheckPreempt(TS, Gt) \\ \wedge CheckFComponents(TS, T) \wedge Gt < End \\ false, \text{ otherwise} \end{cases}
\end{aligned} \tag{3.50}$$

We need a function that decrements all the counters for the running tasks. When a task is finishing we must do some additional work. If a TaskStatus has 1 time left it must set the dependencies accordingly and increment its completion counter.

The function that generates the updated dependencies is shown in Equation 3.51

$$\begin{aligned}
&UpdateCompletions : TaskStatus \times Task \rightarrow P(Task \times \mathbb{N}) \\
&UpdateCompletions(TS, T) = UpdateCompletionD(DCounter) \\
&\quad \text{for } TS(T) = (\_, \_, \_, DCounter) \\
&UpdateCompletionsD(DCounter) = \bigcup_{T \in DCounter} (T, GetCompletions(T, TS))
\end{aligned} \tag{3.51}$$

We do so by first getting the dependency counter for the finishing task and generate a set of updated completions for the task. This function is used when transitioning in time and the counter for a task is 1, meaning that it finishes in the next time transition, is shown in 3.52.

$$\begin{aligned}
&FinishingTS : Task \times TaskStatus \rightarrow TaskStatus \\
&FinishingTS(T, TS) = UnlockTasks(T, TS[T \mapsto \\
&(false, Counter - 1, CCounter + 1, UpdateCompletions(TS, T), false, false, false)]) \\
&\text{where } TS(T) = (\_, Counter, CCounter, \_, \_, \_, \_)
\end{aligned} \tag{3.52}$$

So the task is finishing if the counter that determines the amount of time left is equal to one as shown in Equation 3.53

$$Finshing : TaskStatus \rightarrow TaskStatus$$
$$Fishing(TS)(T') =$$
$$\begin{cases} FinishingTS(T', TS) \text{ if } running \wedge Counter = 1 \text{ for } TS(T') = (running, Counter, \_, \_, \_, \_, \_) \\ TS(T') \text{ otherwise} \end{cases} \tag{3.53}$$

Now that we have determined what is done when a task finishes we can define the function that determines what happens for a task that is running but not finishing which is shown in 3.54

$$TimeTS : Task \times TaskStatus \rightarrow TaskStatus$$
$$Time(T, TS) = TS[T \mapsto (bool, Counter - 1, CCounter, DCounter, locked, false, false)]$$
$$where\ TS(T) = (bool, counter, CCounter, DCounter, locked, \_, \_) \tag{3.54}$$

Then we can define the function that time transitions for all running tasks as shown in 3.55.

$$Time : TaskStatus \rightarrow TaskStatus$$
$$Time(TS)(T') =$$
$$\begin{cases} TimeTS(T', TS) \text{ if running} = true \text{ for } TS(T') = (running, \_, \_, \_, \_, \_, \_) \\ TS(T') \text{ otherwise} \end{cases} \tag{3.55}$$

Furthermore the just-preempted and just dropped Boolean's are set to false. Now that we have defined the behaviour of what happens for running tasks when time transitions we can define what happens to the battery as shown in Equation 3.56. We get all the active components and then sum their costs in order to calculate the load that will be applied to the battery.

$$\begin{aligned} &ApplyLoad : \mathbb{N} \times \mathbb{N} \times TaskStatus \rightarrow \mathbb{N} \\ &ApplyLoad(B, SoC, TS) = \\ &\begin{cases} Cap, \text{ if } SoC - Load(TS) > Cap \text{ for } Battery = (Cap, \_) \\ SoC - Load(TS), \text{ otherwise} \end{cases} \end{aligned} \tag{3.56}$$

Here we have defined that if the battery is charged above its maximum capacity, max capacity is returned. In the case the capacity isn't reached we return the value for the state of charge after the load is applied. The final rule can then be defined for a time transition as shown in Equation 3.57.

$$Time\frac{CanPass(TS, SoC, Gt) = true}{(Gt, TS, SoC) \rightarrow (Gt + 1, Time(UpdateCompletions(TS)), ApplyLoad(B, SoC, TS)} \tag{3.57}$$

So, when time passes we increment the global clock with one, we update finishing tasks and decrement the counter for non-finishing but running tasks and apply the load to the battery.

Chapter 4

Query Language

When defining a system, there can be certain requirements for the resulting schedule. For instance a system might not need to perform the highest possible amount of tasks. Furthermore, the battery resource could also be in some environment where either a high average state of charge or low average state of charge is preferred. This could be a system that is has part of its work scheduled by the BATTCIO formalism and another part that takes over if needed, for instance if some sort of emergency emerges that requires the system to act differently. The other aspect can be if the act of recharging the battery is an expensive action to take you might want to recharge as little as possible while still performing the specified amount of tasks. A schedule is in essence a series of states, and it would be these series of states to which we will put constraints when querying.

There could be a system modelled where we the designers are not interested in the highest throughput, but rather use the formalism for quick planning. A service like system could be considered where the tasks that the system performs are services provided to customers. So, even though the system is able to perform a higher amount of tasks it might not be needed. Therefore a query language will be defined in order to generate more customizable schedules.

4.1 Example Model and Queries

In this section we will make use of an example system as defined in Listing 4.1

Listing 4.1 : a small example system

```
Component Antenna(5);
Component Processor(5);
Component SolarPanel(-20);

Action Receive(Components: {Antenna, Processor} Duration: 2);
Action Send(Components: {Antenna, Processor} Duration: 4);
Action Recharge(Components{SolarPanel} Duration: 6);

Task Send(Actions: [Send]);
Task Receive(Actions: [Receive]);
Task Recharge(Actions: [Recharge]
          Droppable: True);

Interval Send1(6,10);
Interval Send2(16,20);
Interval Send3(26,30);
Interval Send4(36,40);
Interval Receive1(2,4);
Interval Receive2(12,14);
Interval Receive3(22,24);
Interval Receive4(32,34);
Interval Charge1(0,8);
Interval Charge2(10,12);
Interval Charge3(16,26);
Interval Charge4(30,40);

Opportunity(Intervals: Send1,Send2, Send3, Send4
            Task: Send
            Dependencies: Receive: 1);

Opportunity(Intervals: Receive1, Receive2, Receive3, Receive4
            Task: Receive);

Opportunity(Intervals: Charge1,Charge2,Charge3
            Task:ReCharge);

Battery(Capacity: 150
        InitialCharge: 150
        Type: Discrete);

Start(0);
Termination(40);
```

A simple approach to this is to simply say how many times a certain task must be completed in the schedule. This kind of specification would put requirements on the last state in the generated state trace. The queries means that when the entire mission time has passed the tasks must uphold the specified query. Meaning, that the query results in schedules that span the entire mission time and the constraints of the query are upheld.

For instance in the case of our test program a query could be formed in the following form:

$$Receive >= 5$$

This would result in a schedule where all tasks except *receive* will complete as many times as possible while task Receive has to complete five times or more. This could result in a set of candidate schedules that the system designer could browse through in order to find the most wanted one.

Another opposite approach is to say that it must be completed at most 5 times:

$$Receive <= 5$$

Like before, all other tasks would complete as many times as possible while *receive* would be limited to a maximum completion of five.

Another consideration to take is we have a system where we would like one task to complete twice as much as another during the entire schedule. In other words, a balance between tasks might be preferable.

In a large system with a large amount of defined tasks, writing a query that determines the amount of completions for each task can be a time consuming process. Therefore we propose a sort of "balancing" query in which we can define how many times a task completes compared to another task. This is not to be mistaken for some sort of dependency setting, as that is already possible in the formalism itself. A dependency clearly defines which tasks and how many times it must complete before the task can start whereas this balancing in only looks at number of completions in the end of the schedule.

So, this can be defined in such a way:

$$Task1 : 2/Task2 : 1$$

Here we state that for every two times task1 is executed, task2 must also be executed once. However, this does not mean: $Task1 \rightarrow Task1 \rightarrow Task2$, it means that for every two times Task1 has been completed Task2 must also have been completed once at any given time when the schedule has reached its end. This is in a system where strict dependency settings might not yield preferable results.

Another aspect that we can look into in regarding the querying is the battery resource and what we want to happen there. A series of scenarios can be considered. One scenario is the case where we want to always have a certain amount of battery available. In the case of the GomX-3 satellite, there must always be at least 20 percent battery power available [1]. So in a query we can also define this behavior for the battery:

$$Battery >= 20\%$$

When setting limiting parameters on the battery in the query it as a constraint that should be upheld for all the states in the trace schedule.

Another aspect to consider when working with batteries is how they are affected. As research showed in the report [2] how fast the battery is discharged and how deep it is discharged affects the longevity of the battery. Therefore we might be concerned with how often it should take a recharge opportunity.

As learned depth of discharge had a high effect on battery longevity and therefore we should be able to express that it should recharge as often as possible:

$$Battery : HighCR$$

Where HighCR stands for High Charge Rate. This approach makes best sense if the user has defined a query with a very exact schedule. If there are not any upper or lower bounds defined on tasks the resulting scheduling might be one where the only thing that is performed is the recharge of the battery and not much else.

Another approach might be to charge as few times as possible. This can for instance be in the case of a system where the task of recharging locks the system from doing other critical tasks and therefore the desire might be to keep the recharging at a minimum.

$$Battery : LowCR$$

Here LowCR stands for low charge rate.

With these considerations the resulting BNF for the querying is:

⟨*query*⟩ ::= ⟨*taskspec*⟩ ';' ⟨*query*⟩
  | ⟨*taskspec*⟩ '/' ⟨*taskspec*⟩
  | ⟨*taskspec*⟩ ';'
  | ⟨*battspec1*⟩ ';'
  | ⟨*battspec1*⟩ ';' ⟨*battspec2*⟩
  | ⟨*battspec2*⟩ ';'

⟨*taskspec*⟩ ::= ⟨*ident*⟩ '=' ⟨*num*⟩
  | ⟨*ident*⟩ '⟨=' <*num*⟩
  | ⟨*ident*⟩ '>=' ⟨*num*⟩

⟨*battspec1*⟩ ::= 'Battery' '>' ⟨*num*⟩ '%'
  | 'Battery' '>=' ⟨*num*⟩ '%' ';'

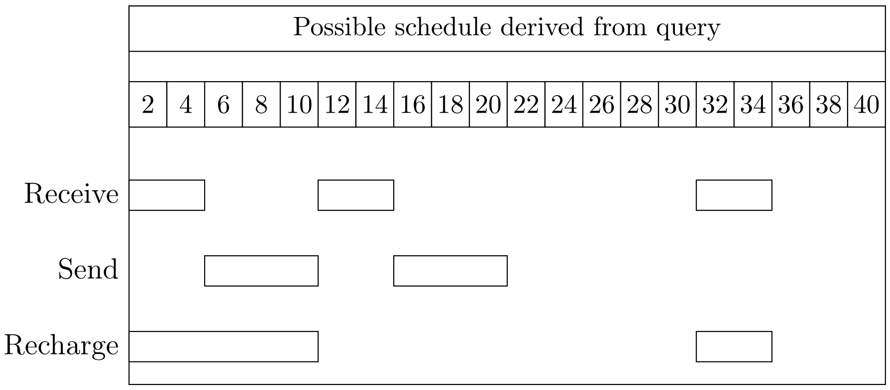
⟨*battspec2*⟩ ::= 'Battery' ':' 'HighCR'
  | 'Battery' ':' 'LowCR'

Here as stated there can be a series of task specifications that determines the amount of times defined tasks are to be executed. Hereafter the lower bound of the battery can be set and, if wanted, the charge rate of the battery be set as well.

We can now define a set of relatively simple queries for the test program as defined in Listing 4.1.

Listing 4.2 : A small query for the test program that quieries for a schedule where Task Send and Task Receive completes at least 4 times

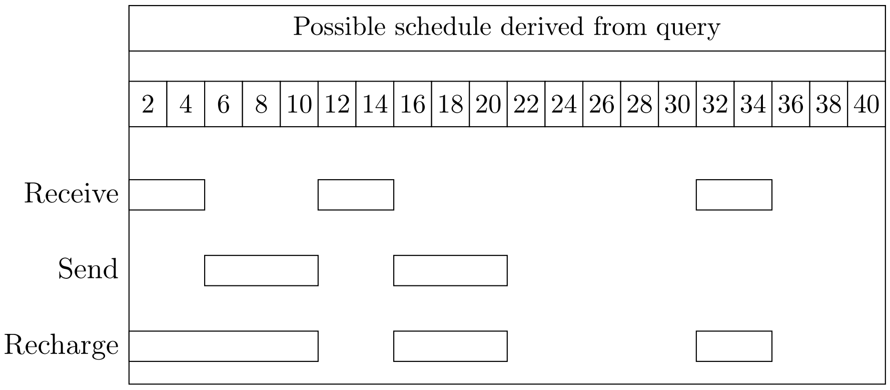
```
Send >= 2;
Receive >= 3;
```



Listing 4.2 shows a small query that says the resulting schedule must have both task Send and Receive completing more than 4 times. Below that is shown a Gantt chart that contains an example of a schedule that such a query could define. As can be seen the amount of times that each tasks run are equal to the amount specified in the query. However, if Task Send had been executed one more time, the resulting schedule would have been valid as well.

Listing 4.3 : A query that states that we want to charge as much as possible

```
Battery : HighCR;
```

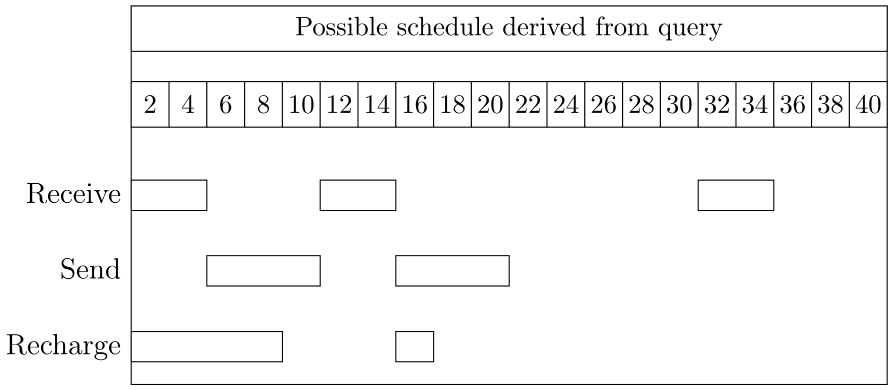


Listing 4.4 queries for the highest charge rate possible for a given schedule. To make the schedules more easily comparable we omit from making too distinct example schedules when regarding battery related queries. With a charge rate as high as possible the system takes every opportunity to charge the battery as it can to keep it at as close to 100 percent as possible.

Note that this given system can perform tasks while charging.

Listing 4.4 : A query that says there must always be at least 20 percent power and have a low charge rate

```
Battery > 20;
Battery : LowCR;
```



The last query in this section shows a possible schedule where the battery should charge as little as possible while always being above 20 percentage of charge. This query could also result in a schedule where the tasks has higher or fewer completions.

Regarding the conceptual GomX-3 satellite that we are using as a main case, the state of charge must not go below a certain threshold as it will stop performing tasks to recover if that happens [1]. This means that specifying that the battery must not go below a certain threshold will in this case be useful. Furthermore, there is a task that must be taken as often as possible, which means that specifying the amount of tasks an important task must run can be very useful. The schedule designer would have to count the amount of times the important task must run, which can be all possible times, and then specify the query containing that the battery must not go below a threshold and the important task must complete said number of times.

4.2 Finding the Schedules

In order to find these schedules there are different approaches that can be taken based on the query which is based on how the implementation of the formalism and schedule generator will be defined. Since the schedules that we generate must span from the defined start time to the defined end time, it is not a viable solution to simply stop exploring the state space once a state has been reached where the tasks has been completed the specified number of times. Furthermore the goal of this is to provide a comparative approach where the system designer can compare several candidate schedules we would need to have a series of schedules that satisfy the query.

Therefore one approach can be to explore the entire state space, and then compare the query to the termination states in order to see if enough completions has been done. However, generating entire state spaces for systems that span over a long period of time might not be feasible as the state space can in result be too big to have in memory.

Another approach can be to make use of a state space reducing algorithm that aims for the specified query. However, as it might not be known initially how many times a task can be completed by the defined system, some measures must be taken in order to accommodate this.

Regarding the battery minimum charge that can be defined in the query, a straight forward approach would be to subtract the specified percentage from the battery before initiating the state-space generation. A task that recharges is a task that is defined in the same way as any other task, and therefore a completion counter is also part of the corresponding task status. Therefore, regarding LowCr and HighCr we would be choose the states where the charge task has the lowest and highest amount of completions respectively.

Chapter 5

Implementation

This chapter covers the considerations behind and implementation of the schedule generator for BATTCIO.

We consider two possibilities for the implementation, one in which a new toolset is constructed specifically for this problem domain, and one in which existing tooling, such as Uppaal is used as a back-end.
As mentioned in Section 2.4 the previous Uppaal models were able to produce a schedule of up to 11 orbits before exhausting the 8GB memory available on the host machine. This was accomplished in $\tilde{2}05$ seconds. While it is likely that the length of the generated schedules could be increased to accommodate the system requirements, it is also likely to take longer time to do so. One of the benefits of using a tool like Uppaal as the system back-end is that it is capable of providing guaranteed optimal schedules.

Contrary to Uppaal, it is unlikely that a the option of creating the domain specific tool from scratch will be able to provide the same guarantees. However, a tool designed specifically for this problem domain would be able to follow the BAATTCIO semantics, as described in Section 3, more closely. If we do not require the guaranteed optimal schedule, it is then possible that we could explore a lesser part of the potential state space, but reach an acceptable solution within a shorter time frame. This behavior would make the tool much more interactive, allowing the operator to quickly experiment with what is possible during the allotted time frame. Furthermore, the person that designs the shedules already passes the candidate schedule throuhg a battery verification software [1]. It could therefore be desirable to generate schedules quickly if the candidate schedules does not pass their verification software. As stated in the [2] we had high memory consumption in the Uppaal implementation.

Therefore this is one thing that we want to address in the specialized schedule generator that we implement. Furthermore we would like to implement this in a compiled language as that is usually faster than an interpreted one. In order to reason on whether or not the specialized tool is taking less space compared to the Uppaal implementation we would need to implement memory monitors as well, as we aim to decrease the amount of memory used when generating a schedule.

We have selected the Rust language for the implementation of this tool, based on the consideration listed below.

- In favor
  - High performance [7]
    - Compiled to LLVM
  - Memory safety
  - Thread safety
  - C bindings
- Opposed
  - New language

As stated above Rust is built on the Low Level Virtual Machine (LLVM), allowing it to take advantage of many optimizations implemented on this platform for popular C compilers, such as Clang. In addition the Rust language has been shown to be competitive with C and C++ in a number of benchmarks [5].
The Rust language provides memory safety via its semantics of ownership, movement, and borrowing of data, which in practice disallows multiple references to the same physical data, unless explicitly circumvented. In addition to memory safety these semantics prevent data races between threads, allowing safe concurrency.
This thread safety stems from the *move*, *copy*, and *borrow* semantics inherent in the Rust language, visualized in Appendix A.3. The *move* semantic ensures that, at any given time, a data structure that uses this semantic can only be referenced once. This facilitates that any data structure allocated on the heap, and thus could conceivably be altered by multiple concurrent processes at the same time, can only have a single reference to it. Alternately, data structures that implements the *copy* semantic, will always be subject to a deep copy action when used.
In addition to thread safety, these two language features should greatly reduce the risk of encountering run-time issues, caused by altering data that are used later on in the system and instead creating a copy of the data that is then altered.

Lastly, the language does provide C bindings, should the need arise to opt out of using Rust for a specific task.
One of the issue with using Rust is however that it is a relatively young language, and may be subject to non-backwards compatible updates in the future.

Based on these criteria, we judge that the Rust language is suitable for the implementation of a schedule generator, due to the performance considerations coupled with strong memory guarantees.

This chapter will give an overview of the modules of BATTCIO as well as their implementation.

5.1 Parser

As there, to our knowledge, are no parser generators available capable of generating a lexer and parser for Rust, we have implemented a rudimentary parser for the BATTCIO formalism using regular expressions. For each element type in the formalism (task, action, etc.) a series of regular expressions are matched against a statement in the source code, extracting the relevant information for each statement. As the language is declarative, i.e. it does not specify how the system will execute, only the contents of it, we do not need to keep track of scoping rules, etc. and as such we can implement this simplified parser without regard for creating an accurate abstract syntax tree. As the the BATTCIO in its current implementation aims to be a proof of concept product we will not spend significant time on manually implementing a proper parser, capable of giving the user meaningful errors messages in the event of syntax errors, however, we recognize that this should be implemented for a complete version of the BATTCIO suite.

5.2 Model

This section covers the implementation of the semantics set forth in Chapter 3, as well as the internal representation of a state. This section will firstly bring an overview as to how we have implemented the representation of the state, the taskstatus and lastly the system. Hereafter we will present how we have implemented the behaviour of the four actions that the model can perform: Start a task, preempt a task, drop a task and transition in time.

5.2.1 State Representation

To represent each state in a simulation we use a combination of two structs, namely the *State* and *TaskStatus* data structures. To ensure that states are not changed in other parts of the system after they have been created, they should implement the *copy* semantics mentioned earlier in this chapter.

State

A State, as seen in Listing 1, consists of three fields for global system time, state of charge, and task statuses respectively. We use the **derive(Copy)** and **derive(Clone)** directives to annotate the data structure. Implementing these directives enforces that the *copy* semantics be used for this data, but adds the requirement that the maximum size of the structure can be determined at compile time. This limitation requires that we set a fixed size for the list of *TaskStatus* structures, in this implementation we have chosen a length of 8. To facilitate that there may be less than the maximum amount of task statuses on a state, we wrap the field in the **Option** type, meaning it can be either of type **None** or **Some(TaskStatus)**, which can then be used for pattern matching to find all related task statuses.

```
type time_t = u32;

#[derive(Debug)]
#[derive(Copy)]
#[derive(Clone)]
pub struct State {
    pub time: time_t,                          // Global system
 ↪  time
    pub soc: u64,                              // State of Charge
    pub task_status: [Option<TaskStatus>; 8], // List of task
 ↪  statuses
}
```

Listing 1: Implementation of a state

Task Status

Since the *TaskStatus* data structure, shown in Listing 2, is a part of the *State*, which uses *copy* semantics, *TaskStatus* must also use these semantics, as a copy occurs recursively. Each *TaskStatus* consists of an id of the task it represents a status for, the global time at which the task was started, an indicator of how long the task has been running, as well as a list of dependencies. Finally a *TaskStatus* contains 4 status fields indication whether or not the task is running, locked, was just preempted, or was just dropped. A status field indicating if a task is preempted has been omitted, as this can be deduced from running status and how long the task has been running (if a task is not running and it has been running for a non zero time, it is preempted).

```
type id_t = u16;
type time_t = u32;

#[derive(Clone)]
#[derive(Copy)]
#[derive(Debug)]
pub struct TaskStatus {
    t_id: id_t,           // ID of the task
    running: bool,        // Is the task running
    locked: bool,         // Is the task locked
    counter: time_t,      // How long has the task been running
    start_time: time_t,   // When did the task start running
    completions: u16,     // How many times has the task been
 ↪  completed
    just_preempted: bool, // Was this task preempted at this
 ↪  time?
    just_dropped: bool,   // Was this task dropped at this time?
    dep: [Option<Dependency>; 4], // Dependencies
}
```

Listing 2: Implementation of a task status

5.2.2 System

In the BATTCIO implementation, the *System* struct, seen in Listing 3, represents the system produced by compiling the formalism described in Section 2.3. In essence, this struct functions as a lookup table for the schedule generator, keeping track of the ids, names, and other properties of the individual parts of the system. Since this structure does not need to be changed during execution as the *State* and *TaskStatus* do, this struct does not need to be copy-able and thus there is no need to assign a maximum size of each of the vectors containing components, actions, etc. Since this struct contains all information about the system, this will be where most of the model behaviour is implemented.

```
pub struct System {
    pub start: Start,
    pub end: Termination,
    pub battery: Battery,
    pub components: Vec<Component>,
    pub actions: Vec<Action>,
    pub tasks: Vec<Task>,
    pub intervals: Vec<Interval>,
    pub opportunities: Vec<Opportunity>
}
```

Listing 3: Implementation of a system

5.3 Behaviour

As most of the model behaviour is built in the *System* struct, the implementations of *Component*, *Action*, etc. used in Listing 3, do not contain any significant logic except for constructors and JSON serializers, and will therefore not be discussed further. We will present in overview how the behaviour is mapped from both the Uppaal implementation and also the semantics.

5.3.1 Available Tasks

In the semantics and the Uppaal model, we have defined when a task can start. As stated in the semantics we check for:

Locked, Preempted or Dropped There are minor differences from the Uppaal implementation. In Uppaal the Task automaton can be in a locked location, however there is no Just-preempted and just-dropped indicators as seen in the Figure 2.5 from preliminaries. This was an additional behaviour that was added in order to not have circular behaviour, meaning that a task that has just been started can be dropped and started again within the same time.

Check dependencies In Uppaal we did handled dependencies by defining a counter for the dependencies. Every time a task completes it increments the dependency counter. Then a given task can only start if its dependency counters are equal to the amount specified for the task within its opportunity. Then the counter is reset once started.

In Rust and the semantics the dependencies are handled by saving the amount of completions the tasks that is depended on once completed. When a task needs to start it gets the difference between its own tracked completions with the current completions of the tasks it is dependent on and compares that with the defined dependencies in the system.

Enough power In Uppaal when the task template takes the transition from the location waiting to running it uses a pre-calculated value for the amount of power needed to finish to determine if there is a enough power. This value were calculated and passed a parameter for the task by us, manually. In the semantics and in the Rust implementation we simulate the start of the task until it has been completed and checks during that there is enough power until completion of the task.

Can lock In Uppaal this behaviour was given by synchronizing with tasks in the waiting location over the task_locked channel. If all tasks that needed to be locked were in the waiting location, the task could lock them all and hereby start. In the semantics and Rust this behaviour is defined by having Boolean's for a task that determines if it is locked or not. If a task is not running it can be locked.

Component availability Within Uppaal and the semantics a task can only start of the components it needs are available to it. In Uppaal this behaviour is represented by the automaton Component where it can be either active or idle. Therefore when starting a task it must activate the components needed, respectively, and if they are already active the task can not start. In the Semantics we have taken another approach. We look at the running tasks and from there derive which components that are active when we want to start a task. The implementation in Rust is more in line with the semantics than the Uppaal implementation, however, the resulting behaviour is the same.

Within opportunity In the Uppaal implementation this was checked by the Opportunity template, used a synchronize channel to prompt the task automaton to being able to start. In the semantics and in the Rust implementation this is performed by a function that checks for the defined intervals for a task whether or not it is within its opportunity. Furthermore, both the Uppaal, semantics and the Rust implementation checks for if the given task can complete within the defined opportunity.

Listing 7 and 8 in appendix A.5 shows the rust implementation of all of these checks. We check for all tasks whether or not they can start and return the set of id's that can start, which is later used in order to generate new states.

5.3.2 Start a Task

When a task is started the following is things happens:

Duration counter In Uppaal this was a parameter that was passed in the initialization of a task, and manually calculated. In the implementation and semantics we calculate the duration, as defined in the semantics.

Running boolean In Uppaal this was represented by transitioning to the running state, where in doing so it activates the action, which then activates the tasks.

Apply locks If the task has locks defined it locks all the given tasks. This was done over synchronization channels in Uppaal, but in the semantics and the implementation we apply the lock_task function to the tasks that needs to be locked.

Listing 9 in appendix shows the Rust code that starts a task.

5.3.3 Can Preempted Task Start

There is also the case of when a preempted task is started. When doing so, we need to only check for two things:

Can task lock We need to check whether or not the tasks that should be locked, can be locked. In Uppaal this behaviour is encompassed in the preempted location. In order to transition back to running it must lock the needed tasks. In our implementation and semantics the check is the same as the one when starting a task.

Component availability The same check as when starting a task, where we check if the components needed are available

Just preempted This is done in order to not start the task again within the same time frame.

The code for starting both preempted and available tasks is in the appendix Listing 13

When the function start is applied to a taskstatus it checks whether or not the counter is equal to zero. If it is the counter is set to the duration of the task, if not, only the Boolean indicating that it is running is set to true.

5.3.4 Preempt a task

When we check whether or not a task can be preempted we do the following:

Can complete We want to make sure that if it is preempted it can still finish within its interval. In the Uppaal implementation this check was not performed, and could result in behaviour where the task was left in the preempted location.

Just preempted Again, in order to remove circular behaviour where we can endlessly transition from preempt to start to preempt etc.

Listing 10 in appendix shows the code that determines whether or not the task can be preempted.

Listing 14 in the appendix shows the code for preempting a task.

When a task is preempted the Boolean indicating whether or not the task is running is set to false. This will later be shown to ensure that when time transitions the counter for the preempted tasks will not decrement.

5.3.5 Drop a task

When dropping a task, all we need to check for is if the task is defined as droppable in the system. If that is the case, we can drop the task. Furthermore, when a task is dropped we also need to unlock the tasks that it locked, if any. It is very similar to the Uppaal implementation, as there was a Boolean value passed in the instantiation of the template indicating if the task could be dropped as seen in preliminaries figure 2.5

The code that drops a task is shown in appendix Listing 11.

5.3.6 Transition in Time

In Uppaal time could not pass if the battery was depleted, as seen in the timer template in figure 2.7 in the preliminaries. Because the semantics and the implementation is defined differently we need to make some extra checks in order to determine if time can pass:

Component overlap In the Uppaal implementation, if we had two tasks running at the same time where each task had actions that shared components at the same time, Uppaal would simply stop the trace there, as the activation of the components would not succeed when starting the next action. In the implementation and semantics, however, we have to make this check explicitly.

Check preempted tasks In the Uppaal implementation we did not take into consideration if tasks could complete if preempted. This is a check that we have decided to add in order to not start a preempted tasks outside its intervals.

Check power In Uppaal as seem in the aforementioned figure 2.7 if the battery depleted it would transition to the location *battery depleted* resulting in a deadlock of the system i.e. meaning that system cannot progress. In the semantics and in the implementation we check for whether or not the battery will deplete if we take a time transition, and of it does, time is not allowed to pass.

The implementation of this behaviour can be seen in appendix Listing 12. If all of these checks are upheld, the time can pass. Listing 15 shows the code for when time passes on a state.

5.4 Schedule Generator

Now that the model is implemented, the schedule generator can be implemented. We will have two ways of generating schedules. We have a greedy naive algorithm that starts a task whenever possible. Secondly we have a Branch and Bound implementation where we define a heuristic in an attempt to gain more optimale schedules.

Firstly, however, we present the general approach as to how the state space generation works.

5.4.1 Generating States

In order to generate all possible states that derives from a specific state, a function named generate_States is created. Listing 16 in the appendix shows the function.

This function returns a vector containing all the states that can be derived from the given state.

5.4.2 Generating the State-Space

In order to generate the state-space a crate called id_tree is used. A crate in rust terms is an external library. This crate is a tree structure data type, where nodes can contain any type of data where it ,in our case, will contain a state. A node can have any amount of children, which is needed as there will be states that can generate more that one state.

The tree is initialised as shown in Listing 4 where the tree is initialized along with the first root node.

```
let mut tree: Tree<State> = TreeBuilder::new()
    .build();

let root_id: NodeId =
↪   tree.insert(Node::new(state),AsRoot).unwrap();
```

Listing 4: Initialization of the tree and root node

The state that is put into the root node is the initial state based on a given system. Because we get the sub-states in batches, meaning that we get more than one, we need to maintain a queue. This queue is of type VecDeque, which is a vector where popping can occour from both the back and the front, meaning that we can determine whether or not we want to make the tree in a breadth first approach or in a depth-first approach.

So, what is done is that the first set of states that are generated from the initial states is put in a node and this node is then added to the tree under the root node, and added to the queue as Listing 5 shows.

```
let generated_states = system.generate_states(state.clone());

for state_c in generated_states {
    id_pointer_list.push_back(tree.insert(Node::new(state_c),
↪   UnderNode(&root_id)).unwrap());
}
```

Listing 5: The first set of generated states are added to the queue id_pointer_list

Now the queue is initialized and the tree, i.e. the state-space, is ready to be generated.

5.4.3 High Throughput Trace

Now that the way a tree is generated is defined, we can define an approach that quickly will lead to a schedule, however not a guaranteed optimal schedule.

Listing 17 in the appendix shows the code for generating the trace. We have a check that compares two states. A better state in this approach is one where a task is started. Meaning that if there is a state where a task is started among the generated states, that state is considered the best.

5.4.4 Branch and Bound

In our branch and bound we add another list of nodes, where the leafs of the tree will be. A leaf node is when the state that has has been generated has its global time equal to the termination time specified for the system.

Another addition is that we redefine a state to contain an upper bound as well. This upper bound will be decremented every time a task is available but not taken, hereby increasing the sentiment that we want to start as many tasks as possible.

Listing 18 shows the code that generates a statespace using a branch and bound algorithm.

The heuristic of when to decrement the upper bound is shown in the code in Appendix Listing 18 and 19. When time passes we check whether for the amount of starts has decreases in the generated state. This amount is saved to a counter. If it has decreased we check for if any of the tasks not running in the parent state are started in the generated state. If it has we decrement the counter for each task that has started between the two time transitions. We lastly return the counter times minus one, as that will be the amount subtracted to the upper bounds.

These are the two approaches that have been implemented at the given time, and both of these will be used to generate schedules in the sections 6 and 7.

5.5 Gantt Chart Generator

Once the core system has generated a viable trace, it is output as a JSON object containing the description of the system, as well as the trace itself. In order to generate the visual representations shown in Section 6, this output is put through a script which generates the image output. Due to its maturity, relative to the available Rust libraries, we use Python and the Matplotlib library to accomplish this task. The script runs through the trace array and extracts the state of charge and status of each task at every point in the trace. This data is then used to generate a graph of the expected state of charge during the execution of schedule, as well as a colour coded status bar for each task in the system, indicating whether or not a task is idle, preempted, locked, or running at any given time. As the Matplotlib library does not allow creating multi-colour bars as described, we accomplish this by successively overlaying bars on each other, thereby imitating the desired behaviour. A shortcoming of this approach is that it does take significantly longer to generate the Gantt chart this way, making the system loose a bit of its interactivity. As we create the array containing all the state of charge values in this subsystem, and we have the desired library support, we calculate a wear score using the WSF described in Section 2.2.

5.6 Code Generator

As a proof of concept, we have implemented a crude code generator to show how one might use a generated trace to also automatically generate executable code that may be placed on an actual system. Just as the Gantt chart generator, the code generator runs through all the states of a trace, outputting the desired code for each change in a tasks status. This implementation outputs a pseudo-code in the form of ‘Time n: Starting/Stopping task: *name*‘, but could be changed to output a hard-coded schedule of function calls starting and stopping the corresponding tasks, a small example of this generated code can be seen in Listing A.1.

This section covered the implementation of the different parts of tool. We have implemented:

- A parser
- The model
- A Schedule generator
- Gantt Chart Generator

The feature of using queries to generate schedules has not been implemented at the given time.

Chapter 6

Test of previous program

In this chapter we will use the same model of the GomX-3 satellite used in the Uppaal implementation from the previous report. We will firstly compare to see if we are able to generate schedules that entails the same behaviour as the one successfully generated in Uppaal. Thereafter we will alter the model in order to see which kind of schedules that the modifications will result in.

We will therefore:

- Compare the exact model
- Compare performance as to see for how long a time span we can generate schedules
- Limit the battery and tasks in ways to show how the schedule changes
- Do changes in order to compare wear score functions

6.0.1 Generating schedules

We defined the same system in our implementation and attempted the same approach where we would, at least, monitor the time it took to generate the state-space and find a trace.

There is a difference, however, as we did not implement the feature where alternative components can be defined. So in order to circumvent this behaviour we define the action that had alternative defined, to only be using the alternative component.

The branch and bound approach results in some not optimal schedules when the intervals for tasks are defined to be longer that the duration of the task. This is likely because of the way we reduce the upper bound. The resulting schedule can be seen in Appendix A.2.

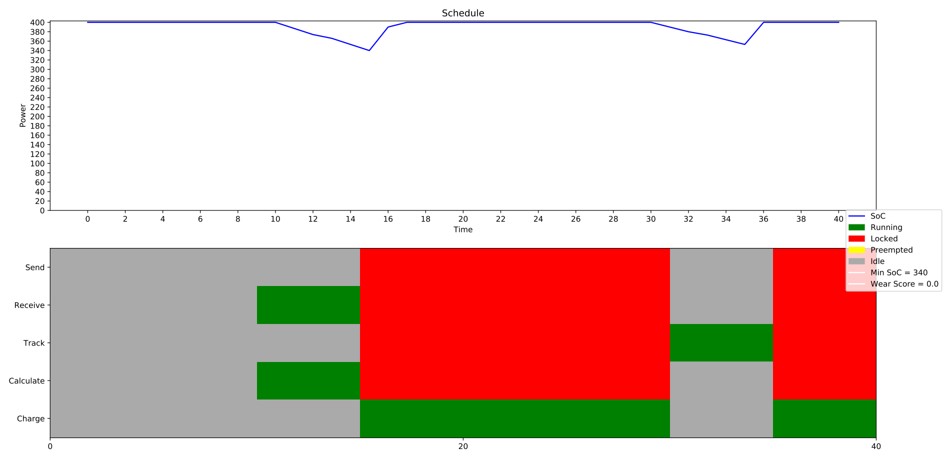


Figure 6.1: The generated schedule for one orbit

In Appendix A.7 the trace derived from Uppaal in [2] is shown. The difference between the two schedules is that in Uppaal task charge stops charging once the battery is full. However, it is preempted and left in a preemtped state thoughout the rest of the trace where in the schedule in Figure 6.1 task charge runs as many times as possible.

The Gantt chart in Figure 6.1 is the resulting schedule for one orbit based in the high thoughput approach.

In order to ensure that the dependency behaviour is correct we generate a schedule for a span of 6 orbits, as shown in figure 6.2, this time with a shortened interval for the task charge, so it matches the duration of the task.

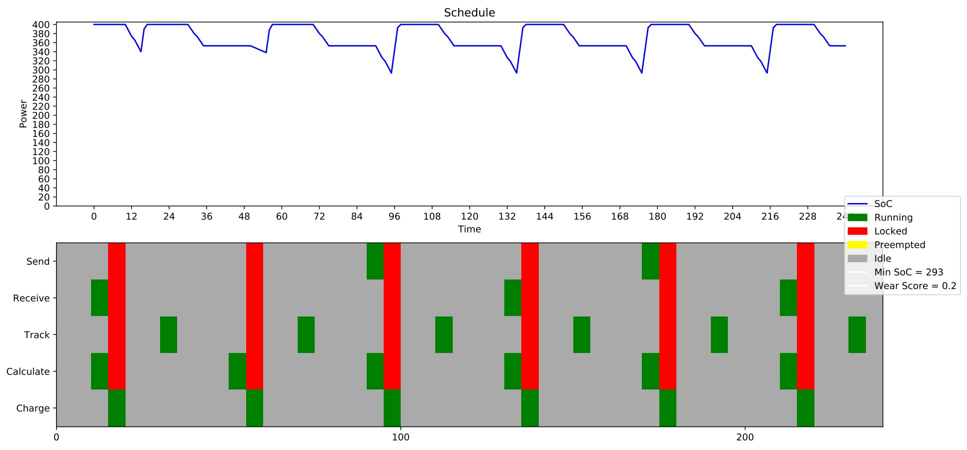


Figure 6.2: The generated schedule for six orbits

As can be seen in Figure 6.2 the track task runs twice before the send task is executed and we can generate an even longer schedule to show that it works more than the one time. Both high throughput approach and branch and bound resulted in this schedule.

6.0.2 Comparing Performance

The implementation does not contain the same guarantees as Uppaal provides which is why comparing with the performance of Uppaal is merely to say what the implementation is able and unable to. In the previous report the results from the verification where as follows:

Table 6.1: Results, from the previous report

Orbits	Nr. of completions each	Time	Memory
3	1	1.023 s	25.900 KB
5	2	24.292 s	406.316 KB
7	3	81.494 s	1.266.984 KB
9	4	112.210 s	3.107.544 KB
11	5	205.620 s	6.146.136 KB
13	6	Out of memory	Out of memory

With our implementation and the highest throughput method we are able to query past 13 orbits with the high throughput approach and no more than 13with the branch and bound approach. Our initial goal was to be able to generate a schedule for three orbits, as that was the duration of the missions specified for the GomX-3 satellite.

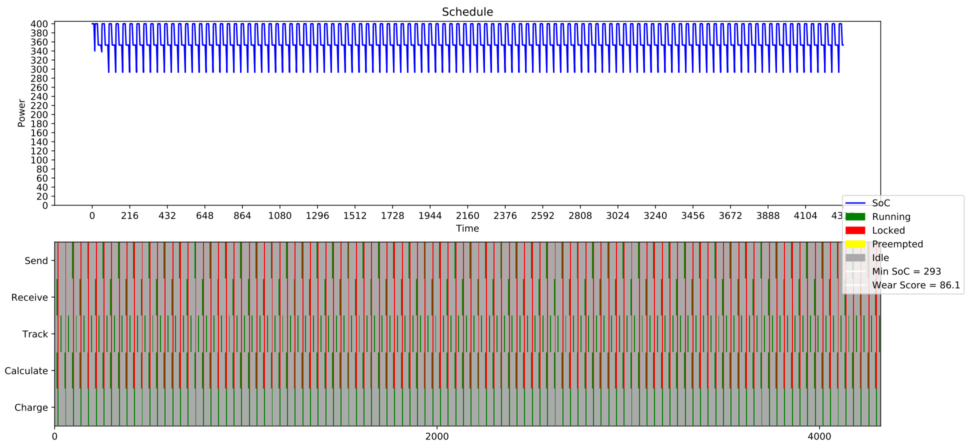


Figure 6.3: The generated schedule for three days

Figure 6.3 shows how a schedule for three days looks for this given system. It is a compact schedule in this page, however it shows that we are able to generate schedules for a span of 3 days. Because we are able to choose which state is better we are able to reduce the state space into a single trace. However, there are no guarantees that can be provided in terms of it is the most optimal schedule, and the branch and bound approach did not take much space, but were unable to generate a schedule for more than 13 orbits. At 14 or more we terminated the generation at it after 7 hours at 14 orbits had not produced results.

To summarize we are able to generate schedules that span for three days, however these schedules provides no guarantees concerning if it is the most optimal schedule. The following schedules throughout this section will be generated from the model where task charge has an interval with the same length as its duration.

6.0.3 Limiting the battery

In this section we will show what happens when the system is limited on its battery capacity and how often it can charge.

Figure 6.4 shows a system that has a lower battery capacity compared to the previous system. However, it is still able to complete its tasks in the same manner.

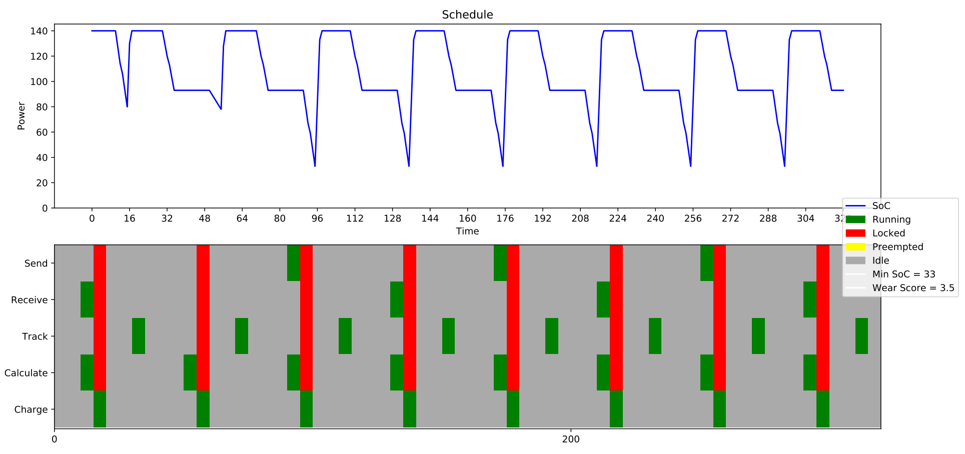


Figure 6.4: Limiting the battery

Therefore, in figure 6.5 we have further reduced the capacity and as can be seen it skips the send in this schedule. It does so for the reason that there is not enough power on the battery for it to complete. The resulting schedules are the same with branch and bound and high throughput approach.

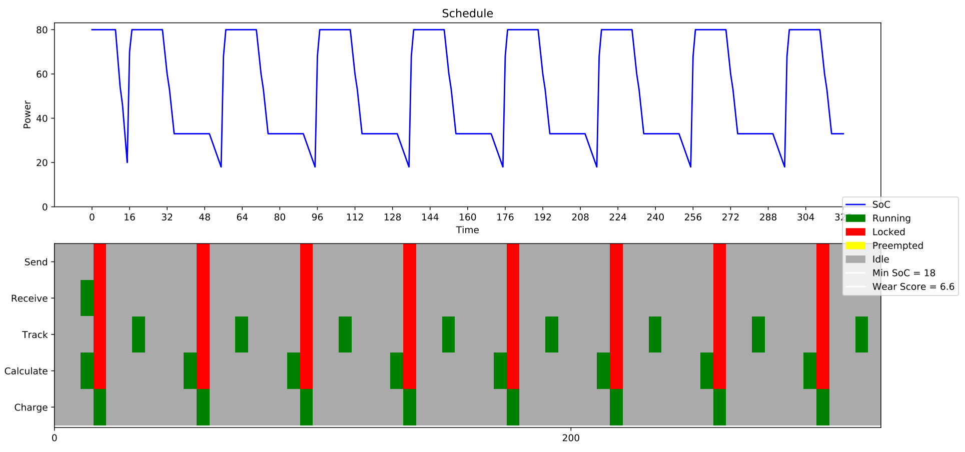


Figure 6.5: Limiting the battery to force the skips of tasks

The reason is that we are trying to generate the trace with the highest throughput and for the task send to be started, the system would have to skip both track and calculate, in order for send to be started, meaning that the throughput of completed tasks would be lower.

6.0.4 Comparing Wear Scores

In this section we generate a schedule and thereafter remove a task in order to lessen the work the system has to perform and generate a new schedule. Then we compare the scores.

Figure 6.7 shows a schedule with a wear score of 3.5

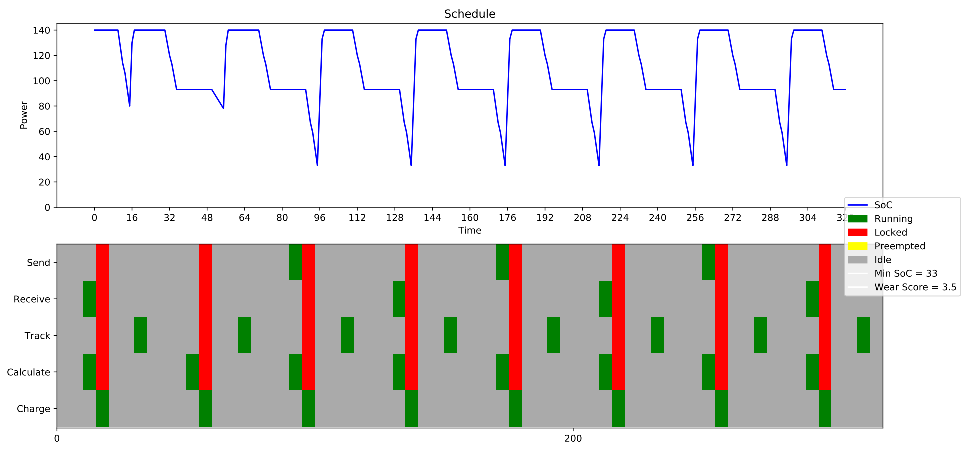


Figure 6.6: A schedule with a wear score of

Figure 6.7 shows a schedule of the same system, where we have omitted the task calculate which lessens the amount of tasks it can complete. The wear score of the second schedule is 2.9 which means that this system wears the battery less compared to the schedule in figure 6.6.

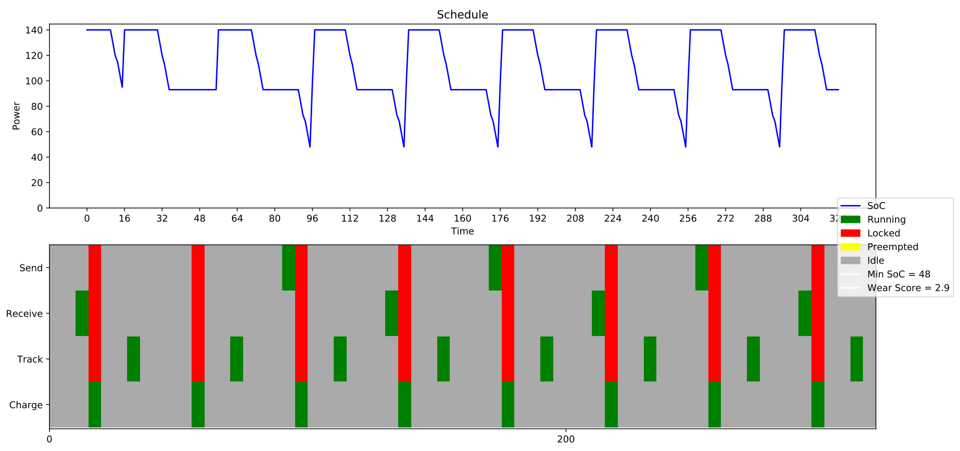


Figure 6.7: A schedule where the task calculate has been omitted

Chapter 7

Beyond Satellite Scheduling

In the report [2] an attempt was done to model a satellite, however there might be a series of other systems that can be modelled in BATTCIO. So on this chapter we will go through brief ideas as to which kinds of systems that might be modelled in BATTCIO.

7.1 Alternative Resource

The Battery definition in the modelling formalism has until now been regarded as a power resource powering battery dependent systems.

However, if we were to change this perception to one that considered the battery definition a resource definition, many kinds of systems could be modelled in this modelling formalism. For instance, lets say that a construction company are starting a building project, where they have a limited amount of workers that can perform a certain task. One such task can be laying the foundation. To do this they would need a resource, this resource would be cement in this case. What could be used

We would define a component that consumes the resource and one that provides it in the correct rate. Since we can declare a discrete battery this can also be considered a discrete resource.

```
Component Consumption(5);
Component Delivery(−20);

Action Work{Components: {Consumption} Duration 8};
Action Deliver{Components: {Delivery} Duration 1};

Task Working{Actions: [Work]};
Task Delivering{Actions: [Deliver]};

Interval WorkHours(0,8);

Opportunity ( Intervals: WorkHours
              Task: Working );
```

```
Opportunity ( Intervals: WorkHours
              Task: Delivering);

Battery(Capacity: 40
        InitialCharge: 10
        Type: Discrete);

Start(0);
Termination(0);
```

In this model we would define a component that refilled with concrete, and we would define opportunities for when this concrete could be delivered. The formalism would then help determine when these shipments of concrete would be needed. A model like this would be useful in the case where the construction project would be in a situation where there is limitations to the storage of concrete and we would therefore be able to “recharge” as little as needed given the query language.

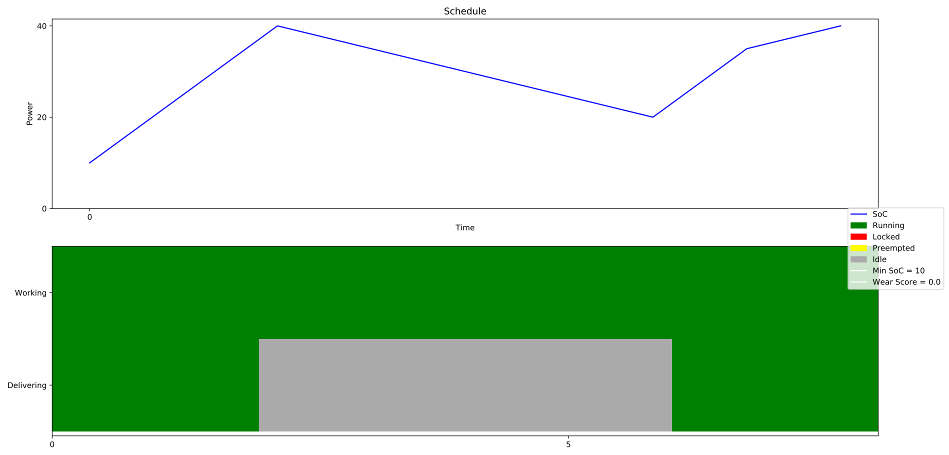


Figure 7.1: Gantt chart generated by BATTCIO, showing a schedule for concrete delivery with the high throughput approach

Figure 7.1 shows the schedule generated when using the high throughput approach.

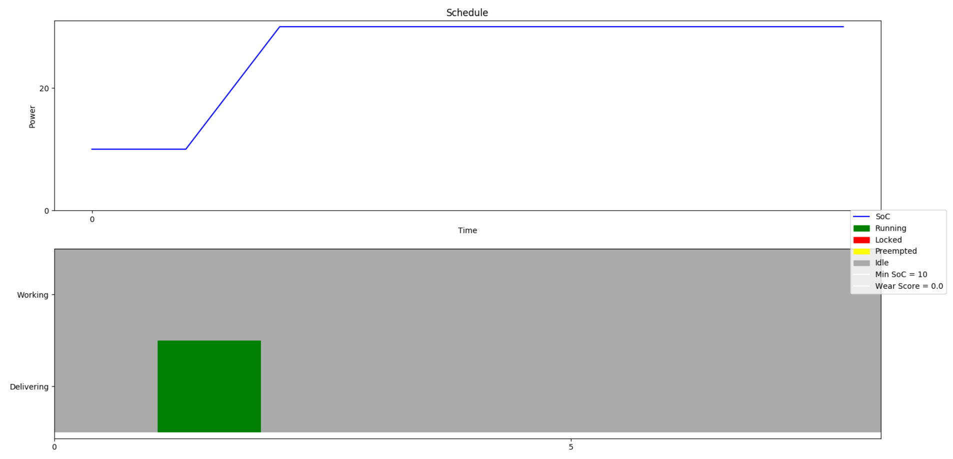


Figure 7.2: Gantt chart generated by BATTCIO, showing a schedule for concrete delivery with the branch and bound approach

Figure 7.2 shows the schedule generated using the branch and bound method. The reason for this schedule is from the method with which we reduce the upper bound as the transitions happens. We measure the among of available starts for each time and it has decreased we check for if a task has started. For each task that has started we decrement the amount we want to lower the upper bound.

So, when we have a system such as this where two tasks can start at the same time and again one time transition later, the amount of tasks that can start will be the same throughout except when we pass the first interval. This results in a schedule such as this one. We would need to redefine the heuristics of lowering the upper bounds in order to gain better schedules for models like this one.

7.2 Self Driving Electrical Vehicles

Imagine, some time in the future, a company that provides services with self driving cars. Electrical cars power consumption differs dependent on the kind of driving that it performs. So, without any deep analysis of how exactly the cars power consumption differs we speculate that city driving takes the least amount of power for driving, country roads take more and highways take the most.

The idea is that if we have a company with a high amount of self driving cars, but the company wants to share recharging stations between cars, the cars will be allotted specific time windows in which they can recharge. So, in BATTCIO a model of a car could be modelled as listing 7.1 shows. The car has three *components* that we set as the driving methods, i.e. city, country roads and highway. We assume that each kind of driving has a certain function, for instance, in the city the car is used a taxi to transport people within the city.

Country and highways are used for transporting goods or long taxi services. Electrical cars are moving towards faster and faster charge times and in this model we are assuming that a car can fully recharge in an hour. The time granularity is set to 1 hour. We would have to lock all other tasks when they are performed. This can be done in two ways in the model. We can either explicitly declare the locks, or we can do it component wise by saying we have a components for city driving. A component for country and one for highway driving. Or, we can define three components that are adding up in the consumption of the battery. So that we have a "base" component used for driving, then we have level 2 and level 3 components.

Listing 7.1 shows the model with lock usage and listing 7.2 shows the model with the use of implicit locks through components. Figure 7.3 shows the generated schedule for the model. As can be seen the only task that locks correctly is the last declared task in the listing. That is due to the way the parser generates the system internally, where if the task is not defined when referenced, it will ignore it.

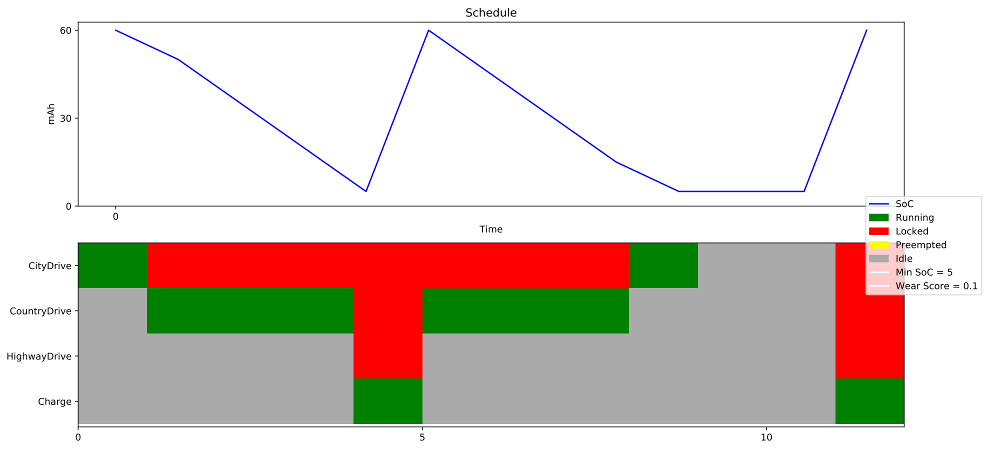


Figure 7.3: Gantt chart generated by BATTCIO, showing a schedule for a self driving vehicle.

Listing 7.1 : The model of a driving car

```
Component City(10);
Component Country(15);
Component Highway(20);
Component Charge(−60);

Action CityDriving(Components: {City} Duration: 1);
Action CountryDriving(Components:{Country} Duration: 1);
Action HighwayDriving(Components:{Highway} Duration: 1);
Action Charge(Components:{Charge} Duration: 1);
```

```
Task CityDrive(Actions: [CityDriving]
               Locks: [ CountryDrive, HighwayDrive ]);
Task CountryDrive(Actions: [CountryDriving]
                  Locks: [ CityDrive, HighwayDrive ]);
Task HighwayDrive(Actions: [HighwayDriving]
                   Locks: [ CityDrive, CountryDrive ]);

Task Charge(Actions: [Charge]
            Locks: [ CityDrive, CountryDrive, HighwayDrive ]);

Interval MissionTime(0,12);
Interval ChargeOpp1(4,5);
Interval ChargeOpp2(11,12);

Opportunity(Intervals: MissionTime
            Task: CityDrive);

Opportunity(Intervals: MissionTime
            Task: CountryDrive);

Opportunity(Intervals: MissionTime
            Task: HighwayDriving);

Opportunity(Intervals: ChargeOpp1
           Task: Charge);

Opportunity(Intervals: ChargeOpp2
           Task: Charge);

Battery(Capacity: 60
        InitialCharge:60
        Type: Discrete);

Start(0);
Termination(12);
```

As can be seen listing 7.2 shows the model with implicit locks through the component usage. Here the there it is not shown that the tasks lock each other, but they are not executed simultaneously meaning that they would not be able to have started. Figure 7.4 shows the generated schedule.

Listing 7.2 : The Car case with implicit locks through components

```
Component Drive(10);
Component LevelTwo(5);
Component LevelThree(5);
Component Charge(−80);

Action CityDriving(Components:{Drive} Duration: 1);
Action CountryDriving(Components:{Drive, LevelTwo} Duration: 1);
Action HighwayDriving(Components: {Drive, levelTwo, levelThree} Duration: 1);
Action Charge(Components: {Drive, levelTwo, levelThree, Charge} Duration: 1);

Task CityDrive(Actions: [CityDriving]);
Task CountryDrive(Actions: [CountryDriving]);
Task HigwayDriving(Actions: [HighwayDriving]);

Task Charge(Actions: [Charge]);


Interval MissionTime(0,24);
Interval ChargeOppOne(4,5);
Interval ChargeOppTwo(11,12);
Interval ChargeOppThree(18,19);

Opportunity(Intervals: MissionTime
            Task: CityDrive);

Opportunity(Intervals: MissionTime
            Task: CountryDrive);

Opportunity(Intervals: MissionTime
            Task: HighwayDriving);

Opportunity(Intervals: ChargeOppOne
           Task: Charge);

Opportunity(Intervals: ChargeOppTwo
           Task: Charge);

Opportunity(Intervals: ChargeOppThree
           Task: Charge);



Battery(Capacity: 60
        InitialCharge:60
        Type: Discrete);

Start(0);
Termination(24);
```

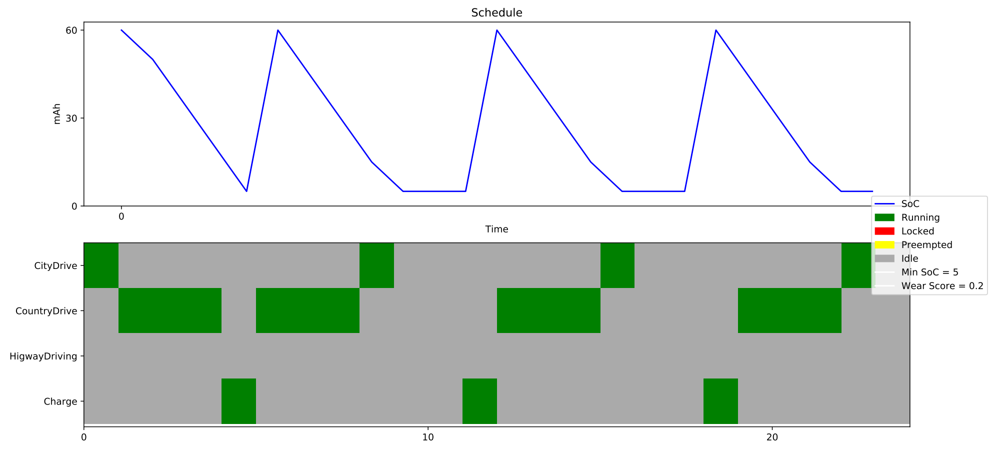


Figure 7.4: Gantt chart generated by BATTCIO, using implicit locks, showing a schedule for a self driving vehicle.

7.3 Generic Planning

Another aspect that we will draw attention to is the use of the model for planning in general. What can be done here is that we can define components with a power consumption of zero and then attempt to make a schedule. There is a risk connected to this being that since the battery is not limiting the state space anymore, we might produce a too big state space. However, we will still make a small example system in which we try to generate a schedule for a given system.

The system that we can try to attempt to create is a schedule for workers in the a working space. The example could be a fair where we have people working at different parts of the fair.

Listing 7.3 : The model of a working space

```
Component workerOne   (0);
Component workerTwo   (0);
Component workerThree(0);
Component workerFour  (0);

Action WorkOne   (Components:{workerOne   } Duration: 1);
Action WorkTwo   (Components:{workerTwo   } Duration: 1);
Action WorkThree(Components:{workerThree} Duration: 1);
Action WorkFour  (Components:{workerFour  } Duration: 1);

Task WorkOne   (Actions:[WorkOne   ]);
Task WorkTwo   (Actions:[WorkTwo   ]);
Task WorkThree(Actions:[WorkThree]);
Task WorkFour  (Actions:[WorkFour  ]);

Interval WorkHours(0,8);
```

```
Opportunity(Intervals: WorkHours
            Task: WorkOne);

Opportunity(Intervals: WorkHours
            Task: WorkTwo);

Opportunity(Intervals: WorkHours
            Task: WorkThree);

Opportunity(Intervals: WorkHours
            Task: WorkFour);


Battery(Capacity: 400
        InitialCharge: 400
        Type: Discrete);
Start(0);
Termination(8);
```

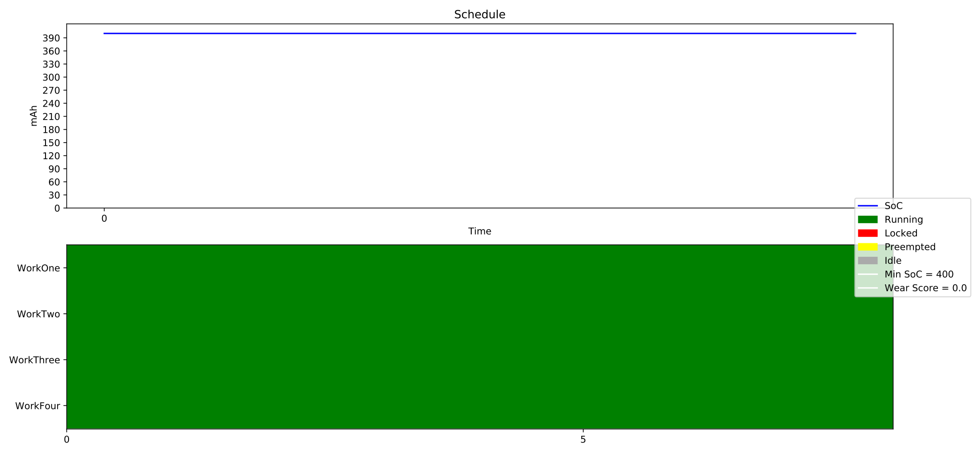


Figure 7.5: Gantt chart generated by BATTCIO, showing a schedule for a generic work space.

As can be seen the model as of now is limited in the sense that this model would generate a schedule where all the workers would work throughout the eight hours. However, if the feature where alternative components could be defined where implemented, a more dynamic schedule could be generated. However, some form of visual indication as to which components are used by the task would also need to be shown in the Gantt chart.

Chapter 8

Epilogue

8.1 Evaluation

In this section we will evaluate on the project and the status of the implementation.

8.2 Overview

Initially, in the previous project, we made an implementation in Uppaal with which we were able to schedule up until 11 orbits.

Based on that we determined that we would implement a seperate model and schedule generator in order to attempt to overcome the obsticale of the high memory consumption.

We chose to go forward with a seperate implementation for several reasons. One is because it would give us more control over the way we generate and traverse the state space, meaning that we would be able to more freely determine the heuristics of the approach traversing and generating the state space. For instance when implementing the branch and bound algorithm. However, this also increased the complexity of the implementation to us. Furthermore a compiled language could grant some additional computing speed as there would be no need for interpreting as well. That would however, require that the code written is also efficient. And lastly, it would be easier for us to implement other battery models than the discreete model, as Uppaal sets restraints when working with Real numbers.

We chose to do the implementation in a language that were unfamiliar to us, but provided us with memory safety, a possibility for safe multi threading, a compiled language and compile time error handling. These factors were the main beneficiaries of the choice of language. The unfamiliarity with the language wound up costing us time by the process of learning to write in it. However with the memory safety that it provided and compile time errors we can speculate that implementing it in another language could have given us other problems that would have wound up costing the same amount of time.

As of now we are able to generate schedules in two ways from a system. The first way we do this, i withs a naive greedy approach that simply starts as many tasks as possible. For some systems this results in good, but in more complex systems where there are more component overlapping and opportunity overlapping the situation might change. A system might in the end generate a better schedule if it waited some time before starting certain tasks.

The other way we do it is with a branch and bound where the heuristic for lowering the upper bound does in some cases result in sub-optimal schedules dependent on the model.

8.2.1 Missing alternative components

As of now the feature where alternative components for an action can be defined is missing. In order to have this be a part of the model, the semantics would need to have these added, along with the implementation being altered as well in order to accomodate this feature.

If this where implemented, the model would be a better fitted to generate schedules for a work place, as in Section 7, where it would be able to choose among all the available components. In the given case a compnent is a worker.

This would also generate the need for displaying in the gantt chart which components are being used by the given task.

This also resulted in the fact that we had to do a small rewrite of the test system used in the previous report in order to acheive comparable schedules.

8.2.2 Missing skipppable

Another difference from the Uppaal implementation is that the tasks can be defined as skippable or not. This could eventually be defined in the query instead, where a query can be defined to run a given taks the amount of possible times it can.

8.3 Conclusion

In this report we have defined the semantics for BATTCIO, defined a query language for the model and implemented a parser, the model and a schedule generator for the model. The semantics encompasses almost similar behaviour to the Uppaal implementation in the previous report with few altercations [2]. The parser, model and schedule generator are all implemented in the language RUST that provides memory safety and compile time errors. We have throughout the report made use of a conceptual GomX-3 model defined in the BATTCIO formalism in the previous report [2].

In order to generate the schedules, two approaches were taken; A naive approach that started a task if possible and a branch and bound approach that more extensively sought to generate an optimal schedule. Both of these approaches are able to generate schedules for the GomX-3 model where the naive approach were able to generate schedules that spans for more than three days, however with no guarantees of it being an optimal schedule. However, the generation is done within a few seconds. The branch and bound approach were able to generate schedules in span for up to 13 orbits, before reaching a situation where the time needed to generate schedules for more than that is unrealistic.

The schedules are outputted as a JSON data format that can be passed to the Gantt chart generator and code generator, that we have developed. The Gantt chart generator is written in Python and uses matplotlib to draw the charts.Additionally the state of charge over time is also shown in the generated charts. Furthermore, this generator also generates the wear score function for the task that can be used to compare schedules and choose the schedule that wears the least on the battery. The code generator is written in Python as well.

We were able to generate schedules for the GomX-3 satellite, that encompassed the same behaviour as the Uppaal implementation. Furthermore we made some altercations to the model where we limited the battery in order to skip tasks which were successful. Furthermore, we removed a task from the system in order to see if the wear score was lower for the schedule that did less work, which is was.

We furtherly explored other systems that could be modelled in BATTCIO and generated schedules af varying quality from those. Among these explorations where planning for self driving vehicles, planning when to receive a shipment of a needed resource and generic planning. This exploration gave light to the limitations of the formalism and implementation as there were schedules generated were sub-optimal.

Overall, the tool is in a proof on concept state where further development is required to have it working for cases outside the gomx3 case. However, it did succeed in generating schedules for the initial problem area i.e. satellites.

8.4 Future Work

This section will present what is considered important next steps if further development of the tool were to happen.

8.5 QueryLanguage implementation

As of now the query language is not yet implemented. If we were to implement it we would firstly find a way to define an euclidian distance between two states and use that in a branch and bound approach to reach the state.

8.5.1 Branch and Bound

Redefining the heuristics of the branch and bound method would be the next step. Finding an approch that consistently results in optimal schedules would require is to exactly figure out when the upper bound should decrease. We would need to determine what the best choice and not decrease the upper bound if the best choice is taken. As of now there are instances where not starting a task results in the same upper bound as starting a task as shown in Figure A.2. Here the choice of starting tasks does not give a better upper bound than not starting one.

8.5.2 More extensive testing

As if now, the implementation is in a *proof of concept* state meaning that the functionality for generating a schedule is implemented, however generating and comparing several schedules has not yet been implemented. As of now it initially seems that the starting of tasks, locks and finishing of tasks are working as exptected. However, we would need to make systems where the use of preemption and drops would result in a better schedule.

8.5.3 Implementing KiBaM

The current state of the implementation does not support a Kinetic Battery Model. That is part because we found it interesting to make a model that we could compare more or less directly with the Uppaal implementation and it would require another semantic rule set, as the it requires real numbers. In the current semantics, the behaviour is based on discrete numbers.

8.5.4 Bundling

As of now, the implementation is not developed with a GUI. This means that when a system is defined it must be done so in a seperate text editor where no highlighting and syntactical analysis is done while defining the system. We would need to develop a dedicated text editor to acheive highlighting and static analysis of the system. We would also need some editor to define queries in, along with syntax highlighting and static analysis. Furthermore the gantt charts are generated by a seperate program. In order to have a easily useable tool we could bundle the text editor and the query editor with the gantt chart generator to a single program where all three parts are incorporated.

8.5.5 The syntax

We needed a redefined syntax for the current implementation based on the fact that alternative components cannot be declared. However, we have also noticed that there are some constructions that could be changed for easier definitions.

For instance, when declaring intervals, only one interval can be declared per decleration. In a small system such as the one used in the test program, no problems occour. However, if a system spans over longer time and therefore has many intervals bound to an opportunity it is unnecessary to define the intervals to many different identifiers. What could be done is to define more that one set of intervals per interval declaration:

```
Interval name([num1,num2],[num3,num4]);
```

Furthermore a small interval calculus has been proposed and can be seen in Appendix A.1

8.5.6 The Parser

With present implementation of the parser, each statement is processed by a regular expression, that will either match and add the statement to the system, or it will not match and ignore the statement all together. This puts all the responsibility of program correctness on the end user, as a malformed program may not produce any meaningful error messages until it is executed. In a fully developed version of the toolkit, a proper lexer and parser should be implemented, to check for program validity and inform the user accordingly. Additionally, the parser leaves it to the user to ensure that identifiers are correct and will leave them out of the system if this is not the case, leading to unexpected schedules being generated.

8.5.7 Testing with actual systems

The systems that have been tested are not representations of real systems, only concepts. This is meaningful when implementing the checker and schedules. However, models that really represents the systems may not have a state space as big, because the battery would more easily be emptied. Our conceptual test model of the GomX-3 satellite has a big battery that is charged fully within one charge opportunity, meaning that there is not much balancing that needs to be done on the battery. Meaning that the battery is a factor concerning the state-space, where a high battery capacity and high charge rate relative to the power a task consumes allows for many tasks to be executed. This was tested, as we limited the capacity of the battery and the amount it can charge within a charge cycle.

8.5.8 Reconsider Uppaal implementation

With the semantics defined, it could prove beneficial to make a new implementation of the Uppaal model, more akin to the semantics. Meaning that instead of having templates for components, actions, opportuniites, etc. we could make the state have the same in Uppaal as in the semantics. In the semantics we have attempted to make balance it towards computational decision making instead of memory decision making. What it means is that instead of having components represented in the state as an automaton, we derive what the used components are instead, just as in the semantics. In essence an attempt to decrease the amounf of automata represented in the state. This was not done as the Rust implementation and Semantics were developed and defined concurrently, whereas the idea came to us when it was too late to attempt a new Uppaal implementation.

Bibliography

[1] Morten Bisgaard et al. “Battery-Aware scheduling in Low Orbit: The GomX-3 Case”. In: *Aalborg University and Saarland University* (2016).

[2] Kristoffer Brodersen and Mads Broen Nielsen. “Batteries Included”. MA thesis. Aalborg University, 2017.

[3] GomSpace. *GOMX-3 (GomSpace Express-3).* Website. `https://directory.eoportal.org/web/eoportal/satellite-missions/g/gomx-3`.

[4] M.R. Jongerden and B.R Haverkort. “Battery Modelling”. In: - (-).

[5] kostya. *Benchmarks.* `https://github.com/kostya/benchmarks`. 2017.

[6] Rufflewind. *Graphical depiction of ownership and borrowing in Rust.* `https://rufflewind.com/2017-02-15/rust-move-copy-borrow`. Feb. 2017.

[7] Rust Team. *Rust FAQ.* `https://www.rust-lang.org/en-US/faq.html`. 2017.

[8] Erik Ramsgaard Wognsen et al. “A Score Function for Optimizing the Cycle-Life of Batttery-Powered Embedded Systems”. In: *International Conference on Formal Modeling and Analysis of Timed Systems* (2015).

Appendix A

Appendix

A.1 Interval Calculus

Interval Calculus Definition 1 *Define*

$$I_1 = Interval(S : t_1, E : t_2) \tag{A.1}$$

$$I_2 = Interval(S : t_3, E : t_4) \tag{A.2}$$

Where I_1 and I_2 are intevals, from time S to time E, and $t_1 \leq t_2$ and $t_3 \leq t_4$.

Interval Calculus Definition 2 *Difference*

$$I_3 = I_1 \setminus I_2 \tag{A.3}$$

if $t_3 \leq t_1 \wedge t_1 \leq t_4 \wedge t_4 < t_2$ then

$$I_3 = Interval(S : t_4, E : t_2) \tag{A.4}$$

if $t_1 < t_3 \wedge t_3 \leq t_2 \wedge t_2 < t_4$ then

$$I_3 = Interval(S : t_1, E : t_3) \tag{A.5}$$

if $t_1 < t_3 \wedge t_4 < t_2$ then

$$I_3 = \begin{cases} Interval(S : t_1, E : t_3) \\ Interval(S : t_4, E : t_2) \end{cases} \tag{A.6}$$

if $t_3 \leq t_1 \wedge t_2 \leq t_4$ *then*

$$I_3 = \varnothing \tag{A.7}$$

if $t_4 \leq t_1 \vee t_3 \geq t_2$ *then*

$$I_3 = Interval(S : t_1, E : t_2) \tag{A.8}$$

Interval Calculus Definition 3 *Union*

$$I_3 = I_1 \cup I_2 \tag{A.9}$$

if $t_1 \leq t_3 \wedge t_2 \geq t_3 \wedge t_2 \leq t_4$ *then*

$$I_3 = Interval(S : t_1, E : t_4) \tag{A.10}$$

if $t_3 \leq t_1 \wedge t_4 \geq t_1 \wedge t_4 \leq t_2$ *then*

$$I_3 = Interval(S : t_3, E : t_2) \tag{A.11}$$

if $t_1 \geq t_3 \wedge t_4 \geq t_2$ *then*

$$I_3 = Interval(S : t_3, E : t_4) \tag{A.12}$$

if $t_1 \leq t_3 \wedge t_2 \geq t_4$ *then*

$$I_3 = Interval(S : t_1, E : t_2) \tag{A.13}$$

if $t_2 < t_3 \vee t_4 < t_1$ *then*

$$I_3 = \begin{cases} Interval(S : t_1, E : t_2) \\ Interval(S : t_3, E : t_4) \end{cases} \tag{A.14}$$

Interval Calculus Definition 4 *Intersection*

$$I_3 = I_1 \cap I_2 \tag{A.15}$$

if $t_2 < t_3 \lor t_4 < t_1$ *then*

$$I_3 = \varnothing \tag{A.16}$$

if $t_3 \geq t_1 \land t_3 \leq t_2 \land t_2 \leq t_4$ *then*

$$I_3 = Interval(S : t_3, E : t_2) \tag{A.17}$$

if $t_1 \geq t_3 \land t_1 \leq t_4 \land t_2 \geq t_4$ *then*

$$I_3 = Interval(S : t_1, E : t_4) \tag{A.18}$$

A.2 Syntax

⟨*system*⟩ ::= ⟨*declarations*⟩

⟨*declarations*⟩ ::= ⟨*declaration*⟩
| ⟨*declarations*⟩ ⟨*declaration*⟩

⟨*declaration*⟩ ::= ⟨*opportunity*⟩ ';'
| ⟨*battery*⟩ ';'
| ⟨*time*⟩ ';'
| ⟨*component*⟩ ';'
| ⟨*action*⟩ ';'
| ⟨*task*⟩ ';'
| ⟨*interval*⟩ ';'

⟨*component*⟩ ::= 'Component' ⟨*identifier*⟩ '(' ⟨*cost*⟩ ')'

⟨*action*⟩ ::= 'Action' ⟨*identifier*⟩ '(' ⟨*actionparamslist*⟩ ')'

⟨*actionparamslist*⟩ ::= ⟨*actionparams*⟩
| ⟨*actionparamslist*⟩ '|' ⟨*actionparams*⟩

⟨*actionparam*⟩ ::= 'Components:' '[' ⟨*identifiers*⟩ ']'
| 'Duration:' ⟨*number*⟩

⟨*actionparams*⟩ ::= ⟨*actionparam*⟩
| ⟨*actionparams*⟩ ⟨*actionparam*⟩

⟨*task*⟩ ::= 'Task' ⟨*identifier*⟩ '(' ⟨*taskparams*⟩ ')'

⟨*taskparams*⟩ ::= ⟨*taskparam*⟩
  | ⟨*taskparams*⟩ ⟨*taskparam*⟩

⟨*taskparam*⟩ ::= 'Actions:' '[' ⟨*identifiers*⟩ ']'
  | 'Locks:' '[' ⟨*lockparams*⟩ ']' !Optional
  | 'Droppable:' ⟨*Bool*⟩ !Optional
  | 'Preemptable:' ⟨*Bool*⟩ !Optional

⟨*lockparams*⟩ ::= ⟨*identifiers*⟩
  | 'All'
  | 'AllExcept' ':' ⟨*identifiers*⟩

⟨*interval*⟩ ::= 'Interval' ⟨*identifier*⟩ '(' ⟨*number*⟩ ',' ⟨*number*⟩ ')'

⟨*opportunity*⟩ ::= 'Opportunity' '(' ⟨*opportunityparams*⟩ ')'

⟨*opportunityparams*⟩ ::= ⟨*opportunityparam*⟩
  | ⟨*opportunityparams*⟩ ⟨*opportunityparam*⟩

⟨*opportunityparam*⟩ ::= 'Intervals:' ⟨*identifiers*⟩
  | 'Task:' ⟨*identifier*⟩
  | 'Skippable:' ⟨*Bool*⟩
  | 'Dependencies:' ⟨*dependencyparams*⟩

⟨*dependencyparams*⟩ ::= ⟨*dependencyparam*⟩
  | ⟨*dependencyparams*⟩ ⟨*dependencyparam*⟩

⟨*dependencyparam*⟩ ::= ⟨*identifier*⟩ ':' ⟨*number*⟩

⟨*battery*⟩ ::= 'Battery' '(' ⟨*batteryparams*⟩ ')'

⟨*batteryparams*⟩ ::= ⟨*batteryparam*⟩
  | ⟨*batteryparams*⟩ ⟨*batteryparam*⟩

⟨*batteryparam*⟩ ::= 'Capacity:' ⟨*number*⟩
  | 'MaxLoad:' ⟨*number*⟩ !Reserved for later
  | 'InitialCharge:' ⟨*number*⟩
  | 'Type:' ⟨*batterytype*⟩

⟨*batterytype*⟩ ::= 'KiBaM'
  | 'Discrete'

⟨*time*⟩ ::= 'Start' '(' ⟨*number*⟩ ')'
  | 'Termination' '(' ⟨*number*⟩ ')'

$\langle$*identifier*$\rangle$ ::= $\langle$*String*$\rangle$

$\langle$*identifiers*$\rangle$ ::= $\langle$*identifier*$\rangle$
| $\langle$*identifiers*$\rangle$ ',' $\langle$*identifier*$\rangle$

$\langle$*Bool*$\rangle$ ::= 'true'
| 'false'

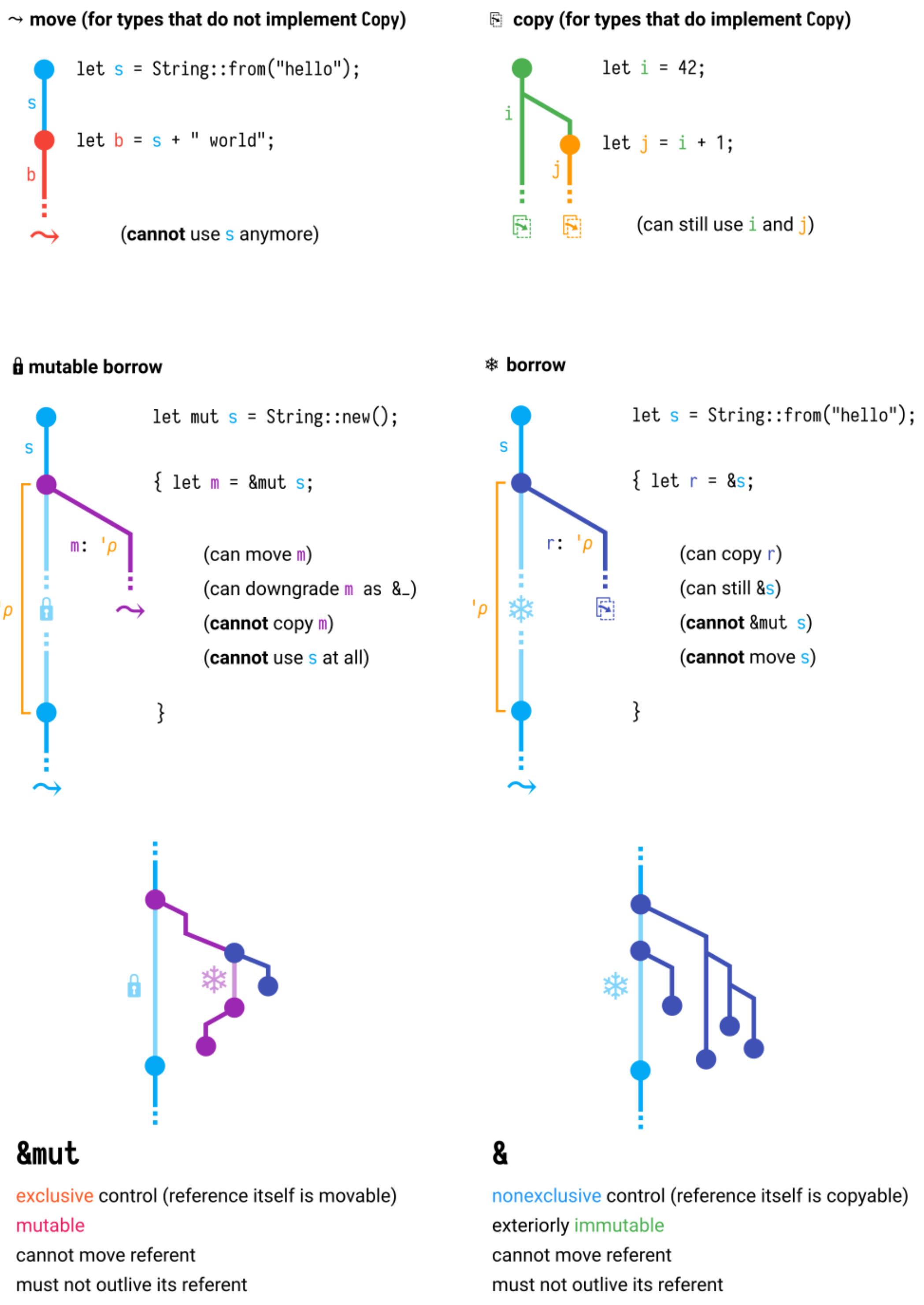
$\langle$*String*$\rangle$ ::= $\langle$*Letter*$\rangle$
| $\langle$*Letter*$\rangle$ $\langle$*String*$\rangle$

$\langle$*Letter*$\rangle$ ::= 'A'
| ...
| 'Z'

$\langle$*cost*$\rangle$ ::= $\langle$*number*$\rangle$

$\langle$*number*$\rangle$ ::= $\langle$*digit*$\rangle$
| $\langle$*digit*$\rangle$ $\langle$*number*$\rangle$

$\langle$*digit*$\rangle$ ::= '1'
| ...
| '0'


~ move (for types that do not implement Copy)
s
b
copy (for types that do implement Copy)
i
j
mutable borrow
s
m: 'ρ
'ρ
borrow
s
r: 'ρ
'ρ
&mut
&

```
let s = String::from("hello");

let b = s + " world";
```

(**cannot** use s anymore)

```
let i = 42;

let j = i + 1;
```

(can still use i and j)

```
let mut s = String::new();

{ let m = &mut s;
```

(can move m)

(can downgrade m as &_)

(**cannot** copy m)

(**cannot** use s at all)

```
}
```

```
let s = String::from("hello");

{ let r = &s;
```

(can copy r)

(can still &s)

(**cannot** &mut s)

(**cannot** move s)

```
}
```

exclusive control (reference itself is movable)
mutable
cannot move referent
must not outlive its referent

nonexclusive control (reference itself is copyable)
exteriorly immutable
cannot move referent
must not outlive its referent

Figure A.1: Visualization of the Rust move/copy/borrow semantics [6]

A.3 Rust Semantics

A.4 Branch And Bound

```
pub fn has_missed_opp(&self,system: &System,  stateP:State,
↪ stateC: State) -> i32 {
        let mut res = false;

        let mut counter: i32 = (system.nr_of_starts(stateP) as
↪ i32) - (system.nr_of_starts(stateC) as i32);
        if counter > 0 {
            for tasksC in system.get_running_tasks(stateC) {
                for tasksP in
↪ system.get_not_running_tasks(stateP) {
                    if tasksC.get_id() == tasksP.get_id() {
                        counter = counter - 1;
                    }
                }

            }
        }

        counter = counter * (-1);
        return counter;
    }
```

Listing 6: The experimental Branch and Bound algorithm used in the implementation

A.5 IsTaskAvail

```
// is_task_available
// Part 1
pub fn is_task_available(&self, state: State, status: TaskStatus)
↪ -> bool {
        let mut result = true;
        let task = self.get_task(status.get_id());

        // Check if task is already running, locked by another
↪ task, or has just been preempted
        if status.get_running() || status.get_locked() ||
↪ status.get_was_just_preempted() ||
↪ status.get_was_just_dropped() {
            result = false;
        }

        let deps_res = self.check_deps(status, state.clone());
        // Check if dependecies are met
        if result && !deps_res {
            result = false;
        }

        if result && !self.is_power_enough(state.clone(),status)
↪ {
            result = false;
        }

        // Check if the ones that needs to be locked aren't
↪ running
        for index in 0..state.task_status.len() {
            if let Some(stat) = state.task_status[index] {
                if stat.get_running(){
                    for id in self.get_locks(status.get_id()){
                        if stat.get_id() == id {
                            result = false;
                        }
                    }
                }
            }
        }
```

Listing 7: The function determining if a task is available for start - Part 1

```
// is_task_available
// Part 2
    // Check if componenents are availableLLEVEN CAN START
↪   ACCORDING TO TASK READ
    for rcom in self.required_components(status.get_id()) {
        for ucom in self.get_used_components(state) {
            if rcom == ucom {
                result = false;
            }
        }
    }

    let (in_opportunity, _, _) = self.task_in_opportunity(state,
↪   status.get_id(),false);

    if !in_opportunity {
        result = false;
    }
    return result;
}
```

Listing 8: The function determining if a task is available for start - Part 2

```
pub fn start_task(&mut self, id:id_t, system: &System)
   {

       for index in 0..self.task_status.len(){
           if let Some(mut status) = self.task_status[index]
           {
               if status.get_id() == id
               {
                   status.start(system.task_duration(id).0);
                   let lock_ids = system.get_locks(id);
                   for lock_id in lock_ids {
                       self.lock_task(lock_id);
                   }

                   self.task_status[index] = Some(status);

               }
           }
       }
   }
```

Listing 9: The function starting a task

```
fn is_task_preemtable(&self, state: State, status: TaskStatus) ->
↪  bool {
        let mut result: bool = false;

        if status.get_running() {
            let current_time = state.time;
            let (_, _, end) = self.task_in_opportunity(state,
↪  status.get_id(), true);

            let remaining_time = (end as i32) - (current_time as
↪  i32);

            // Check if this task has just been preempted, and if
↪  it has enough time to finish
            if let Some(task) = self.get_task(status.get_id()) {
                if !status.get_was_just_preempted()
                    && task.preemptable
                    && (self.task_remaining_time(status) as i32)
↪  < remaining_time as i32 {
                    result = true;
                }
            }
        }

        return result;
    }
```

Listing 10: The function that checks of a task can be preempted

```
fn is_task_droppable(&self, status: TaskStatus) -> bool {
        let mut result = false;
        let running = status.get_running();
        if let Some(task) = self.get_task(status.get_id()) {
            if task.droppable && running {
                result = true;
            }
        }
        return result;
    }
```

Listing 11: The function that drops a task

```
fn can_time_pass(&self, from_state:State) -> Option<State> {
        let mut new_state: State = from_state.clone();

        if( from_state.time >= self.end.time) {
            return None;
        }

        if let Some(newSoC) = self.get_future_soc(from_state) {
            if newSoC > 0 {
                new_state.time(self, newSoC as u64);
                let mut used_c =
 ↪  self.get_used_components(new_state);
                let mut ceck: id_t;
                for c in 0..used_c.len() {
                    ceck = used_c[c];
                    for m in 0..used_c.len() {
                        if( c == m ) {
                            continue;
                        }
                        if(used_c[c] == used_c[m]) {
                            return None;
                        }
                    }
                }

                for tasks in self.get_preempted_task(new_state) {
                    let (in_opportunity, _, _) =
 ↪  self.task_in_opportunity(new_state, tasks, true);
                    if !in_opportunity {
                        return None;
                    }
                }
            }
            return Some(new_state);
        }
        return None;
    }
```

Listing 12: The function determining if time can pass

```
pub fn start_task(&mut self, id:id_t, system: &System)
    {

        for index in 0..self.task_status.len(){
            if let Some(mut status) = self.task_status[index]
            {
                if status.get_id() == id
                {
                    status.start(system.task_duration(id).0);
                    let lock_ids = system.get_locks(id);
                    for lock_id in lock_ids {
                        self.lock_task(lock_id);
                    }

                    self.task_status[index] = Some(status);

                }
            }
        }
    }
```

Listing 13: The function that starts a task

```
pub fn preempt_task(&mut self, id: id_t)
   {
       for x in 0..8{
           if let Some(mut task) = self.task_status[x]
           {
               if task.get_id() == id
               {
                   task.preempt_task();
                   self.task_status[x] = Some(task);
               }
           }
       }
   }
```

Listing 14: The function that preempts a task

```
pub fn time(&mut self, system: &System, new_soc: u64) {

        self.time += 1;

        self.unlock_all();

        for index in 0..self.task_status.len() {
            if let Some(mut status) = self.task_status[index] {
                let running = status.get_running();
                if running {
                    status.time_transition(self.clone());

                    self.task_status[index] = Some(status);

                    if status.get_running()
                    {
                        let lock_ids =
 ↪  system.get_locks(status.get_id());
                        for lock_id in lock_ids {
                            self.lock_task(lock_id);
                        }
                    }

                }
            }
        }
        self.soc = new_soc;
    }
```

Listing 15: The function that applies the time transition

```rust
pub fn generate_states(&self, mut from_state: State) ->
 ↪  Vec<State> {
    let mut return_states: Vec<State> = Vec::new();

    // States where tasks are started
    if let Some(ids) = self.get_available_tasks(from_state) {
        for id in ids {
            let mut new_state = from_state;
            new_state.start_task(id,self);

            return_states.push(new_state);
        }
    }

    //where tasks are resumed
    for id in self.get_resumable_tasks(from_state) {
        let mut new_state = from_state.clone();
        new_state.start_task(id, self);
        return_states.push(new_state);
    }

    // Where tasks are dropped
    for id in self.get_droppable_tasks(from_state) {
        let mut new_state = from_state.clone();
        new_state.drop_task(id);
        return_states.push(new_state);
    }

    //where tasks are preempted
    for id in self.get_preemptable_tasks(from_state) {
        let mut new_state = from_state.clone();
        new_state.preempt_task(id);

        return_states.push(new_state);
    }

    if let Some(time_state) = self.can_time_pass(from_state) {
        return_states.push(time_state.clone());
    }

    return return_states;
}
```

Listing 16: The function that generates states based on a given state

```
if max_throughput{

        while id_pointer_list.len() != 0
        {

            if let Some(node_id) = id_pointer_list.pop_front()
            {
                let mut c_state =
↪   tree.get_mut(&node_id).unwrap().data().clone();


                for state_gen in system.generate_states(c_state)
                {

                    if best_state.is_state_better(state_gen)
                    {
                        best_state = state_gen.clone();

                    }
                }
                id_pointer_list.push_front(tree.insert(
↪   Node::new(best_state.clone()),UnderNode(&node_id)).unwrap());

                if best_state.time == system.end.time{
                    break;
                }
            }

        }

    }
```

Listing 17: Generates a trace where tasks are started when they can

```
// Branch and bound algorithm
// Part 1
while id_pointer_list_bb.len() != 0 {}
    let length = id_pointer_list_bb.len();
    let mut index = 0;
    let mut best_ub = 0;
    // Determines the one with the lowest bound
    for x in 0..length {
        let ubb =
↪   treeB.get_mut(&id_pointer_list_bb[x]).unwrap().data().u_b;
        if  ubb > best_ub {
            best_ub = ubb;
            index = x;
        }
    }

    //Checks for leaf nodes
    let l_length = leaf_node_list.len();
    for x in 0..l_length {
        if treeB.get(&leaf_node_list[x]).unwrap().data().u_b >
↪   best_ub {
            best_found = true;
            best_l_index = x;
        }
    }
    if best_found {
        break;
    }
```

Listing 18: The Branch and bound algorithm - Part 1

```
// Branch and bound algorithm
// Part 2
    if let Some(node) = id_pointer_list_bb.remove(index) {
        let this_ub = treeB.get_mut(&node).unwrap().data().u_b;
        let bbstate = treeB.get_mut(&node).unwrap().data().state;
        let generated_states =
 ↪  system.generate_states(treeB.get_mut(&node).unwrap().data().state);

        for states in generated_states {
            if states.time == system.end.time {
                        leaf_node_list.push_front(

 ↪  treeB.insert(Node::new(generate_node(states,this_ub)),

 ↪  UnderNode(&node)).unwrap());
                    }
            if bbstate.time < states.time {
                if  0 > bbstate.has_missed_opp(&system, bbstate,
 ↪  states) {
                    id_pointer_list_bb.push_back(treeB.insert(
                        Node::new(generate_node_l(states,(this_ub
 ↪  as i32),

 ↪  bbstate.has_missed_opp(&system, bbstate, states))),
                        UnderNode(&node)).unwrap());
                } else {
                    id_pointer_list_bb.push_back(treeB.insert(
                        Node::new(generate_node(states,this_ub)),
                        UnderNode(&node)).unwrap());
                }
            } else {
                id_pointer_list_bb.push_back(treeB.insert(
                    Node::new(generate_node(states,this_ub)),
                    UnderNode(&node)).unwrap());
            }
        }
    }
}
```

Listing 19: The Branch and bound algorithm - Part 2

A.6 Generated Pseudo Code

Listing A.1 : An example of generated code for a small schedule

```
Time 10: Starting task: Receive
Time 10: Starting task: Calculate
Time 15: Stopping task: Receive
Time 15: Stopping task: Calculate
Time 15: Starting task: Charge
Time 20: Stopping task: Charge
Time 30: Starting task: Track
Time 35: Stopping task: Track
```

A.7 Uppaal Trace

The trace derived from Uppaal in [2]

1. At time 10 in orbit one the task receive is prompted to start.
2. The task activates the action *slew*
3. The action slew activates the component *Gyroscope*
4. Also, at time 10 the task calculate is prompted to start
5. Task calculate prompts action calculate
6. Action calculate prompts Processor two to start, leaving processor 1 open for the Receive task.
7. At time 12 the gyroscope is released by task Receive
8. The Task receive then activates the components processor 1 and x-band.
9. At time 13 the action communicate is done, whereas the task Receive then starts the action slew again, resulting in the release of component x-band and processor 1, and then it acquires the gyroscope component.
10. At time 15 the action slew is done, releasing the component. Furthermore the task calculate is done resulting in the release of processor 2.
11. At time 15 the task charge is prompted resulting in the lock of all other tasks.
12. At time 17 the Task Charge is preempted.
13. At time 30 task track is prompted to begin hereby acquiring the component gyroscope

14. At time 32, the action slew is done, releasing the gyroscope, and the L-band component is then acquired for the tracking along with the acquiring of the processor 1.
15. At time 33 the action track is done releasing components processor 1 and l-band.
16. The gyroscope is activated for the action slew.
17. At time 35 it is done slewing resulting in the release of component gyroscope.
18. One orbit passes and the clock is reset

A.8 Gantt charts

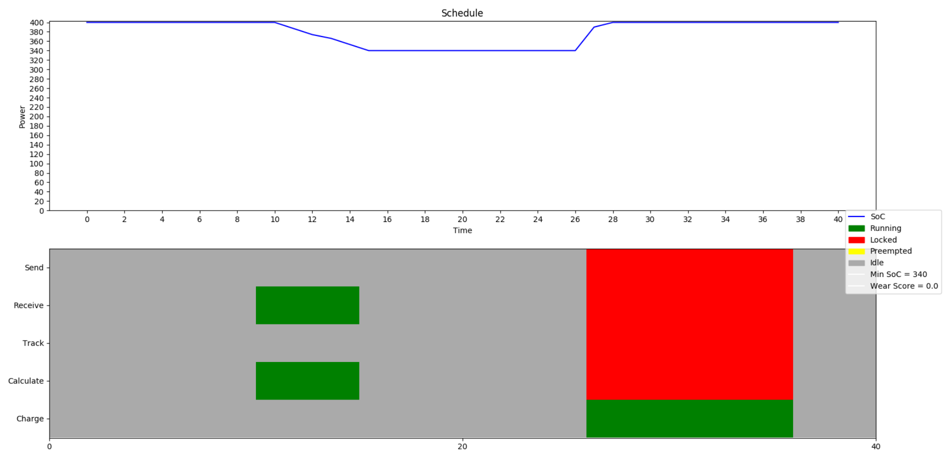


Figure A.2: The generated schedule for one orbit