March 1, 2019

MASTER THESIS

ENERGY MANAGEMENT IN SMART GRIDS USING TIMED AUTOMATA

Niels Geuze

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Design and Analysis of Communication Systems (DACS)
Internet Science and Technology (IST)

Exam committee:
Prof.Dr.Ir. B.R.H.M. Haverkort
Dr.Ir. M.E.T. Gerards
Dr.Ir. R. Langerak

UNIVERSITY OF TWENTE.

Abstract

Due to a shift in the way we produce and consume energy, the current method of balancing the electrical grid is no longer sufficient. Currently the grid is balanced using flexibility on the production side, which is provided by power plants. Due to a rise in Renewable Energy Sources (RES) the flexibility on the production side decreases. A solution is to make use of flexibility on the demand side for balancing the grid instead. This is challenging due to the amount of devices that need to be scheduled.

This research investigates whether Timed Automata (TAs) can describe and simulate Demand Side Management (DSM) scenarios. To answer this, a scheduling algorithm has been made. This algorithm uses Uppaal Stratego to create the schedules for each device. The results show that the algorithm produces schedules equivalent to the current state of the art in demand side management. While the current implementation of the algorithm takes a long time to compute the schedules, multiple suggestions are made to improve the performance of the algorithm.

Although the scheduling algorithm is slower than the current state of the art, it has shown potential. Using Uppaal Cora to schedule timeshiftable devices proved to be very efficient. The algorithm has also shown that a DSM scenario can be modelled using TAs, which allows for statistical analysis to be performed on the model.

Keywords

Demand Side Management; ALPG; DEMKit; Uppaal; Cora; SMC; Stratego; Online; Compositional

Acknowledgments

This thesis was conducted at the Design and Analysis of Communication Systems (DACS) group and in cooperation with the Computer Architecture for Embedded Systems (CAES) group, specifically the people working with energy related subjects. I would like to thank both groups for being welcoming and helping me whenever needed.

I would specifically like to thank my supervisors Boudewijn Haverkort, Marco Gerards and Rom Langerak. Boudewijn was my first supervisor and helped me immensely during my master, not only during my final project. He helped me find an interesting graduation project and was always enthusiastic about my work, while remaining critical enough to give me valuable feedback. I want to wish him all the best with his new job at the Tilburg University.

Marco taught me most of what I currently know about energy management and was always available when I had questions regarding energy management. He also introduced me to the ENERGY group and pointed out flaws in my writing of papers and thesis, which I have since tried to improve.

Rom was originally not listed as my supervisor, however, during the project I got stuck with an Uppaal problem and he came to the rescue. He not only pointed me onto the right track, but became my third supervisor in the process.

From the CAES group I would like to specifically thank Gerwin Hoogsteen for teaching me how to use DEMKit and helping me fix any bugs I found along the way. Without his help the comparison between DEMKit and this research would have looked very differently.

I would like to thank my parents, brother, family, friends and colleagues for supporting me throughout my study. In particular, I want to thank two friends who read through my thesis and gave me feedback.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ALPG Artificial Load Profile Generator.

ANOVA Analysis of Variance.

DEMKit Decentralized Energy Management Kit.

DER Distributed Energy Resource.

DSM Demand Side Management.

DVFS Dynamic Voltage and Frequency Scaling.

EV Electrical Vehicle.

KiBaM Kinetic Battery Model.

LPTA Linearly Priced Timed Automaton.

PHEV Plug-in Hybrid Electrical Vehicle.

PSM Production Side Management.

PV Photovoltaic.

RES Renewable Energy Source.

SMC Statistical Model Checker.

SoC State of Charge.

TA Timed Automaton.

Chapter 1

Introduction

At the time of writing this thesis there is an ongoing change in the way we produce and consume energy. This change can be characterized by three developments.

The first development is that an increasing percentage of the energy production comes from Renewable Energy Sources (RES), such as Photovoltaic (PV) systems and wind turbines. This development is, amongst others, prompted by an increasing concern for the environment. The increase in RES results in less flexibility in the times at which energy is produced, due to most RES being weather dependent. The dependence on weather also means that it is challenging to accurately predict the energy production of RES.

The second development is the rise in popularity of Distributed Energy Resources (DERs). DERs are devices that have a controllable load or produce energy. There are many examples of DERs, however, a commonly known example is a smart dishwasher. Smart dishwashers can be scheduled to have completed their program before a selected moment in time. This allows the dishwasher some flexibility in when it starts its program. There can be several factors that determine when the dishwasher starts. These factors are device dependent; one factor that can be used is the price of energy. Because the load of the dishwasher can be shifted over time, by starting its program at a later moment, it is considered a DER. Another example of a DER is a PV panel, which produces energy and is thus considered a DER. These DERs provide flexibility in when energy is consumed. This flexibility can be utilised to save costs and to more efficiently use the energy produced by RES.

The third and last development is called electrification. Electrification is defined by the Oxford Learner's Dictionary as "the process of changing something so that it works by electricity" [1]. The *something* in electrification is generally an object which consumes fossil fuel. There are many

examples of electrification, most notable are Electrical Vehicles (EVs), electrical stoves and electrical heaters. These examples traditionally consumed gas, whereas nowadays they can consume electricity. This results in an increasing electricity demand.

The aforementioned developments have a large impact on the electrical grid. The electrical grid needs to ensure that enough energy is available to meet the demand of the consumers. Traditionally, ensuring that enough energy is available is handled by using the flexibility on the production side. The decrease in flexibility on the production side, caused by an increase in RES, reduces the viability of matching the demand by changing the amount of energy that is produced. The unpredictability of RES, due to their dependence on weather conditions, exacerbates this. Combined with the increase in energy consumption due to electrification, this results in an increased probability of overloading the grid. Due to their reliance on the weather, the RES can also synchronize their energy production. This can give sudden peaks in the energy production, which also has the potential to overload the electrical grid. To better match the energy production to the energy consumption, the added flexibility provided by DERs can be used. This results in the electrical grid making a shift from Production Side Management (PSM) to Demand Side Management (DSM). This shift has its upsides and downsides. The downside is that PSM is relatively easy to manage due to the limited number of points of flexibility. When using DSM, each device with a controllable load becomes a point of flexibility. Due to the existence of many controllable loads, managing all of them is challenging. The upside of using DSM is that it has the potential to save $1522 billion over the next 20 years in the US alone [2].

Making use of DSM is only possible in a smart grid [3]. Smart grids provide energy providers with the necessary means to monitor the state of the electrical grid. Energy providers can use the smart grid to steer the DERs, thereby reducing the strain on the electrical grid. The strain on the grid can only be reduced when the DERs have knowledge about the optimal time to consume energy. This information is usually passed to the DERs by using steering signals. There are currently two kinds of steering signals being used, namely, price signals and profile signals [4]. Calculating the optimal steering signals is in general NP-hard for both price and profile signals [4]. Generating price signals is often accomplished using game theory [5][6][7]. Profile signals are generated using profile steering [4].

This research investigates whether Timed Automata (TAs) can be applied to the DSM scheduling problem. To accomplish this, a model is developed that adheres to the constraints set by a scenario. Based on this model, a schedule is calculated for each DER. These schedules aim to reduce the strain on the grid as much as possible. We have used the tool Uppaal [8] for making this model and generating the schedules. Three different versions of Uppaal have been used throughout this research, namely, Uppaal Cora [9], Uppaal SMC [10] and Uppaal Stratego [11]. Uppaal Stratego has been integrated into an online compositional algorithm, which works similar to profile steering. This algorithm proved that it is possible to use TAs to generate good DSM schedules.

1.1 Problem statement

PSM has been the standard when it comes to energy management. This has seen great success due to the predictability of the aggregated electricity consumption of large groups [2]. When the demand can be accurately predicted and the production is flexible, the electricity production can be matched to the demand. The rise in RES causes problems since they have little flexibility in when the electricity is produced. RES also create challenges due to being unpredictable because of the dependence on weather conditions. When the required amount of electricity is difficult to predict, the energy providers have to compensate for the uncertainty. Compensating is either done by buying excessive electricity or by buying additional electricity close to the deadline. Either method incurs additional costs, either due to using less than the purchased amount or due to buying expensive last minute energy.

To deal with this we can add a large number of batteries to the electrical grid. These batteries can store excessive energy and discharge the energy when needed. The downside is that batteries are expensive, require maintenance and do not have a perfect conversion rate. Using the flexibility provided on the demand side by DERs is a better solution [12]. This flexibility can be used to better consume the electricity produced by RES, make the electricity demand more predictable and reduce the strain on the grid.

To successfully lower the strain on the grid, all DERs need to work towards the same goal. There are multiple approaches that can achieve this goal, however, the focus of this research is on an approach that aims to create a schedule for each device. Creating optimal schedules

is challenging due to the problem in general being NP-hard. There currently exists an implementation of profile steering that creates these schedules inside of the Decentralized Energy Management Kit (DEMKit) [13]. This makes use of a simulation environment designed specifically for simulating decentralized energy management algorithms. Instead of using such a custom simulator, the goal of this research is the use the generic modelling technique of TAs. TAs have proven useful for scheduling problems in the past, e.g. [14][15][16], although never at the scale of DSM.

1.2 Research goals

The main goal of this research is to discover whether TAs have good potential for modelling a DSM scenario. This is investigated by attempting to create a model that contains several types of DER and can create an efficient schedule for these devices. Specifically, the tool Uppaal is used to schedule the devices in a scenario created by the Artificial Load Profile Generator (ALPG). The resulting schedule is compared to the schedule created by DEMKit. Our main research question is: *Can TAs correctly and efficiently model and simulate a residential DSM scenario?* To answer this question, the following research questions are answered.

1. Are timed automata suitable for solving the demand side management scheduling problem?
   (a) How do the profiles compare to profiles created by existing solutions?
   (b) Can the models be adapted to new situations?
2. How do the created timed automata scale?
   (a) What amount of data can be handled, expressed in the amount of days that need to be scheduled, devices and households?
   (b) Can performance be increased by sacrificing accuracy?
3. What are strengths and weaknesses of using timed automata for solving the scheduling problem?

1.3 Thesis outline

After this chapter, Chapter 2 gives background information about DSM and TAs, specifically the TAs implementation of Uppaal. This includes the

difference between the three Uppaal versions that are used throughout this research. It is highly recommended for anyone who is unfamiliar with either of these concepts to read these sections. There are references to the background throughout the thesis, however, it is assumed that the most concepts are understood.

Chapter 3 discusses the works of several other researchers and how their work relates to this research. The related works are split into two sections, namely, the state of the art and the existing Uppaal scheduling algorithms. The state of the art focusses on profile steering and a tool that implements, amongst others, profile steering. The second section discusses several papers that use Uppaal for multiple kinds of scheduling problems. It is discussed what the differences between these papers are and how each paper relates to this research.

Chapter 4 discusses several specific Uppaal details that go beyond the background information. These details are used for developing the model and are referenced in Chapter 5 and 6 whenever they are of importance.

Our new methodology is discussed in Chapter 5. This is the core chapter of the thesis. It explains the data that is used, how the data is used by Uppaal and the model that is used to create schedules. The chapter also details how the different versions of Uppaal are used to create schedules and how the developed model is transformed into an online compositional algorithm.

Chapter 6 describes how the tests are executed. This chapter is split into two parts, namely, the comparing the different versions of Uppaal and the comparison between the online compositional algorithm and DEMKit.

The results of the aforementioned tests are given in Chapter 7. This chapter also discusses the results and answers the research questions related to the performance of the new approach.

All the research questions are answered in Chapter 8 and the limitations and some ideas for future work are discussed in Chapter 9.

Chapter 2

Background

This chapter gives background information that is needed to understand the remaining part of the thesis. First, Section 2.1 describes the studied Demand Side Management (DSM) problem and details the device types that are used throughout this thesis. Then, Section 2.2 gives a brief introduction to finite automata. Section 2.3 expands on the concept of finite automata by introducing Timed Automata (TAs). The implementation of TAs that is used throughout this research is described in Sections 2.4, 2.5, 2.6 and 2.7. Then, three sections explain concepts of algorithms that are used within this research: Section 2.8 explains valley-filling, after which Section 2.9 details the difference between offline and online algorithms. Finally, Section 2.10 introduces compositional algorithms.

2.1 Demand Side Management

DSM is the concept of using flexibility on the demand side to balance the electrical grid, rather than using flexibility on the production side. Due to the increase in Renewable Energy Sources (RES), which provide no flexibility, the need for DSM is increasing. This lack of flexibility prevents the grid to be balanced on the production side, which is called Production Side Management (PSM). DSM is computationally challenging due to the amount of controllable devices that exist on the demand side. Because scheduling the demand side devices is generally an NP-hard problem [17], algorithms for creating the schedules do not scale well. Therefore, it is computationally infeasible to have an algorithm that optimally schedules all demand side devices for large neighbourhoods.

The goal of DSM is not only to balance the grid, but also to lower the strain on the electrical grid. The strain on the grid is expressed as the square of the combined energy consumption, often computed per neighbourhood. High strain on the grid reduces the efficiency of the

electricity transport and can damage the grid, therefore, it is essential to keep the strain as low as possible.

There are several different types of devices that exist on the demand side of the electrical grid. During this research we considered the uncontrollable devices, timeshiftables, Electrical Vehicles (EVs) and batteries. Each of these device types is detailed below.

2.1.1 Uncontrollable devices

Devices that have no flexibility in when they consume or produce energy are called uncontrollable devices [13]. This includes many devices, such as stoves, TVs and many more everyday appliances. Because these devices lack flexibility, their combined cost is referred to as the static or uncontrollable costs throughout this thesis.

Hoogsteen [13] also details curtailable devices. Curtailable devices are RES that are able to lower their energy output when needed. We did not consider the energy output from RES curtailable, therefore, the RES are also considered uncontrollable devices.

2.1.2 TimeShiftables

Timeshiftables are devices with an uncontrollable consumption profile, but can be started at any time during a prescribed time period [13]. Examples of this are a dishwasher or washing machine. Modern versions of these devices often include technology that allows the user to instruct the device to be finished before a selected moment in time. This allows the device flexibility in when it starts its program. Because the device can only be shifted in time, it is called a timeshiftable. There also exist timeshiftable devices that can be interrupted during their program. However, these are not considered in this research.

2.1.3 Electrical Vehicles

There exist two types of EVs, namely EVs and Plug-in Hybrid Electrical Vehicles (PHEVs) [18]. The difference is that PHEVs can use both electricity and gasoline. For DSM both of these are considered equal. The EVs have an interval in which the battery must at least reach a given charge level. The amount of electricity that is consumed during a charging interval is controllable, however, there is a limit to the charging rate. There are EVs that can discharge electricity to the grid as well, however, these are not considered during this research.

2.1.4 Batteries

Batteries, or buffer devices as they are called in [13], are devices that can charge electricity from the grid, store the energy and discharge it to the electrical grid at a later point in time. The only restrictions batteries have, are their capacity and how rapidly they can charge and discharge. Batteries are not perfect and energy is lost when storing it in the battery. The amount of energy that is lost depends on the type of battery and can be challenging to calculate, especially when taking advanced battery models into account, such as the Kinetic Battery Model (KiBaM) [19].

2.2 Finite automata

Finite automata, or finite state-machines, are mathematical models that can be used to describe discrete-state behaviour [20]. An automaton consists of a set of states and edges. The automaton can only ever be in one state at a time. Transitioning to other states is accomplished by using edges that correspond to an action. These actions are given as input to the model. An automaton is called finite when the amount of states is finite. An example of a finite automaton is given in Figure 2.1. The example models the state of a door, which can either be closed or open. When the door is closed and it receives the input *Open*, the door will transition to the state *Open*. When the door is open and the model receives the *Close* input, the state changes to *Closed*.

This is one of the most basic examples of a finite automaton and is only used to illustrate its functionality. Finite automata are often used to model the state of a computer program, in order to validate that the program never crashes and adheres to the intended behaviour. The reverse is also possible, namely, generating a model from a program. In that case the model is a human readable representation that a person can use to validate the behaviour of the program. There are many more use cases, due to it being a generic modelling language.

Multiple automata together create a network, which can be combined into a single automaton. Using multiple automata keeps the model readable and prevents mistakes. When talking about a network of automata, the states are called locations and the edges are called transitions. This terminology is used throughout this thesis to distinguish between a network of automata and the automaton that is created based on the network.

To express different systems, several extensions to finite automata

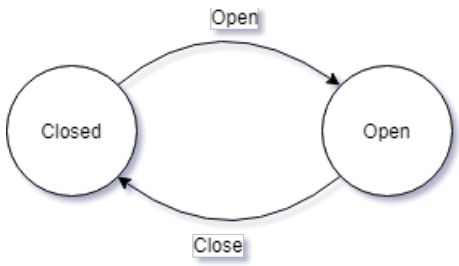

Figure 2.1: Example door modelled as a finite automaton.

have been developed. The next section explains one such extension, namely, timed automata.

2.3 Timed automata

TAs are an extension of finite automata [21]. The extension consists of adding a finite set of real valued clock variables. All clocks increase at the same speed. These clocks can be used to enable or disable transitions. Disabling a transitions is done by comparing a clock to an integer. Clocks can also be reset to time how long certain actions take. An example of a TA is given in Figure 2.2. This example is an extension of the example for a finite automaton. The example models a door that automatically locks itself five minutes after closing. This means that, when a door is closed by the user, it can be opened without needing a key for five minutes. When the user wants to open the door after more than five minutes have passed, they are required to use a key to unlock the door before opened it. When opening the door, it is checked whether the clock is smaller than five, representing five minutes in this example. If the clock is smaller than five the door can be opened, otherwise the door needs to be unlocked. Unlocking the door resets the clock, indicating that it is no longer locked. When closing the door, the clock is reset, ensuring that the door will be locked again after five minutes.

This example includes input actions. However, TAs are generally used for checking reachability of states. In order to check reachability, the inputs are omitted, such that the only restrictions on the automaton are the clocks. There are several mathematical methods that can be used to represent such a model efficiently and verify reachability properties based on it.

This type of model is often used when timing is essential. By using mathematical techniques, it is possible to efficiently compute statistics about the model. It is also possible to extend such a model with additional

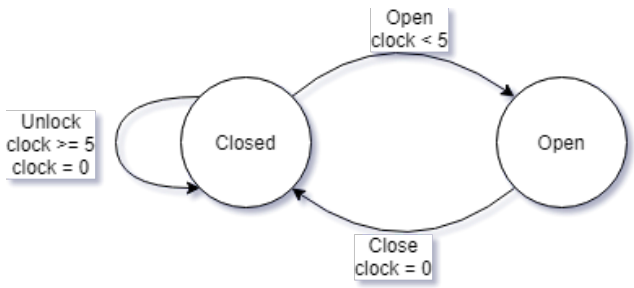

Figure 2.2: Example door modelled as a timed automaton.

data types. However, this is implementation specific, as will be shown in the next section about one such implementation of TAs.

2.4 Uppaal

Uppaal [8] is a tool for modelling, validation and verification of real-time systems. The models are networks of TA that have been extended with several data types. It allows the user to verify whether reachability properties hold and to retrieve an example that proves a property holds. Uppaal is a complex tool and it is not feasible to discuss every detail in this thesis. Instead a brief explanation about the basics is given in this section, for a more detailed explanation, please refer to [22][23]. Section 2.5 introduces the different components of Uppaal models, after which Section 2.6 explains the different components of Uppaal. Finally, Section 2.7 discusses the differences between the multiple versions of Uppaal.

2.5 Uppaal model components

An Uppaal model is combination of code, TAs and queries. Code is put into declarations, of which there are global and local variants. Templates are similar to classes inside of object-oriented programming languages, such as Java and Python. These templates consist of an automaton, a set of local declarations and a set of parameters that are needed to create an instance of the template. The templates are instantiated and combined into a model in the system definition. Queries can be used to verify, amongst others, reachability properties of the system. All of these concepts are explained in this section.

2.5.1 Declarations

Declarations contain the code of the Uppaal model. The declaration language is a custom subset of the programming language *C*. Similar to an object-oriented programming language there are global declarations, which can be accessed from within all template instances, and local declarations, which are unique to each instance. Declarations consist of a set of variable definitions and assignments, type definitions and function definitions, each of which are discussed below.

Variables definitions

Variable definitions and assignments work similar to *C* and many other programming languages. An example of variable definitions and assignments can be found in Listing 2.1. There are certain definitions that are unlike most programming languages. The following keywords do not exist in *C* but do in Uppaal.

- *int[min, max]*: Defines an integer that can take all values from *min* until *max*, including *min* and *max*.
- *meta*: Used to indicate a variable has no impact on the state space.
- *clock*: Defines a clock, which is the core component of TAs.
- *hybrid*: Indicates that a clock is not allowed to be used in comparisons. Therefore, the clock does not change the behaviour of the system and does not increase the state space.
- *chan*: Defines a channel that is used to synchronize between template instances.
- *broadcast*: Makes a channel synchronize all instances that are listening and makes the emitting transition non-blocking.

There exist more keywords that *C* does not have, however, these are the most important ones.

```
const int array[3] = {1,2,3};
int variable = Function(arguments);
variable = 0;
int[min, max] customRangeInteger;
meta int metaVariable;
hybrid clock time;
broadcast chan channel;
```

Listing 2.1: Uppaal variable example

Type definitions

It is possible to define custom types in Uppaal. There are two kinds of type definition. An example of each type is given in Listing 2.2. The first kind of type definition is by using an existing type and declaring it anew. This is useful for integers and allows, amongst others, 32-bit integers to be declared, whereas the native *int* type defaults to 16-bit.

The second kind of type definition is for creating structures. Structures comprise several data types and allow each attribute to be accessed by using their name. The example given in Listing 2.2 creates an object named *item*, which has the attributes *price* and *weight*. In this case both price and weight are integers, however, this does not need to be the case. Instances of the structure are declared in a similar fashion to an array. After instantiation it is possible to access the *weight* attribute by using its name. This is a convenient method of keeping the data readable for humans and allows data with different types to be stored together. It is also possible to nest structures, which is used extensively throughout this research.

```
typedef int[0,1] boolean;
typedef struct {
  int price;
  int weight;
} item;
item newItem = {5, 3};
bool trueStatement = newItem.weight == 3;
```

Listing 2.2: Uppaal variable example

Function definitions

Functions are defined similar to *C*. An example function definition is given in Listing 2.3. It is also possible to pass parameters as references, such that the input can be directly manipulated. The only restriction is that recursion within functions is not allowed, although we are unaware of why this restriction has been implemented.

```
bool exampleFunction(int example) {
  return example == 5;
}
```

Listing 2.3: Uppaal variable example

2.5.2 Templates

Templates are the main components of Uppaal models. A template consists of three parts, namely, an automaton, a declaration and parameters. Each part is described below.

Automaton

The automaton is the core of the template. Like a regular automaton it consists of a set of locations and transitions. An example automaton is given in Figure 2.3. Each automaton needs exactly one location that is indicated as the *initial* location, meaning it is the starting location of the automaton. Each location has three attributes, namely, the *invariant*, the *rate of exponential* and whether the location is *urgent* or *committed*. Each of these attributes is described below. Transitions have five attributes, namely, *select*, *guard*, *sync*, *update* and *probability*, these are also described below.

Invariant The invariant is a statement that must always hold while the automaton is at the location. An invariant consists of conditions that can be used to ensure that a location is left in a timely manner. The invariant can also contain special conditions, these set the rate at which a clock increases. In the example automaton, given in Figure 2.3, the initial location of the first process, named *InitialActive*, has the invariant $time <= 5$. This means a transition must be taken before the clock variable *time* reaches five. When no transition can be taken when *time* reaches five, and the automaton in still inside the *InitialActive* location, a time-lock occurs. A time-lock resembles a deadlock for clocks. It occurs when a state is reached in which any action prevents the constraints of the model from being satisfied. Letting time pass is also considered an action. Therefore, a time-lock can only occur in locations that have an invariant, otherwise the model can always choose to let time pass.

Rate of exponential The rate of exponential changes the distribution that is used to decide at what time a transition is taken. This can be useful when simulating waiting times, however, during this research we do not use this functionality.

Select The select attribute is used to randomly choose a value from a range of values when taking a transition. The result is as if there are multiple transitions, each using one of the values from the desired range.

In the example automaton, Figure 2.3, an example can be seen when transitioning from the *InitialActive* location to the *Committed* location. The transition uses the variable *number* as an integer between zero and five.

Guard Guards are used to restrict when a transition is allowed to be taken. When the condition of the guard holds, the transition is allowed to be taken. In the example automaton, an example guard can be seen when transitioning from the *InitialActive* location to the *Committed* location. The guard $time == 5$ checks whether the clock variable *time* is equal to five. The only restriction to guards is that it is not allowed to combine multiple clock comparisons using an *or*, such as $clock1 == 5||clock1 == 3$. To achieve this behaviour, multiple transitions need to be used.

Sync The sync attribute is used to synchronize with channels. Channels being one of the data types that can be defined. Using the $channelName!$ expression emits a signal on the channel, whereas $channelName?$ synchronizes on the channel. Synchronizing on a channel means that a transition is taken when a signal is emitted on said channel, assuming the guard on the synchronizing transition holds. An example of this can be seen in the example automaton, Figure 2.3. When transitioning from the *InitialActive* location to the *Committed* location, a signal is emitted on the channel that is conveniently named *channel*. The second automaton in the example synchronizes when this signal is emitted and transitions from the *InitialWaiting* location to the *Urgent* location.

Update The update attribute is used to run code when taking a transition. This piece of code can contain any combination of assignments and function calls. There are multiple examples in the example automaton. All of the blue texts that assign the *cost* variable are update statements.

Probability The probability attribute is used when there are two transitions that can be taken and the likelihood of each transition being taken is not equal. This is not used during this research and is therefore not further discussed.

Urgent and committed locations Locations can be indicated as being either *urgent* or *committed*. Locations that are indicated with either of these keywords are required to take a transition before any time can pass. Committed locations do not create a new state in the state space, which

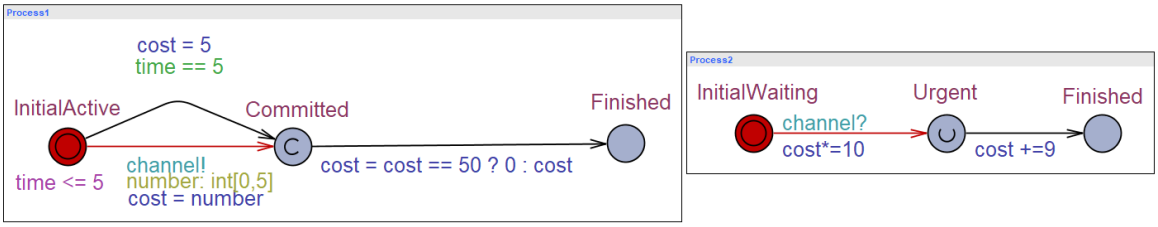

Figure 2.3: Example Automaton

can be very useful when trying to reduce the state space. Urgent locations do create a new state in the state space. Therefore, urgent locations cannot transition while there are still automata in a committed location, otherwise a committed location would end up in the state space. This behaviour can be seen in the example automaton. When a synchronization occurs, the automata end up in the *Committed* and *Urgent* locations. In this case, the *Committed* location must first transition to the *Finished* location, before the *Urgent* location can transition.

Local declaration

Each template has its own local declaration. Everything inside of this declaration is bound to the instance of the template. This is similar to local variables in object-oriented programming languages. There are no additional restrictions compared to the global declarations. Local declarations can have the same name as existing global declarations. Accessing an attribute or function by name gives priority to local declarations over global declarations.

Parameters and instantiation

Templates also have a set of parameters. These parameters are filled during the instantiation of a template. This is similar to how constructors in object-oriented programming languages take arguments to create an instance of a class. The parameters can also be given as references, similar to how functions can take arguments as references. Parameters are accessible to the template, similar to how local declarations are accessible.

2.5.3 System

The Uppaal system definition is what combines multiple templates into a single automaton. The system definition allows instances of the templates to be created, which can then be added to the system definition. An

example system definition is given in Listing 2.4. In this example two instances are created from the same template. The first instance has passed *one* as the first argument, whereas the second instance has passed *two* as the first argument. Both of these instances are then added to the system definition.

It is also possible to leave some of the parameters of a template empty. This results in one instance being created for every value the empty parameters can take. Combined with the custom types that can be defined this is extremely useful. By defining a new type that can take the values in the range of one until five and using it as a parameter, it is possible to create five instances of the same template. This means it is not needed to define all of the instances and the parameters they require to the template.

```
Instance1 = Template(1);
Instance2 = Template(2);
system Instance1, Instance2;
```

Listing 2.4: Uppaal variable example

2.5.4 Queries

Queries are used, amongst others, for verifying properties of the system. There are many different types of queries and it is outside the scope of this thesis to discuss everything that is possible. The queries that are possible also depend on the type of model. With a classical TAs model the queries verify reachability of states of the system. Evaluating such reachability queries allows a trace to be obtained, which is an example path through the state space that proves the state is reachable. Using the Statistical Model Checker (SMC) disables standard reachability queries, instead it enables, amongst others, queries that test the likelihood of a state being reached. The SMC queries also allow for running simulations and monitoring how the values of expressions change over the duration of the simulation.

2.6 Uppaal Components

The Uppaal program consists of three components, namely, the GUI, the verifyta and the server, discussed in Sections 2.6.1, 2.6.2 and 2.6.3 respectively.

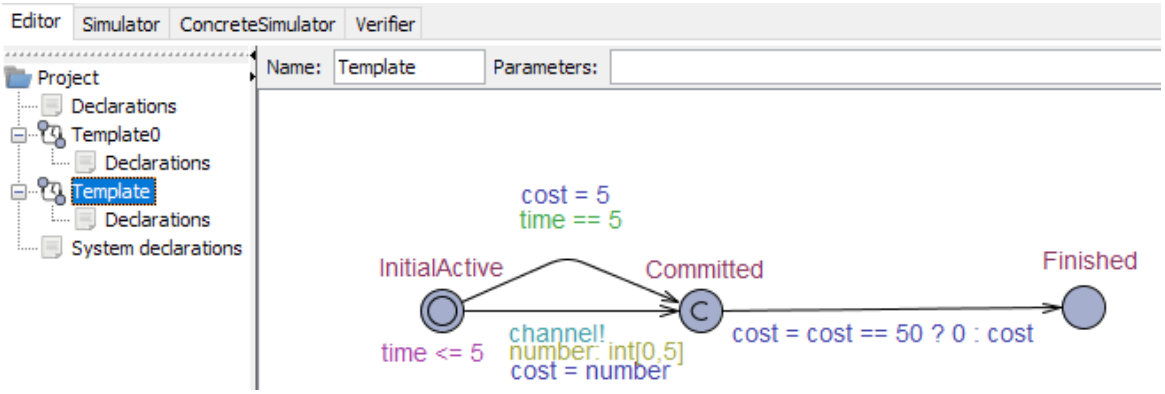

Figure 2.4: Uppaal Editor

2.6.1 GUI

The GUI is the visual component of Uppaal. It consists of three parts, namely, an editor a simulator and a verifier. Each part is described below.

Editor

Figure 2.4 shows the editor. The editor is used to create the model. On the left it shows the project. The project contains a *Declarations* file, *System declarations* file and a set of templates, each of which has a *Declarations* file as well. On the right the model can be edited. The right can either be the automaton view, as displayed in the figure, or the code view for declarations. Above the automaton view the name of the template and the parameters can be set.

Simulator

The simulator is displayed in Figure 2.5. The simulator is used to manually make the decisions of a simulation or to investigate a trace provided by Uppaal. On the top left the choices for the next transition are listed. Below that is the simulation trace, which contains all of the transitions that are taken. The simulation trace shows the state of the system after each transition. The list of variables and their values is shown in the middle. The top right shows the current state of the automaton for each instance. The bottom right shows the simulation trace in a more readable format.

The simulator is mainly used as a debugging tool that can be used to discover flaws in the model. When not using the SMC, it can also display the optimal trace that was found during verification. This can be used to analyse the choices that were made to minimize the cost.

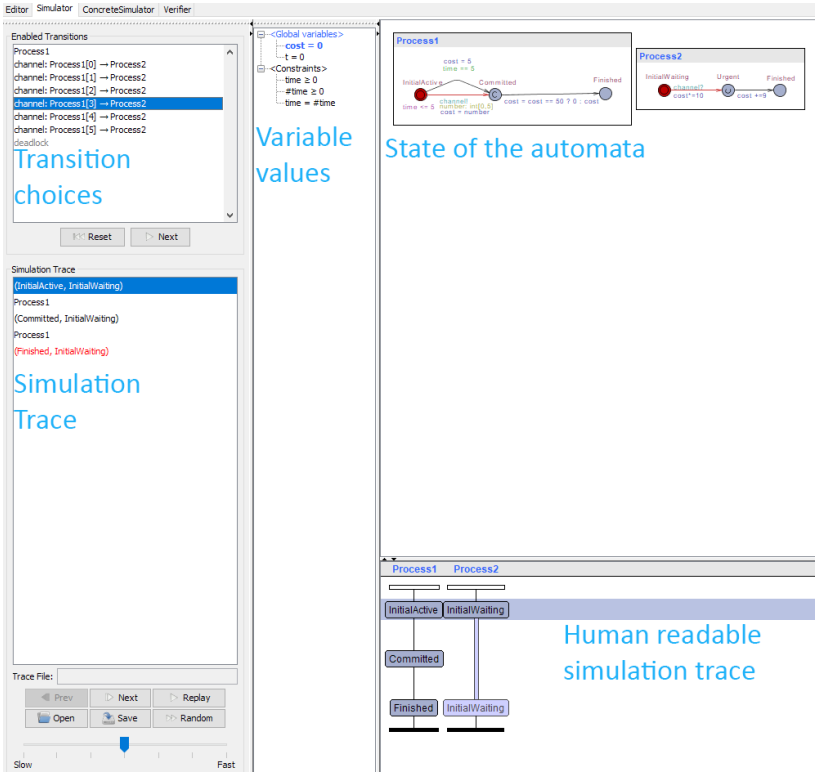

Figure 2.5: Uppaal Simulator

Verifier

The verifier is used to execute a query and display the result. The main part of this interface is a text editor for the queries. When using the SMC, it is possible to view the results as a plot by right clicking a query that has been verified. However, there are multiple kinds of plots that are available. Therefore, we did not include any figures for this. For more information about what can be plotted have a look at [23].

2.6.2 Verifyta

The verifyta program is the command line interface that is used for validation and verification of the model. It takes a model and a set of queries that are to be verified and returns the result of each query. It has many parameters. The exact parameters depend on the version of Uppaal that is being used. The parameters can be used to change the output or the fashion in which the verification is handled. Several of these parameters are discussed in Chapter 4.

2.6.3 Server

The server is used by the GUI to execute all of its validation and verification work. There is little information publicly available about how the server works. However, it fulfils the functionality of the verifyta in a way such that the GUI can properly communicate with it.

2.7 Uppaal Versions

2.7.1 Uppaal Cora

Uppaal Cora [9] extends the model specification with a *cost* variable. The *cost* variable behaves mostly like an integer, with the restriction that it can only increase. The *cost* variable can also increase over time by adding a special invariant to a location. When incrementing the *cost* using an invariant, the invariant determines the rate at which the *cost* increases. What makes the Cora version of Uppaal special is that it includes an algorithm to find the path that minimizes the *cost* variable. The regular version of Uppaal can only find the shortest route required to reach a condition. Having the *cost* variable changes the model from a TA into a Linearly Priced Timed Automaton (LPTA), the definition of which is given below. The minimization of the *cost* variable is done by exploring all paths, the result being the path with the lowest *cost* [24]. Depending on the number of possible paths, this can take a very long time. Especially for problems with an exponentially sized state space this gives problems with respect to the time and memory required to calculate the optimal solution.

Linearly Priced Timed Automata

LPTAs are an extension of TAs [25]. The difference is that a LPTA has a price associated with each state. This price increases linearly based on the state the automaton is in. The price is not allowed to be used in any conditions, otherwise the price would have an effect on the model, thereby exploding the state space. Because of these restrictions, all paths leading to a state that are more expensive then another path leading to the same state, are discarded [24]. This results in the path to each state being the lowest possible. Hence, the path to the end state with the lowest cost must be the optimal path.

2.7.2 Uppaal SMC

Uppaal SMC [10][26] no longer verifies a query, but rather gives a statistical estimation of the validity of the query. This is accomplished by monitoring several runs of the system and using the results to give the confidence interval of whether the query property holds. The models themselves are extensions of the regular models, although some minor restrictions have been added. The major difference is that the user can now define probabilities for transitions to be taken. These probabilities matter due to the entire state space no longer being calculated. Instead, multiple simulations are run, in which the probabilities determine the odds of a path being taken.

It is also possible to run simulations and get the state information for each simulation. This can be used to replace traces, which are not generated when using the SMC. Although a trace, as generated when verifying classic reachability queries, is more detailed and contains the exact transitions that occur, a simulation often reveals most of the transitions that are taken. The downside of using simulations to generate traces lies with committed and urgent locations. Because there can only be one data point at each time interval of the simulation, the intermediate values when transitioning through a committed or urgent state are not present in the output. The time interval of the simulation being determined by a setting within Uppaal, which is discussed in Section 4.4. Committed and urgent locations transition without any time passing. Therefore, it is not always possible to obtain the complete trace, which would include the transitions involving committed or urgent locations.

2.7.3 Uppaal Stratego

Uppaal Stratego [11][27] is an extension of Uppaal SMC that adds an algorithm that uses simulations to determine a strategy that tries to minimize or maximize the value of an expression [28]. To find a strategy, some of the transitions can be indicated as controllable. Controllable means that the optimization algorithm is allowed to choose when it comes to those transitions. All the attributes associated with the transitions, such as guards and assignments, are still in place and influence the model. Only when a choice arises the strategy is allowed to decide which transition to take. The strategy is created using a reinforcement learning algorithm, which is explained below. Due to the usage of a reinforcement learning algorithm, the strategy is often not optimal. The more simulations the

algorithm is allowed, the better the strategy becomes. This is a significant difference with Uppaal Cora, because Cora has to find the optimal path. Because the Stratego cost minimization algorithm does not need to explore every possibility, it is a good alternative for Cora when it comes to finding a schedule for large scheduling problems. However, because Stratego runs complete simulations and can account for randomness, it is quite a bit slower than Uppaal Cora when attempting to find the optimal solution.

It is also possible to create strategies that ensure a criterion always holds, such as that a variable never reaches a given value. However, this does require all possible choices to be evaluated, otherwise this cannot be guaranteed. Therefore, this is only feasible for small models. Multiple strategies can also be combined. For example, a model that simulates a car can create a strategy that ensures the car never crashes whilst optimizing the speed. This flexibility combined with the ability to handle randomness makes Stratego a powerful tool for strategy generation.

Reinforcement Learning

Reinforcement learning is an iterative process of making a choice and evaluating whether the choice was good or bad, according to some criterion [28]. The algorithm then uses its past experience to attempt to make the correct choice. Each additional choice that is made increases the knowledge of which choices are good and which are bad. Therefore, it is an iterative process of which the results are better the more iterations are done. In Uppaal Stratego each iteration consists of running a set of simulations using its past experience. The algorithm evaluates the outcome of those simulations and attempts to decide which choices were good and which were bad. This process repeats until the configured maximum number of simulations is reached or until no significant process is being made.

2.8 Valley-filling

Valley-filling [29] is a technique used for calculating the optimal amount of electricity consumption for buffer devices, such as batteries and EVs. It aims to create a flat profile in order to minimize the quadratic cost of the profile. This is achieved by choosing a *fill limit*, also referred to as a *fill level*. At time intervals where less energy is consumed than the fill limit, the buffer device charges to compensate for the difference. An

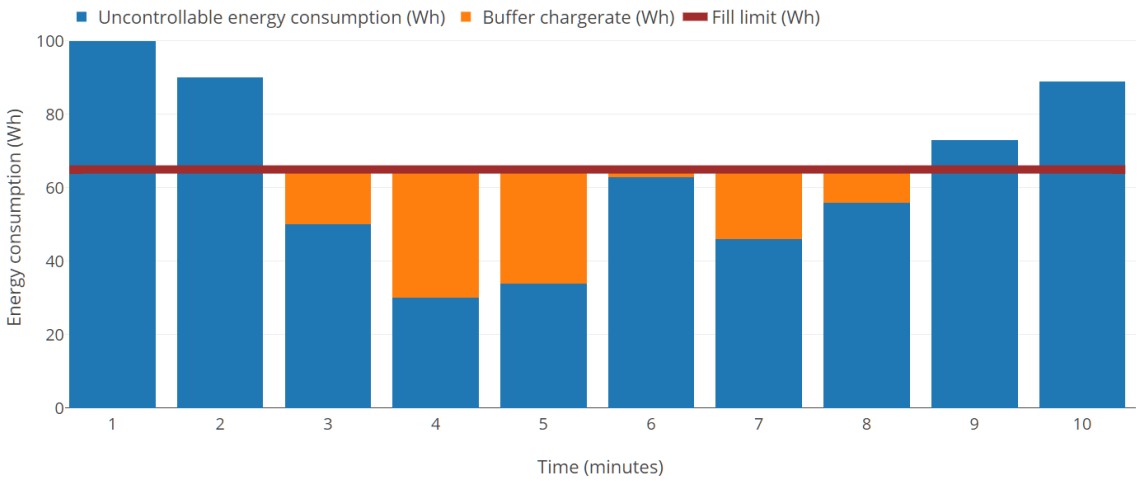

Figure 2.6: Valley Filling Example

example of valley filling is shown in Figure 2.6. In the example, the blue part of the bars is the uncontrollable load. The red line is the fill limit that has been chosen for the buffer device. The orange part of the bars is the amount of energy that is being consumed by the buffer device. As can be seen, whenever the uncontrollable energy consumption drops below the fill limit, the buffer device charges such that the combined load within the interval is equal to the fill limit.

It is very important to choose a correct fill limit, especially for EVs. When the fill limit is chosen below its optimal value, it results in the device charging less than intended, which is especially detrimental for EVs that need to reach a charging target. A fill limit that is too high results in higher peaks, which is undesirable for the electrical grid.

2.9 Offline and online scheduling

Scheduling algorithms can be divided into two classes, namely, offline and online algorithms. Offline scheduling means that the schedule for the entire period is calculated in one time. Online scheduling is used when new information is revealed over time. Because not all information is available at the start, it is more efficient to create the schedule one piece, or sub-period, at a time, waiting for new information before scheduling the next sub-period. An online approach can also be applied to a scheduling problem for which it is infeasible to schedule the entire period at once. In that case, the schedules for all sub-periods are combined into the schedule for the complete period. Depending on the scheduling problem, this can improve the scalability of the algorithm. This solution is often

applied when the scheduling problem becomes exponentially harder for longer periods. The downside is that the schedules could be less good than when calculating the entire period using an offline algorithm. This is due to the sub-periods not taking future scheduling choices into account. The method to somewhat negate this downside is to schedule two sub-periods at a time. After scheduling the two sub-periods, only the schedule of the first sub-period is accepted, meaning that the second sub-period is rescheduled in the next iteration. This results in the schedule of each sub-period taking the scheduling choices for the next sub-period into account.

2.10 Compositional Learning

When scheduling algorithms are required to be able to schedule many devices, it is not always feasible to schedule all devices simultaneously. Instead, the devices can be divided into component groups, each group containing several devices. The devices in each group are then scheduled together. This is called compositional learning. After the devices in one component group have been scheduled, the next component group can be scheduled. Each component group takes the choices, by the devices that have already been scheduled, into account when being scheduled themselves. This splits the scheduling problem into smaller sub-problems. The resulting schedule is most likely not as good as when all devices are scheduled simultaneously, due to the component groups not being able to take component groups that have yet to be scheduled into account. However, when it is not feasible to schedule all devices simultaneously, compositional learning may provide a good alternative. Compositional learning is especially effective when the computation time increases exponentially with the number of devices.

Chapter 3

Related work

This chapter discusses several works of other researchers that relate to this research. It starts by giving an overview of the state of the art in Demand Side Management (DSM) scheduling in Section 3.1. Section 3.2 discusses multiple existing scheduling solutions that make use of Uppaal. Finally, Section 3.3 summarizes the work that has been discussed and the implications it has for this research.

3.1 State of the art

There are multiple different methods for balancing the grid on the demand side. These methods can be split into groups based on how the devices are steered. The two main groups are price-based steering and profile-based steering. The approach taken in this research is similar to profile steering due to the method of calculating the schedules. Therefore, the focus of this section is on the state of the art of profile steering. First, the paper in which profile steering was introduced is discussed. Then, the simulation tools used to implement and test the current version of profile steering are discussed.

3.1.1 Profile Steering

Gerards *et al.* [4] introduces the concept of profile steering. The paper proves that profile based scheduling is in general NP-hard. To work around it they developed a heuristic. This heuristic is similar to what is done in this thesis, although there are slight differences. The heuristic creates a hierarchy from the DSM scenario. The layers within the hierarchy are, for example, a neighbourhood, a house and a device. The heuristic works by first having the top layer, which is the neighbourhood in this example, create its base profile. It then asks the houses to schedule

themselves, such that the cost, given in Equation 3.1, is as low as possible. The houses then ask the same for each of their devices. The devices are the lowest layer and choose a schedule. Once the house has received all of the schedules, the devices are each asked how much they are able to improve the schedule of the house. The device that can improve the schedule of the house the most gets to change its schedule. This continues until the improvements obtained by the devices are insignificant. The schedule of the house, which includes the schedules of the devices, is then given to the neighbourhood. Once the neighbourhood has received the schedules of all houses, it asks which house can improve the schedule the most. This repeats until the improvements made by the houses are no longer significant, the final schedule of the neighbourhood being the result.

This heuristic is quite similar to the algorithm created in this research. The difference is that in our algorithm it is not checked which device can provide the biggest improvement. Instead it is checked whether a device can improve the schedule at all. There is also no layered structure, due to a real world application not being the goal of this research.

$$cost = \int_{start}^{end} {load_i}^2 di \tag{3.1}$$

3.1.2 Simulation tools

Testing DSM approaches is often not feasible in a real world scenario. Therefore, Hoogsteen [13], in cooperation with other researchers at the University of Twente, developed two tools to evaluated the efficiency of different DSM approaches. The first tool is the Artificial Load Profile Generator (ALPG) [30]. This tool is used to create an example DSM neighbourhood scenario. The second tool is the Decentralized Energy Management Kit (DEMKit) [31]. DEMKit makes use of the scenario created by the ALPG and schedules the devices based on the provided algorithm. Both of these tools are detailed below.

ALPG

The ALPG is an open source Python program [18] used to create a DSM scenario. Based on a configuration file the ALPG randomly generates a scenario. The configuration file is used to adjust the types of houses, the likelihood of devices existing in houses and the settings for each device type. This allows the user to quite realistically create a neighbourhood,

including all of the devices inside of the houses. All uncontrollable devices have the amount of energy they consume expressed in intervals of one minute. All controllable devices have their activation intervals indicated in seconds since the start of the year. Each controllable device also has its other relevant settings, such as the battery capacity, given. This gives a thorough description of the scenario.

DEMKit

DEMKit is a Python program developed at the University of Twente (not publicly available). It is used to schedule the controllable devices according to one of the decentralized energy management techniques that have been implemented. Based on the schedules created it simulates the behaviour of the neighbourhood and writes the output to a database. It requires a scenario from the ALPG and a configuration project as its input. The configuration is a Python project that informs DEMKit of what the input data is and which devices to schedule. The configuration project also sets the optimization algorithm and can indicate whether the predictions of the energy usage are always correct. This allows the user to schedule the scenario exactly to their liking and view the output, either by using a third party tool or by reading from the database directly. For use within this research, the configuration project has been made to use exactly the data provided by the ALPG and to use profile steering with perfect predictions. This way, the assumptions about the data should be exactly the same and the only difference in the schedules should come from the difference of profile steering and our algorithm.

3.2 Existing Uppaal scheduling solutions

There already exist scheduling algorithms that make use of Uppaal. This section discusses some of these algorithms and explains the differences with scheduling a DSM scenario. This section is split up into three parts, one for each version of Uppaal. First, Section 3.2.1 discusses several algorithms that make use of Uppaal Cora. Section 3.2.2 details the most interesting case we could find that used Uppaal Statistical Model Checker (SMC) for scheduling. Lastly, two existing algorithms making use of Uppaal Stratego are discussed in Section 3.2.3.

3.2.1 Uppaal Cora

Example applications

In Behrmann *et al.* [16] several applications of Uppaal Cora are discussed. All of these applications are examples of scheduling problems, yet none resemble the DSM scheduling problem. The main difference is that all of the examples have a cost that is relatively steady. For example, each job has a cost associated with it or each processor has a cost associated to its usage. These are linear costs, whereas DSM takes the uncontrollable energy consumption into account. The uncontrollable energy consumption changes relatively often, especially when combining the energy consumption of multiple houses. This increases the state space dramatically.

Another difference is that, in a DSM scenario, all tasks have their own device. The examples only have several devices that can process tasks. In the case of a processor, these devices are the processor cores. Each core can execute one task at a time, which limits the amount of tasks that can be executed concurrently. This reduces the state space compared to all tasks being able to execute at any time within their interval.

The final difference is the variety of the tasks. In all of the examples, there is one type of scheduling being done. DSM requires multiple types of scheduling to be done, due to different device types consuming energy in a different fashion. This makes the model more complex and requires closer attention to detail to ensure there are no timing issues within the model.

Scheduling framework

In David *et al.* [32] an Uppaal Cora framework for scheduling problems is created. Unfortunately, this framework is not able to be adapted to DSM. There are multiple significant differences, all of which occurred in the examples of [16] as well. There are no uncontrollable costs included, all tasks are of the same type and all tasks are executed on some type of resource, such as a processor. These are the types of scheduling problems Uppaal Cora is intended for, therefore, it is logical that both papers discuss similar scheduling problems. This indicates that Uppaal Cora might not be the best option for DSM scheduling.

Multiprocessor scheduling

Ahmad *et al.* [33] uses Uppaal Cora to schedule tasks on a processor with Dynamic Voltage and Frequency Scaling (DVFS). For this they take a static cost into account for each state of the processor. Their implementation has two DVFS states. One state has a higher static cost but completes tasks faster. Switching between the two states cost additional energy.

This is closer to a DSM scheduling algorithm, however, it is still quite far off. Although a static cost is present, the static cost only changes when the model wants it to change. This is different from a DSM scenario, where the uncontrollable load is predetermined and not constant, meaning the model has to change the state after each time unit to be able to access the correct uncontrollable load. The differences of having homogeneous tasks and executing the tasks on one object are present here as well.

Battery scheduling

During this research, batteries also need to be scheduled. Jongerden [19] developed an Uppaal Cora model for the Kinetic Battery Model (KiBaM). This is a realistic battery model that takes into account that batteries can recover energy. The model was used to determine the schedule that results in the longest life time for systems connected to multiple batteries. Because the model is closer to reality than most other battery models, we initially imagined it could be implemented during this research. However, due to how challenging calculating a battery schedule turned out to be, this has not been attempted.

3.2.2 Uppaal SMC

David *et al.* [14] solves quite a difficult scheduling problem using Uppaal SMC, much more complicated than the scheduling algorithms we could find for Uppaal Cora. The problem is heating five rooms with only three heaters. These heaters can be moved throughout the rooms to heat the different rooms. The challenge is to minimize the discomfort to users and minimize the energy consumption. The system contains different outside temperatures over time, which changes the rate at which the rooms lose heat. In addition, the desired temperature changes over time, depending on when the users will be present. The heat generation of the heaters is automated by using two controllers. The only parameters that are given are the thresholds for the two heater controllers. The goal is to find the best thresholds for the heaters.

Uppaal SMC is then used to run simulations using the different threshold values. The model contains randomness, therefore, multiple runs are needed per threshold configuration. Analysis of Variance (ANOVA) is then applied to the simulations to analyse the simulations. When the outcome of ANOVA is that the factors are not significant, additional simulations are run. When the factors are significant the means and errors of each threshold configuration are calculated. Based on the means and errors a Pareto frontier of discomfort and energy is calculated, which is presented as the outcome of the algorithm.

Although this problem is not related to our scheduling problem, it does show what is possible. It calculates good values for quite a complex problem and shows that it is sometimes useful to combine Uppaal with other tools to solve problems.

3.2.3 Uppaal Stratego

Floor heating

In Larsen *et al.* [34] a complex scheduling problem is solved using Uppaal Stratego. The scheduling problem is to heat eleven rooms that are heated using floor heating. Floor heating works by having warm water go through the floor. The following considerations need to be taken into account.

- Heating capacity of the system.
- Behaviour of the doors.
- Physical layout of the water pipes.

This means that not all rooms can be heated at once, the amount of energy required to keep a room warm differs and there are dependencies between the rooms. The dependencies between the rooms are that, if one room needs to be heated, another room is consequently also heated. These dependencies are given by the layout of the rooms. This is due to the water passing through another room before reaching the desired room.

This is a complex problem, because the valve in each room can be opened or closed after every time interval, which they chose as fifteen minutes. The resulting amount of choices each fifteen minutes is then 2^{11}, due to there being eleven vales each with a choice between two options. Because this needs to be done after every interval, the amount of choices for the following t time intervals is 2^{11*t}. This becomes infeasible to calculate very quickly. To solve this problem two heuristics were applied.

The first heuristic is to make the algorithm online. This was done by making the process iterative. During each iteration a strategy is created for the next five time units. Then a simulation is run using the strategy for the next time-unit. The state at the end of this simulation is then stored in the model and the process continues. This means only five time units are calculated at a time, limiting the amount of choices.

The second heuristic is to calculate several components at a time, which is called compositional learning. The valves are split into two component groups. The first component group is then scheduled, assuming the second component group is controlled using a simple deterministic controller. Afterwards, the second component group is scheduled, taking into account the schedule that has just been created for the first component group.

This paper was an inspiration for the online compositional algorithm created in this research. The online part of this algorithm is used, only the length of the scheduling periods is different. The compositional part has been changed during this research. The first change is that the devices that have not yet been scheduled do not make use of a default scheduler, but rather remain inactive. This is due to not having a good deterministic scheduling algorithm to implement. The second change is having multiple iterations within the compositional learning part of the algorithm. Once the second component group has been scheduled, the first component group is rescheduled. When rescheduling the first component group, the schedule of the second component group is taken into account.

Multiprocessor scheduling

Ahmad *et al.* [33] describes a multiprocessor scheduling algorithm using Uppaal Cora. Ahmad *et al.* [15] extends upon this approach by using Uppaal Stratego instead. The setup is mostly the same as the paper using Uppaal Cora, therefore, only the conclusions are discussed.

The model that makes use of Uppaal Stratego is slower than the model that makes use of Uppaal Cora, due to the Stratego model containing uncertainty. However, they did not present the learning parameters, which has a significant impact on the duration. The model is slower due to additional behaviour having been added, which slows down the simulations.

The model was slightly altered from the Uppaal Cora version, because they state that the *cost* variable, which is a hybrid clock, cannot be incre-

mented using discrete jumps on transitions. We noticed this behaviour is possible within Uppaal Stratego, albeit with a different syntax than Uppaal Cora. They most likely thought this was not possible due to the absence of documentation regarding Uppaal Stratego.

Although they encountered problems using Uppaal Stratego, they do state that it has an enormous potential for energy savings in processors, compared to the Uppaal Cora approach.

3.3 Summary

This chapter focussed on discussing work related to the current state of the art in DSM scheduling and scheduling problems that have been solved using Timed Automata (TAs). Profile steering is currently the state of the art in DSM scheduling and has an implementation that is available to us. This implementation can be used to generate schedules based on scenarios created by the ALPG and is used as a reference for determining how well TAs are suited for solving the DSM scheduling problem. Therefore, it is important that we create a TAs model that can schedule such a scenario.

Several scheduling problems have been solved using TAs, however, none of them compare to the DSM scheduling problem. In the related works it became clear that TAs are unable to solve large scheduling problems directly. The problem of scalability was solved by combining Uppaal with external tools in order to create an iterative algorithm. Two such examples are given, one using Uppaal SMC and one using Uppaal Stratego. Due to the DSM scheduling problem being a problem that can scale almost indefinitely, it is clear that such an approach is needed in this research as well. The online compositional algorithm that was created using Uppaal Stratego is similar to how profile steering works. Therefore, such an online compositional algorithm appears to be the most suitable solution when using TAs to solve the DSM scheduling problem.

Chapter 4

About Uppaal

This Chapter discusses several cases that show how details of Uppaal work. Everything that is mentioned relates to the choices made during this research and is referenced in later chapters.

4.1 Missing Functions

As mentioned in Section 2.4, Uppaal has a small subset of *C* as its declaration language. With Uppaal SMC doubles were added and a list of functions [35] that can be used with doubles has been provided. However, only some of these functions have been implemented. As an example, the functions *random* and *floor* have been implemented, but the functions *fint* and *normal* have not. The function that are implemented can be found by looking at the source code of the parser [36], the results of which can be found in Appendix A. It is unfortunate that so many functions have not been implemented, several of which would have been useful or can be useful in other research.

4.2 Input determinism

When using broadcast channels there is only allowed to be one possible transition for each template. This occurs in both Uppaal Statistical Model Checker (SMC) and Uppaal Stratego when running simulations. When a template can choose between more than one transition to synchronize on an error is returned, stating that the transition *violates input determinism*. An example model that proves this is given in Figure 4.1. In the example, *go* is a broadcast channel that synchronizes to a transition with a *select* statement. The select statement returns a number between zero and ten. Therefore, the synchronization has eleven options, which violates the

input determinism constraint. Using the simulator on this model causes the exception to occur.

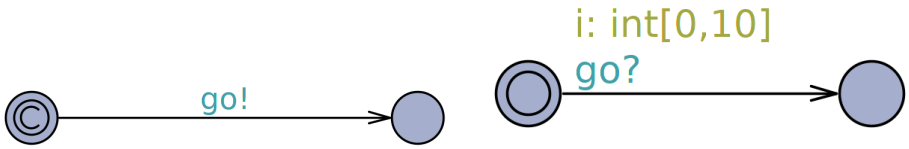

Figure 4.1: Input determinism.

4.3 Committed channels

The order in which transitions are executed is important. In many cases the execution order is straightforward, which is as follows:

1. Synchronization.
2. Committed locations.
3. Urgent locations.
4. Regular transitions.

When introducing broadcast channels this becomes more complex due to multiple instances synchronizing on the same channel, multiple of which could end up in a committed location afterwards. This raises the following questions:

- In what order are synchronizations executed?
- Do committed locations have an order to how they are executed?
- Can Uppaal Stratego arrange the order in which committed location are executed?

To answer the first question, a simple model with a synchronization to multiple instances is used. Each of the instances sets a variable to a unique value corresponding to the instance. An example system is given in Figure 4.2. It turns out the synchronizing transitions are always executed in the order that the instances are defined in the system definition, which is also the order they have in the simulation window. Therefore, the behaviour of the system can be altered by changing the order in which template instances are added to the system definition.

Knowing that synchronizations are always executed in the same order, the second question to answer is whether the committed locations have

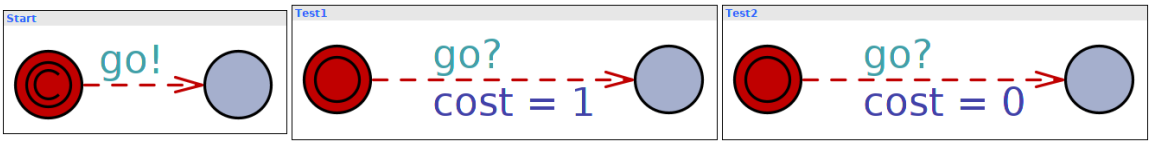

Figure 4.2: Synchronization order example.

an order to how they are executed. Without synchronization it is not possible to have multiple instances in a committed location at the same time. Additionally, it can also be tested whether the committed locations reached by synchronization have priority over the committed location that the emitter transitioned to. An example system can be found in Figure 4.3. When using the simulation window, it can be observed that the value of *cost* ends up at either minus one, zero or one. Therefore, there is no priority between the committed locations.

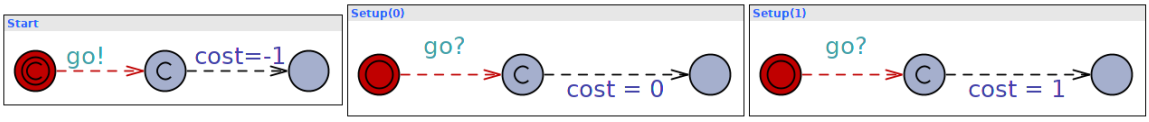

Figure 4.3: Committed order example.

As described in Section 2.7.3, Uppaal Stratego can decide the order in which transitions are taken to get an optimal cost. Since the system can be in multiple committed locations simultaneously, the question arises whether the optimization algorithm can define the order in which the committed locations transition to the next location. We can use the same system as before, only this time with controllable transitions. The example system can be found in Figure 4.4. When minimizing the *cost* variable, the results are either *minus one* or *zero*. Because the result is never *one*, the order of the controllable transitions must be controlled by the optimization algorithm. Although the result cannot be *one*, it can still take one of two values. This means that the algorithm can only control the ordering of the controllable transitions and has no influence as to when these transitions happen in relation to other committed transitions.

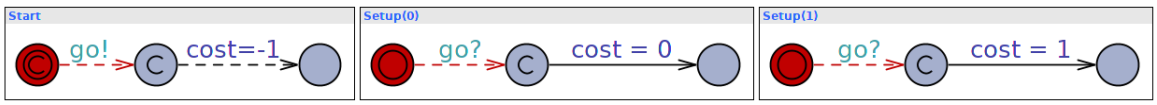

Figure 4.4: Stratego committed locations order example.

Additionally, in order to be entirely confident in how much power the optimization algorithm has over the model, we decided to test whether

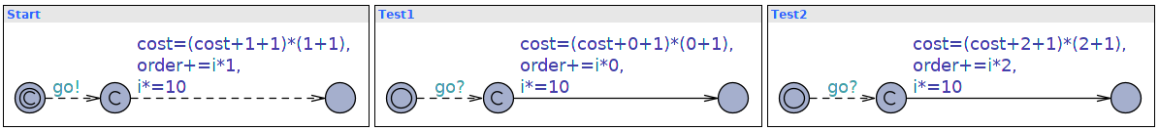

Figure 4.5: Stratego committed locations interleaving example.

all of the committed controllable transitions happen sequentially, or can be interleaved by non-controllable transitions. This could not be seen in the previous test due to the *cost* variable being assigned a new value. Therefore, the exact order could not be determined. To get the exact order, we used the model in Figure 4.5. When optimizing this model there are three orders that could be the result. Each time the two controllable transitions were taken in the same order and the uncontrollable transition was taken either before, in between or after the controllable transitions. Thus it can be reasoned that the optimization algorithm calculates what the most efficient order between the controllable transitions is. The simulator will adhere to this order, but the transitions are still treated as individual transitions when running a simulation.

4.4 Simulation resolution

When running simulations, which is done by Uppaal SMC and Uppaal Stratego, there is one setting that is very important, namely the trace resolution. The result of simulations is a plot, or a textual representation of the plot when using the command line version of Uppaal, that shows the value of expressions over time. The expressions that are shown are determined by the simulation query that is used. David *et al.* [26] states the definition of the trace resolution parameters as follows: "When computing a simulation using the simulate query, the tool filters out the data on-the-fly and retains points that are distinguishable w.r.t. a certain resolution when plotted on a screen. This parameter controls the maximum width of the plot in pixels". Specifically, the last line, stating that the trace resolution controls the maximum width of the plot in pixels, is interesting. This would mean that, when simulating from time x up to, and including, time $x + y$, the trace resolution needs to be $y + 1$ in order to have one data point at each time unit. This is because a resolution of $y + 1$ means there can be up to $y + 1$ different values for expression that is shown, the $+1$ being necessary to include both the start and end values. Any additional data points are not distinguishable with respect to the resolution of $y + 1$, because each column of pixels can only display one

value for each expression. However, to our surprise the trace resolution actually needs to be $x + y + 1$, even though the time from zero until x is not plotted. When using a trace resolution that is too low, data points are removed due to them not being distinguishable with respect to the width of the plot. This caused several problems when developing the algorithm described in Section 5.4.

4.5 Seed

Uppaal uses a random number generator for multiple aspects, most notably the order in which traces are evaluated and when transitions are taken during simulations. By default, the seed is equal to the timestamp of the current time. To be able to reproduce results, Uppaal allows the seed to be set when using the command-line interface.

4.6 Uppaal Stratego Specific

There are several noteworthy aspects specific to Uppaal Stratego, these aspects are discussed here.

4.6.1 Bugs

There are many unresolved bugs in the Uppaal Stratego implementation. Many bugs have already been reported on the bug tracker for Uppaal [37]. None of the reported bugs have been fixed, therefore it is important to mention the most impactful ones and discuss how they affected the research and the resulting algorithms. It should be noted that the bugs are related to the operating system; all of these bugs were encountered on Linux.

The most important bug is one where a correct model results in a time-lock. This bug occurs only in the later versions of Uppaal Stratego, which forced us to use Uppaal Stratego version 4.1.20 for all of the verification queries. The largest downside being that several parameters for the reinforcement algorithm are not included, which could have helped improve the speed and results.

In general, version 4.1.20 is stable. However, there are still some minor bugs which are a nuisance more than a limitation. As an example, exporting a plot as figure results in an error. There have been several other times where a crash occurred, however, these were rare and not directly caused by the model.

4.6.2 Invariants

There is a specific bug that only occurs during simulations in Uppaal Stratego version 4.1.20, which has been fixed in later versions. This bug has an easy workaround, however, to justify the design on the models it needs to be explained. An example model that exhibits this bug is given in Figure 4.6 and a fixed version is given in Figure 4.7. The bug is that any location with an invariant needs to have at least one outgoing transition with a guard that can only contain comparisons with clocks or conditions that hold when the location is entered. Additionally, in case the guard contains conditions on clocks, the conditions are not allowed to have passed already. An example would be a guard with the condition $time == x$, where *time* is a clock and *x* another variable, which would not prevent the time-lock when $time > x$ at the time the location is entered. One solution is to simply add a self-edge with a clock condition that will be never true, such as $time == 1073741822$, which is the largest integer value a clock can take.

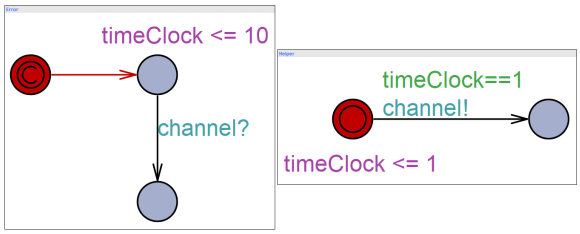

Figure 4.6: Time-locked model.

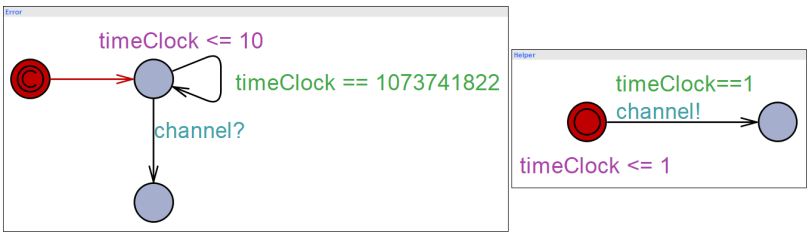

Figure 4.7: Working model.

4.6.3 Memory limit

Uppaal Stratego only has a 32-bit version, meaning the amount of memory it can use is limited to 4GB. When using larger models, as was required for our research, this limit is reached fairly easily using minimization queries. The memory used by a strategy is also not freed when a new model is uploaded. To free used memory the server needs to be restarted,

this can either be done by opening a new model or via the command line. This was an inconvenience while developing the model, due to the loss of previous results and the loss of editing history.

Besides being an inconvenience, the memory limit also means that there is a limit to the amount that can be calculated. This was especially influential when creating an offline algorithm, since everything has to be calculated at the same time. Although an increase in the amount of memory would have allowed for more options to be evaluated, it would not have significant impacted the outcome due to time becoming the limiting factor.

4.6.4 Channels

Similar to Uppaal SMC, Uppaal Stratego requires all channels to be broadcast channels. When using broadcast channels in Uppaal Stratego the uncontrollable transitions do not synchronize when the emitting transition is controllable. This could either be a bug or a feature, however, it is not documented or mentioned in any papers that we have read.

4.6.5 Strategies

Uppaal Tiga [38] is the first Uppaal version that included the possibility of creating strategies, although these strategies cannot be used to minimize or maximize an expression. Uppaal Stratego extends upon this functionality by implementing a reinforcement learning algorithm that can minimize and maximize and expression. In Uppaal Tiga it is possible to create strategies and print them in a human readable format. We discovered that this is not possible on Linux for the versions that are available. On Windows we were able to use Uppaal Tiga to print a strategy, using version 0.17, the same version did not print the strategy on Linux. Unfortunately, neither Windows nor Linux is able to print strategies for Uppaal Stratego. The printing of strategies is implemented, or at least the required strings are included in the binary. However, we were unable to find any combination of parameters that results in the strategy being printed, which is in line with the findings of Ahmad et al. [15]. We assume this is a bug that was never fixed, although we cannot be certain due to receiving no response from the developers.

The strategies could have made parts of this research easier and would provide more insight into what is happening. Depending on what such a controller looks like it could be possible to generate a controller based

on semi random input and then apply it to an actual scenario. This would have been an interesting addition to the research, which perhaps can be done in future research if it becomes possible to extract strategies.

4.6.6 Learning settings

Uppaal Stratego has several settings that affect the learning behaviour of the optimization algorithm. There are the learning filters, learning methods and learning parameters. Each of these settings is described below.

Learning filters

The learning filters are used to filter the simulations and only choose the best simulations, which will then be used for learning. There are three choices for the learning filter, namely global, local and sweep. We were unable to find a good description of each type of filter, however, we noticed that the global filter gave slightly better output and therefore the global filter is used throughout the research. We have an assumption of how it works, however, due to a lack of documentation and no way to validate the assumption, this has been omitted.

Learning methods

The learning method is the learning algorithm used to extract a strategy from simulations. The learning method also alters the structure in which the data is stored, which is required for the different learning algorithms. The method has four options, namely co-variance, splitting, logistic regression and naive, which are best explained in [28]. The model created during this research, which is described in Section 5.4, executes all its main functionality at the start of each new time unit. This results in all of the choices happening without any time passing. Because co-variance and splitting are both used to optimize based on the clock value, these methods are not effective. Logistic regression creates a strategy where each transition is labelled with a weight, which is exactly the type of strategy that is useful for our model. When testing the difference in results between the different learning methods, we noticed that logistic regression resulted in strategies similar to Decentralized Energy Management Kit (DEMKit), whereas co-variance, splitting and naive learning methods all gave significantly worse outcomes for timeshiftables and batteries, whilst giving the same result for Electrical Vehicles (EVs).

Therefore, throughout the rest of this research, the logistic regression learning method has been used.

Learning parameters

There are seven learning parameters, however, three of these are limited to Uppaal Stratego version 4.1.20-4. Because we are using Uppaal Stratego version 4.1.20, the three parameters that do not exist in this version are omitted. These parameters are as follows:

- The maximum number of runs.
- The number of successful runs.
- The number of good runs.
- The number of runs to be evaluated.

The only explanation we were able to find for these parameters is the one given in David *et al.* [28]. The maximum number of runs is the maximum number of simulation runs that are used for learning. The number of successful runs is the maximum number of simulations runs that were successful, which is all runs in our case. The number of good runs is the number of runs that are used for learning. The number of runs to be evaluated is the number of simulations that are run to change the non-deterministic strategy into a deterministic one.

Chapter 5

Methodology

This chapter details the different approaches that have been used to generate a schedule based on the data from the Artificial Load Profile Generator (ALPG). First, in Section 5.1, it is described how the ALPG data is converted into a representation that can be used by Uppaal. Then, Section 5.2 discusses the structure of the data and the impact it has on using Timed Automata (TAs). Afterwards, Section 5.3 discusses how Uppaal Statistical Model Checker (SMC) can be used to generate schedules and why it is not suitable. How Uppaal Stratego can be used to generate schedules is described in Section 5.4. The multiple algorithms that have been created using Uppaal Stratego are described in Sections 5.5, 5.6, 5.7 and 5.8. In Section 5.9 it is explained why Uppaal Cora was not used and how, in future work, it could still be valuable. Lastly, Section 5.10 discusses the models that have been made and proposes several ideas for improvement.

5.1 Data representation in Uppaal

Before discussing how Uppaal can be used to generate schedules, it is important to understand how the ALPG data can be accessed by Uppaal. The only inputs that Uppaal can use are the model and queries, as described in Section 2.4. Therefore, the data needs to be stored directly inside of the model. There are many different formats in which the data can be stored, due to the Uppaal declaration language being a subset of the programming language *C*. We have chosen to use several structures, each of which can represent part of the data, for their ability to remain readable for humans. The structures used are given in Figure 5.1, in which a relation indicates that the structure contains data with the type of the related structure. The textual definition, as declared in Uppaal, can be found in Appendix B.

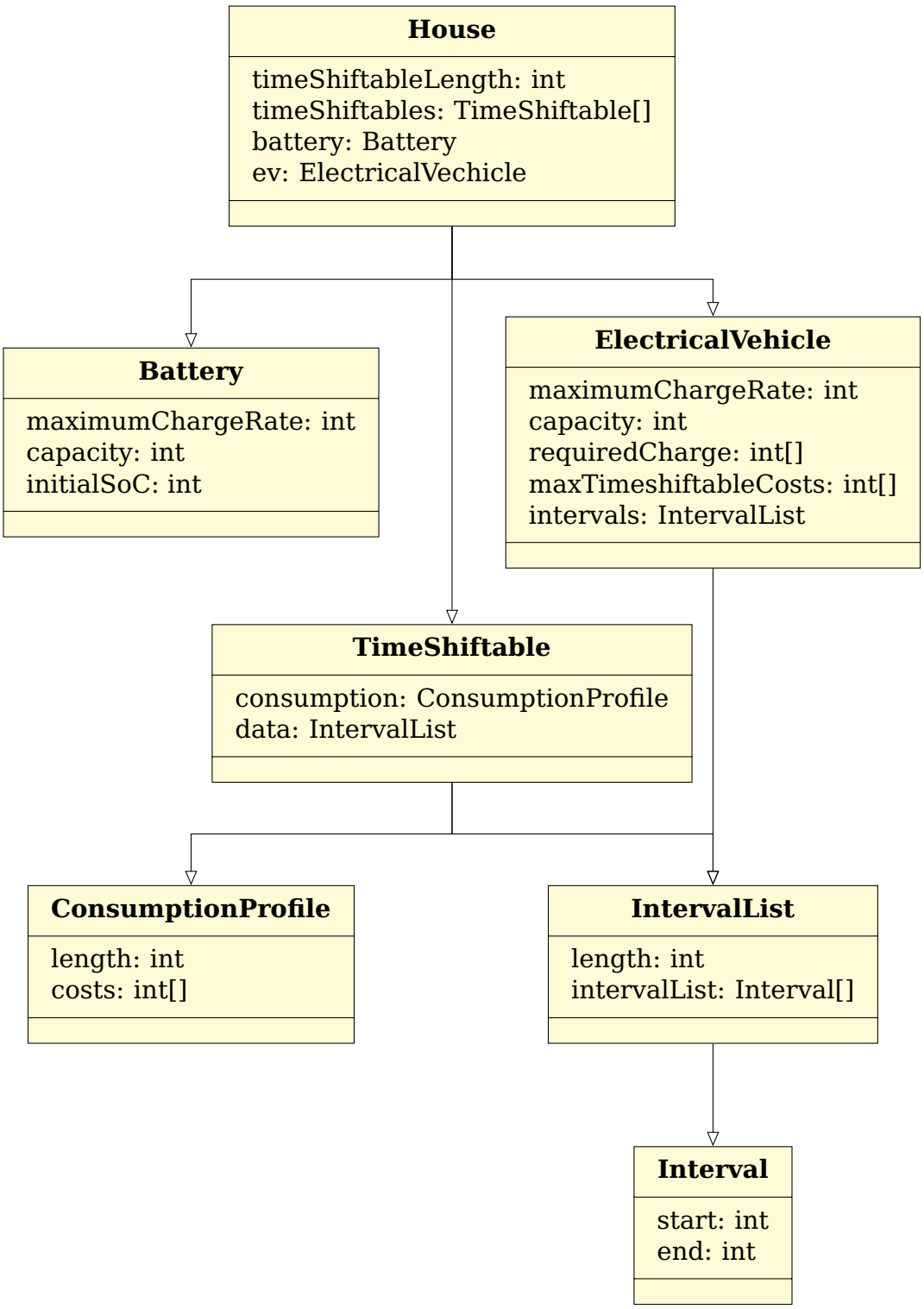

Figure 5.1: Visual representation of the data Uppaal structures.

In Uppaal the ALPG data is represented as a list of *House* elements, each element containing references to all of the data required by the devices. This list, and every piece of data stored in it, is declared as a *constant*, such that it does not impact the memory size of each state. When devices access the data they use their house and device identifiers, which are given as device parameters, as indexes for the lists. This format allows devices to quickly find their relevant data and remains readable for humans.

The data from the ALPG is parsed using a Python script and rewritten into the Uppaal representation making use of the aforementioned structures. The Python script that is used is given in Appendix C. The conversion from ALPG data to Uppaal definition is for the most part a rewrite of the data, with the exception of lists, doubles, edge cases, maximum timeshiftable costs and reactive values.

5.1.1 Lists

Lists are needed for several data elements, such as intervals. The downside of lists in Uppaal is that their length needs to be predefined by a constant value. A workaround for this is to define all lists as their maximum possible length, pad the end of the array with filler data and keep a counter indicating the length of valid data. This workaround is present in Figure 5.1.

5.1.2 Doubles

For timeshiftable devices, the energy consumption is defined as doubles by the ALPG. Uppaal Cora does not use doubles, as described in Section 2.7.1, therefore all doubles have been converted to integers. To account for this minor alteration, the actual energy values have to be recalculated in Python, using the schedule that has been created by the Uppaal model.

5.1.3 Edge cases

The Decentralized Energy Management Kit (DEMKit) algorithm requires at least one day of data more than will be scheduled, as explained in Section 3.1.2. The ALPG can create intervals that can be scheduled either in the last or second to last day. By scheduling the device in different days, the comparison between DEMKit and Uppaal would become unfair. This is due to one of the schedules having completed more work. To prevent this, any intervals with overlap between the final two days are removed.

The resulting schedule is still realistic and, due to the algorithms being unbiased, should give a fair comparison.

5.1.4 Maximum timeshiftable costs

To speed up the model, each charging interval of an Electrical Vehicle (EV) has an associated maximum timeshiftable cost. This value contains the maximum energy that can potentially be consumed by all timeshiftables during the charging interval. How this value is used is described in Section 5.4.

5.1.5 Reactive values

The timeshiftable devices have both an active and reactive energy consumption value, represented by the ALPG as the real and imaginary parts respectively. These values could both be considered, in which case their values should be summed for the Uppaal model. However, DEMKit can optimize for only the active energy consumption as well, therefore a choice had to be made whether to consider both or only one of these values. The choice was made to only account for the active energy consumption, due to this being the default for DEMKit and it allows for easier validation of the model.

5.2 Data discussion

Before explaining how we attempted to use Uppaal to schedule the devices, a brief explanation about the data is required. The uncontrollable part of the data provided by the ALPG is a piecewise linear function, with each minute being a new piece. This means that each minute has its own energy consumption associated with it. Although it is logical that the data look like this, it is counter intuitive to use this type of data inside of Uppaal.

To optimize the model, an optimization criterion needs to be calculated. For our research, the optimization criterion is the quadratic function of the energy consumption. Calculating this criterion requires the uncontrollable costs to be retrieved at every time unit. Because this requires a different index at every time unit, the state of the system must change after every time unit. Because a clock cannot be used as an array index, it is required to keep an integer which is always equal to the current index of the array. The result is that the clock variables have almost no use

within the system. This goes against the idea behind TA and increases the state space. However, as far as we are aware there is no other good way of calculating the optimization criteria.

5.3 Uppaal SMC

Uppaal SMC can be used to run simulations, which in turn can be used to find a schedule. The most basic method is to run a number of simulations and pick the best one. A more sophisticated method is given in [14] and described in Section 3.2.2.

Running several simulations and picking the best one works well for small scale situations, however, for larger situations the odds of getting an energy efficient run are too small. Uppaal Stratego makes use of the same models as Uppaal SMC, albeit with some transitions being indicated as controllable. Therefore, the models that are used are described in Section 5.5.

We decided against attempting to apply the algorithm used in [14], due to a similar and more closely related problem being solved on a larger scale in [34]. This more recent approach makes use of Uppaal Stratego to generate schedules and enables scalability by splitting the problem into multiple steps. This method matches closely with the approach profile steering takes and therefore we anticipated this would yield good results. The next sections further describe this process and explain the implementation.

5.4 Uppaal Stratego

Uppaal Stratego provides a unique opportunity for scheduling, as was explained in Section 2.7.3. This is due to its ability to create a controller that tries to minimize an expression. In reality, the devices inside of a smart grid have a controller that makes the scheduling decisions, therefore, having an algorithm that generates a near optimal controller is an intuitive choice. Although the controllers from Uppaal Stratego cannot be used in a real world scenario, due to the controller needing to have the full state space information, they can be used to create a schedule. This is done by running a simulation that uses the controller to determine which actions are taken by devices at what time. Since the controller tries to minimize an expression (in our case the sum of the quadratic energy consumption), the simulation result should be an

efficient schedule.

Although Uppaal Stratego fits the problem well, it is not a silver bullet. There exist several problems with the software, which have already been described in Section 4.6. Regardless of the problems encountered, several algorithms have been developed using Uppaal Stratego. An offline algorithm, discussed in Section 5.5, has been developed. This algorithm uses solely the reinforcement learning algorithm within Uppaal. To improve scalability in the time domain, an online algorithm has also been created, which is explained in Section 5.6. To improve the scalability with the number of devices, a compositional algorithm has been developed, which is detailed in Section 5.7. Finally, in Section 5.8 the online and compositional algorithms are combined into an online compositional algorithm that scales well with time and the number of devices.

5.5 Offline algorithm

The offline algorithm calculates the entire schedule in one go. An offline algorithm can result in the optimal solution, due to every possible schedule being considered. However, it has poor scalability due to the problem being in general NP-hard as described in Section 2.1. The entire schedule is created within Uppaal and both the online and compositional algorithms described in later sections make use of this same model.

The model consists of several templates. There are three templates that are always required, namely the *Time*, *CostCalculator* and *MainLoop* templates. There are also templates for each device type, which are the *TimeShiftables*, *EVs* and *Batteries*. Each template is described below, which details their purpose and the reasoning behind the implementation.

Combined these templates form the entire system. Due to the method of creating instances of templates, modelling another ALPG scenario requires only the global declaration to be replaced by the rewritten version of the ALPG scenario. The *MainLoop* is the heart of the system and calculates how much energy each device is consuming, whereas the devices only need to inform the *MainLoop* of the choice they made. The devices can inform the *MainLoop* by setting local variables that are references to global variables stored in arrays. This ensures that devices do not interfere with each other and keeps the model readable.

5.5.1 High level model explanation

Each template in the model has one specific task and together they form the entire system. As mentioned before, there are three templates that are required, namely, the *Time*, *CostCalculator* and *MainLoop* templates. These templates must have exactly one instance for the model to work. The *CostCalculator* template keeps track of the sum of the quadratic energy consumption, which is used by the optimization algorithm to find the most efficient schedule. The *Time* template ensures that the *MainLoop* template starts a new loop at every time unit. The *MainLoop* template orchestrates all the different devices and calculates how much energy each instance of a device template has consumed. Each instance of a device template determines when the device activates and sets the variables required for valley-filling. The *MainLoop* then uses the information it has to calculate how much energy each device is consuming or producing.

Calculating the schedule is done by running the Stratego optimization algorithm and using the resulting strategy to run a simulation. The optimization works well and schedules can be created for small scenarios. Thus we decided to use these same model in both the online and compositional algorithms described in Sections 5.6 and 5.7 respectively. Be that as it may, problems arise when the system is too large or the duration that has to be optimized is too long. When the optimization algorithm cannot explore enough paths, the resulting schedule is bad, as explained in Section 2.7.3. This is especially troublesome when taking into account the memory limit as described in Section 4.6.3 (which is easily reached).

5.5.2 CostCalculator

Architecture

The *CostCalculator* template is, as its name suggests, the template that calculates the accumulated costs. The *CostCalculator* is the only template that changes the *cost* variable. Therefore, it has full control over the optimization criterion used. This allows for substitutions of the optimization criterion without knowing the specifics of the model.

Design choices

The optimization criterion that is used during this research is the strain that is put on the electrical grid. The strain on the grid at any given

time is calculated as the square of the energy consumption. Calculating the strain on the grid over time leads to Equation 5.1, in which *energyConsumption(i)* is the total energy consumption at time *i* and *time* is the current time interval of the model. This formula must hold during the entire simulation of the scenario.

Uppaal specifics

The Uppaal template that has been created to match the aforementioned behaviour is given in Figure 5.2. The *cost* variable has been implemented as a hybrid clock, hence, it is not taken into account when representing each state, thereby reducing the state space. The rate of the *cost* variable is then set to the square of the combined energy consumption. The energy consumption is split into three parts, namely, the *HouseBases*, the *HouseDynamicCosts* and the *HouseBatteryCosts*.

The *HouseBases* is the energy consumption that cannot be controlled, such as lights, solar panels and several other devices. This is stored inside of the Uppaal model as a constant array, in which each element is the sum of all uncontrollable devices during that interval. Using an integer representation of the time, named *intTime*, gives the amount of uncontrollable energy for the current interval. Because *intTime* is automatically incremented in the *Time* template, this ensures that the correct amount of uncontrollable costs is always used.

The *HouseDynamicCosts* represents the summed costs of all time-shiftable devices during the current interval. This value is recalculated each interval by the *MainLoop* template, ensuring that the correct costs are always taken.

The *HouseBatteryCosts* is the combined energy consumption or production of all buffer devices, which includes both EVs and batteries. This value is also recalculated by the *MainLoop* template. Positive values indicate energy consumption, whereas negative values indicate energy production.

The combined energy consumption is divided by the *reductionFactor* before being squared. This is required to prevent the *cost* variable from overflowing. The *reductionFactor* is calculated when creating the Uppaal declaration from the ALPG data. Once there are no more static costs to calculate the template transitions to another location, ensuring that the *cost* is no longer increased. This transition ensures that the model will not give an array index out of bounds exception.

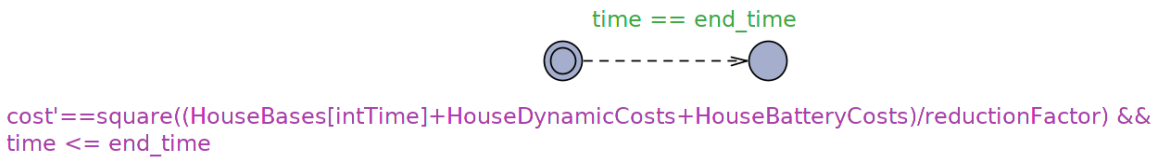

Figure 5.2: CostCalculator Template

$$cost = \int_0^{time} energyConsumption(i)^2 di \tag{5.1}$$

5.5.3 Time

Architecture

The purpose of the *Time* template is to maintain the counter *intTime* and to notify the *MainLoop* template when one time unit has passed. The *intTime* needs to be equal to $floor(time)$ at all times. The *MainLoop* template then uses the newly set *intTime* to calculate the values for the controllable devices. It is therefore important that the *intTime* is set before the *MainLoop* is notified.

Design choices

The functionality of this template and the functionality of the *CostCalculator* template could have been combined, however, we decided against doing so. Having both functionalities split over two templates provides modularity and clarity. It also means that alterations are less prone to mistakes.

Uppaal specifics

The *Time* template is given in Figure 5.3. The template accomplishes its task by keeping an internal clock *t* that is reset after every time unit. Combined with the invariant, this ensures that a transition is taken exactly at the end of every time unit. The transition then increments the *intTime*, which thereby complies with the $floor(time)$ formula, assuming that $time == intTime$ at the start. This only holds when the rates of *t* and *time* are always set to *one*, which is their default value. The *MainLoop* template is then notified using the *step* channel, which happens once the assignments have finished, as explained in Section 2.4.

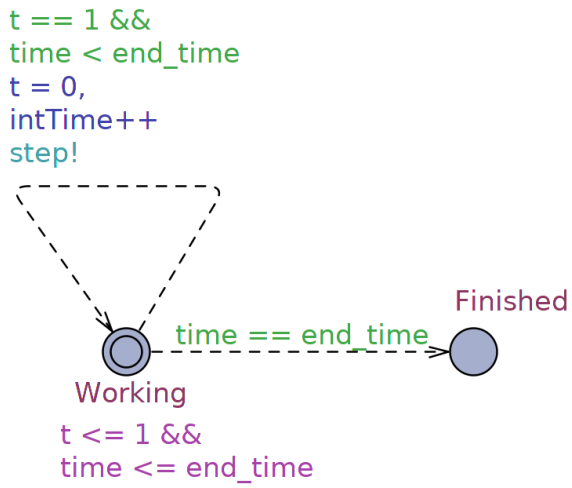

Figure 5.3: Time Template

When the model has finished, which happens when *time* is equal to the length of the array with uncontrollable energy consumptions, the template transitions to the *Finished* location. This ensures that the *MainLoop* is not notified, thereby improving the performance and preventing indexing errors.

5.5.4 TimeShiftables

Architecture

The purpose of the timeshiftable template is to inform the *MainLoop* template of the times at which the timeshiftable is active. Informing the *MainLoop* template is accomplished by storing the start time inside of a variable. It is the responsibility of this template to ensure that the timeshiftable device complies with the starting and ending time constraints given by the ALPG. This includes ensuring that the timeshiftable device is started at all and ensuring that the device is not started more times than required.

Design choices

The choice has been made that a timeshiftable device cannot be interrupted during its routine. In reality there are timeshiftable devices that can take a break during their routine, however, this is not included in the ALPG data and DEMKit also assumes this is not possible. This means that for each interval one choice has to be made, namely, the time at which the timeshiftable starts its routine. Each time during the interval should

be a possible choice for the controller, such that the best schedule can be found.

The *MainLoop* template calculates the energy consumption for all timeshiftables. This lowers the amount of state changes, thereby improving efficiency. Having the *MainLoop* control the energy consumption also means that the timeshiftable template can remain small and readable.

Uppaal specifics

The template for the timeshiftable devices is given in Figure 5.4. The timeshiftable is instantiated using unfilled parameters, as explained in Section 2.4. The unfilled parameters are a *house id* and a *device id*. This means that there always exist two instances of the template for each house, due to each house having at most two timeshiftable devices according to the ALPG. Not all of these instances are necessarily used. Therefore, the first step is to check whether the timeshiftable should exist, which is done using the local *active* constant. This constant checks whether the house, obtained using the *house id*, has a timeshiftable with its *device id*. If the house does in fact have a device with that *device id*, it is also checked whether the timeshiftable has any intervals associated with itself. A timeshiftable that is not supposed to exist immediately transitions to the *Finished* location and is, hence, inactive.

In case the *active* variable evaluates to *true*, the template enters its main loop, starting in the *Waiting* location. The template has a local variable that indicates the interval the template instance is currently in. The current interval can then be used to get the start and end times. To give the controller a choice at each time unit during the interval, a synchronization on the *tsStep* channel is used. The *MainLoop* emits a signal on this channel every time unit, thereby allowing the timeshiftable to activate at every time unit. The synchronization is blocked by a guard that ensures that the current time is larger or equal to the start time of the interval, ensuring that the timeshiftable does not activate too early. After the synchronization, the template is in a committed location. This location has two controllable transitions that can be taken, allowing the Stratego optimization algorithm to control which transition is taken. One transition activates the timeshiftable and the other returns directly to the *Waiting* location, remaining there until another synchronization occurs. The transition that returns directly to the *Waiting* location has a guard, ensuring that the current time is not the latest possible activation time. This means that, if the current time is the latest possible activation

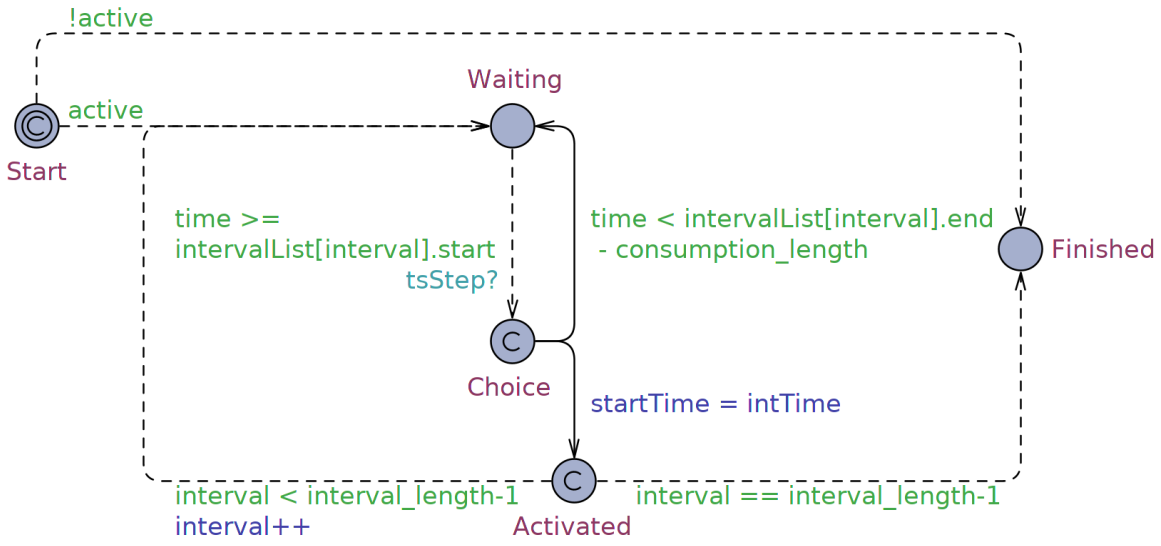

Figure 5.4: TimeShiftable Template

time within the interval, the timeshiftable must activate. This ensures compliance with the ALPG restrictions. The latest possible activation time is calculated by subtracting the amount of time the timeshiftable takes to do its program from the end of the interval.

When the choice is made to start the timeshiftable, the *startTime* variable is set. The *startTime* is a reference to a unique index in an array of start times, allowing the *MainLoop* template to access the *startTime* of every timeshiftable. The *startTime* is set to the current time, ensuring that the *MainLoop* knows which energy consumption needs to be taken. After setting the *startTime*, the template either transitions to the *Finished* location or increments the *interval* variable and returns to the *Waiting* location. The transition that is taken depends on whether the timeshiftable has to activate at least once more. When this is not the case, the timeshiftable transitions to the *Finished* location, indicating it has finished all of its tasks. In case the timeshiftable still has more work to do, the transition to the *Waiting* location is taken. This process continues until the timeshiftable reaches the *Finished* location.

5.5.5 Electrical Vehicles

Architecture

The EV template has one purpose, namely, to determine the fill limit at the start of each charging interval. The fill limit needs to be chosen such that the strain on the grid remains as low as possible. The *MainLoop* template uses the fill limit to calculate the energy consumption of each

EV. When the EV is not going to reach its charging target, the *MainLoop* template will increase the energy consumption at the end, such that the target is reached.

Design choices

The choice was made to always have an EV reach its charging goal, even when the fill limit does not allow it. This was done such that no simulation runs have to be discarded, thereby improving efficiency.

To choose a correct fill limit, a lower limit and upper limit are calculated. The lower limit is the optimal value considering there are no timeshiftable devices. The upper limit is calculated by adding the maximum possible timeshiftable costs during the interval, divided by the length of the interval, to the lower limit. The upper limit is not the optimal value in case all timeshiftables are activated, due to the maximum timeshiftable costs having been calculated using Python. This was done to reduce the computation time during simulation of the model. Choosing a fill limit is done by taking a random value from the uniform distribution between the lower limit and upper limit. In reality there is a larger probability that the optimal fill limit is towards the lower limit, therefore, a normal distribution could have been a better solution. However, using a normal distribution would incur a high penalty in case a high fill limit is required, due the spacing between the choices being large. Therefore, we decided to use a uniform distribution instead, such that the algorithm does not create a bias for scenarios that benefit from lower fill limits.

The *MainLoop* template calculates the energy consumption of EVs for a multitude of reasons. The reasons why the *MainLoop* calculates the timeshiftable energy consumption apply here as well, namely, to increase performance and maintain readability of the template. Calculating the energy consumption for all EVs in one place also has the advantage of being able to control the calculation order. The order in which the energy consumption of the EVs is calculated is imported due to how the consumption is calculated. The calculation subtracts the combined energy consumption from the fill limit. The EVs for which the energy consumption has already been calculated are included in the combined energy consumption. Therefore, the order of the energy consumption calculation has a large impact on the resulting schedule. Because the energy consumption is done by the *MainLoop* template, the calculation order can be controlled which improves the results.

Uppaal specifics

The automaton of the *EV* template can be found in Figure 5.5. Similar to the *Timeshiftable* template, the first step is checking whether the EV exists in the scenario and whether the EV has any charging intervals. When this is the case, the template finds itself in the *Waiting* location. The template will transition from the *Waiting* location to the *Choice* location at the start of the interval. This transition is triggered by the *MainLoop* emitting on the *tsStep* channel. A channel is used rather than an invariant, in order to ensure the order in which actions happen. The *Choice* location is committed and immediately requires the fill limit to be chosen. The *fillLimit* is a reference to a unique index in an array, thereby allowing the *MainLoop* to view the chosen fill limit of each EV.

The fill limit is chosen using the *getLimit* function, which takes a value out of the *possibilities_t* type. This custom type is an integer in the range of zero until thirty. The *possibilities_t* determines how many parts the uniform distribution is split into, because it is not feasible to simulate every integer value of the distribution. The transition uses a *select* label to duplicate the transition for every value *possibilities_t* contains. This *select* label is the reason that the fill limit cannot be chosen directly when transitioning from the *Waiting* location, as explained in Section 4.2. Because the transition is controllable, the Stratego optimization algorithm can determine the best value from the *possibilities_t* to be taken. When one of the transitions is traversed, the value selected from the *possibilities_t* is used as input for the *getLimit* function. After the fill limit is selected, the interval counter is incremented. If there is another charging interval the template will return to the *Waiting* location. When there are no more intervals the template transitions to the *Finished* location.

Calculating the fill limit The *getLimit* function first calculates the lower limit and then uses the input value to evenly divide the maximum timeshiftable costs. As mentioned above, the maximum timeshiftable costs have been pre-calculated using Python and are stored in the Uppaal declaration as an attribute of the EV. The calculation of the lower limit needs to be done inside of Uppaal, due to this relying on the fill limits chosen by other EVs. Algorithm 1 explains how the lower limit is calculated.

The algorithm starts by storing the maximum energy consumption at each point in time during the interval, assuming there are only static costs and EVs. These energy consumptions are sorted from low to high.

The energy consumptions are then filled until the charging target is reached, using the sorted list of maximum energy consumptions. This is the same process as valley-filling, only the fill limit increases until the charging target has been reached. Filling the energy consumptions is done by adding the difference between the current consumption and the current fill limit to all energy consumptions that are lower than the current value. Because the consumptions are sorted, all consumptions below the current consumption are filled up until the same level.

Algorithm 1 Calculate EV lower limit

```
baseCosts = list()
for i in range(interval.start, interval.end) do
    baseCosts += max(static[i], getHighestEVFillLimit(i))
sortedCosts = sort(baseCosts)
neededCharge = interval.requiredCharge
previousCost = 0
for index, cost in enumerate(sortedCosts) do
    if (cost-previousCost)*index >neededCharge then
        return previousCost + neededCharge/index
    else
        neededCharge -= (cost-previousCost)*index
        previousCost = cost
return prevCost + neededCharge/(interval.end-interval.start)
```

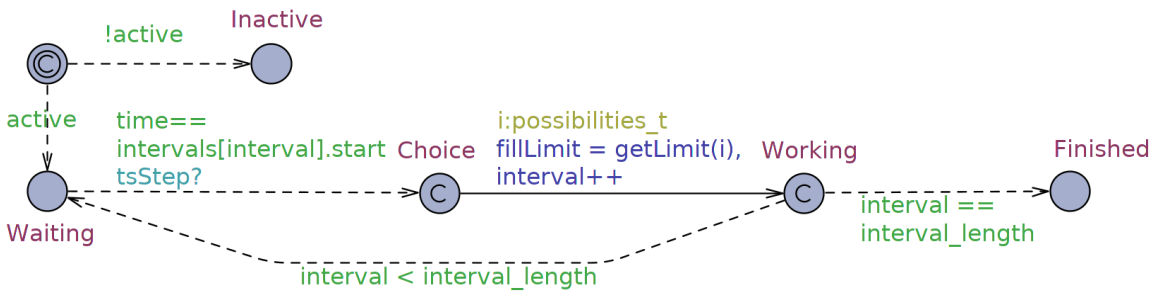

Figure 5.5: Electrical Vehicle Template

5.5.6 Batteries

Architecture

The *Battery* template is responsible for setting the fill limit of the battery. This is different from an EV, because the fill limit can be set at any time and needs to be reset to its default value when the battery deactivates. The battery also has multiple reasons to stop charging or discharging. This

can happen when the State of Charge (SoC) has reached the capacity of the battery when charging, or when the SoC reaches zero whilst discharging. However, the battery can also turn off because there is no longer a need for the battery, at least not with the same fill limit. The *Battery* template also needs to ensure that it is always in the correct location. The *MainLoop* calculates the amount of energy that is charged or discharged, thereby ensuring that the SoC never exceeds the capacity and never falls below zero.

Design choices

Batteries are difficult for Uppaal to optimize. This is due to the amount of possible schedules. At each time unit the battery can decide to stay in the current location, either active or inactive, or decide to switch to the other location. This means each time unit has two choices, which results in 2^{192} choices when scheduling two days with fifteen minute intervals. Due to Uppaal being unable to calculate a schedule for a battery when all options are left open, several restrictions had to be put into place. These restrictions aim to restrict as many options as possible, whilst leaving most of the good schedules unrestricted.

Merged battery A choice was made to merge all batteries into one battery that has the combined capacity and available charge rate of the original batteries. This can lead to impossible schedules when batteries have different capacities or maximum charging rates. However, we assumed this to be a small chance due to the odds of having multiple batteries being slim and due to the ALPG choosing one of three predetermined sets of values for a battery. When there exist different batteries in one scenario, the odds are that the schedule can still be made using these batteries. In case a schedule that cannot be created using the batteries does occur, it should not be too challenging to find battery schedules that, when combined, resemble the schedule of the merged battery.

Always improving The second restriction is to only allow charging and discharging when it lowers the quadratic cost function. This eliminates all schedules that worsen the quadratic cost at one time unit to improve it at a later time unit. This can prevent the model from finding optimal schedules, however, disregarding these schedules lowers the amount of possible schedules considerably.

Turn off when not (dis)charging Using the restriction that the battery can only lower the quadratic cost function, we decided that batteries must turn off when the (dis)charging rate reaches zero. Stopping earlier would result in an inefficient schedule, due to the profile having a peak, and stopping later would be identical as switching off and on again, due to batteries getting the choice to switch on immediately after switching off. The one exception to this restriction is when the battery is either completely full or completely empty, in which case the battery needs to switch off in order to comply with the constraints of the battery. This only eliminates inefficient schedules, thus is a good improvement.

Limit activation possibilities We also made the choice that a battery can only turn on when the total cost in the previous time unit was closer to zero. The one exception to this restriction is when, in the previous time unit, energy was being produced and in the current time unit energy is being consumed, or vice-versa. This eliminates all situations in which it would have been beneficial if the battery had turned on earlier, thereby eliminating a set of schedules that are suboptimal. This drops the amount of options significantly without removing the best schedules.

Valley-filling restrictions Finally, the options for the fill limit need to be limited. Because of valley-filling and the restrictions put on the model, the fill limit has to be in between the current combined energy consumption and the previous combined energy consumption. The reasoning behind this is that a value further away from zero would not charge and thus stop immediately. Whereas going closer to zero than the previous energy consumption means the battery should have turned on earlier to create an optimal schedule. This calculation already takes into account the energy consumption set by timeshiftables and EVs. The fill limit is chosen as one of the values of a uniform distribution between the two values. The amount of values to pick from this uniform distribution needs to be limited, due to each option exponentially increasing the amount of choices for the controller. This amount of choices could be determined based on the scale of the situation being modelled, however, we chose to leave it at only two choices to limit the impact of the scale on the results. As is shown in Chapter 7, scheduling batteries already takes an enormous amount of time with only two options.

The aforementioned restrictions on the battery drastically reduce the amount of possible schedules. This allows Uppaal to find a schedule for

batteries. Several of the restrictions only prevent suboptimal schedules, however, some of restrictions can prevent the optimal solution. Considering that this is the only solution to enable Uppaal to generate a battery schedule it is an acceptable sacrifice.

Uppaal specifics

The template of the *Battery* is given in Figure 5.5. The first step of the template is to check whether the capacity of the battery is zero, in which case the battery should not exist within the scenario. When the capacity is not zero, the template enters the *Inactive* location.

In the *Inactive* location the battery gets the choice to activate every time unit, as long as the *startCondition* holds. This condition checks whether the previous combined energy consumption was lower than the current combined energy consumption. The *startCondition* also checks whether the SoC is not equal to the capacity in the case of charging or not equal to zero in the case of discharging. Thereby, the condition ensures that the restriction to always improve is met and that the battery will not immediately return to the *Inactive* location. When the choice is made to activate the battery, the transition is made to the *Working* location. The transition sets the *batteryFillLimit*, which is a reference to a unique index in an array, such that the *MainLoop* template can obtain the *batteryFillLimit* value. The fill limit is set to a uniformly distributed value between the previously combined energy consumption and the current combined energy consumption. This transition is controllable, such that the Stratego optimization algorithm can decide the best fill limit.

The template transitions back to the *Inactive* location when the SoC has reached either of its limits. The battery also transitions back when the combined cost, without taking any batteries into account, is closer to zero than the fill limit. This means the battery would not charge, therefore, it might as well switch to the *Inactive* location.

The *batteryFillValue* is included to keep track of how much energy each Battery is consuming. This variable is a reference to a unique index in an array as well. The *batteryFillValue* is set by the *MainLoop* template, however, it needs to be set to zero when the battery switches back to inactive. To reduce the state space, the array of battery fill values is marked as *meta*.

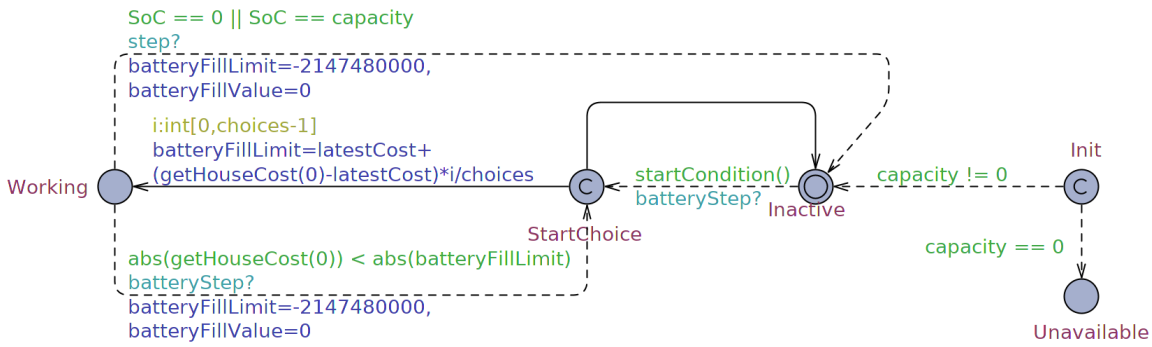

Figure 5.6: Electrical Vehicle Template

5.5.7 MainLoop

Architecture

The *MainLoop* template acts as an orchestrator for all of the devices. It notifies devices they have to make a choice and calculates the energy consumption of each device. This needs to happen in the following order:

1. Allow the timeshiftables and EVs to make a choice.
2. Calculate the energy consumptions of the timeshiftables.
3. Calculate the energy consumptions of the EVs.
4. Allow the battery to make a choice.
5. Calculate the energy consumptions of the batteries.

Design choices

As explained when discussing the *EV* model, the amount of energy each EV consumes is calculated by the *MainLoop* template. This is done by subtracting the combined energy consumption from the fill limit, which is set by the *EV* template. The combined energy consumption takes the EVs, for which the energy consumption has already been calculated, into account. Therefore, the order in which the energy consumption of the EVs is calculated matters. The desired order is that EVs that have chosen their fill limit first, also have their energy consumption calculated first. This order is chosen because the EVs that choose their fill limit at a later time, take into account that the EVs, that had already chosen a fill limit, would be consuming energy. When this order is not maintained, the EVs that have been scheduled later can consume more energy than intended, thereby reaching their charging target earlier than intended. The EVs that have been scheduled first would suffer because of the other EVs

raising the combined energy consumption. By doing so the EVs that were scheduled earlier cannot charge the amount of energy they expected, resulting in them charging more at the end to compensate, resulting in peaks.

The energy consumption of the batteries is also calculated by the *MainLoop* template. There is only ever one battery inside of the system due to the choices made when designing the *Battery* template. Therefore, the calculation of the energy consumption of the battery consists mostly of subtracting the combined energy consumptions from the fill limit. This calculation needs to happen after the energy consumption of both the EVs and timeshiftables has been calculated, such that the valley-filling approach gives the best results.

Uppaal specifics

The automaton of the *MainLoop* template is given in Figure 5.7. The entire automaton is a loop that is executed every time a signal is emitted on the *step* channel. The signal is emitted by the *Time* template at the end of every time unit. The *HouseBatteryCosts* is immediately reset, such that it does not influence any choices. The first step of the automata is to emit a signal on the *tsStep* channel. This signals to the EVs and timeshiftables that it is their turn to make a choice.

After emitting on the *tsStep* channel, the template is in an urgent location. The urgent location allows all of the committed locations of other templates, namely, the committed locations of the EVs and timeshiftables that are making a choice, to finish before continuing. Once there are no more committed locations, a transition is taken. This transition first calls two functions that are executed sequentially, namely, *alterTsCosts* and *calculateFillValues*.

AlterTsCosts The *alterTsCosts* function is used to calculate how much energy each timeshiftable consumes, its pseudocode can be found in Algorithm 2. The algorithm relies on the *startTimes* set by the *TimeShiftable* template instances. Timeshiftables indicate they are inactive by setting the *startTime* to minus one. For each timeshiftable that has a *startTime* that is not minus one, the function calculates how much energy the timeshiftable is consuming. Calculating the consumption is accomplished by subtracting the *startTime* from the current *time* and using this value as an index for the consumption list. It is also checked whether this index is in fact part of the consumption list. When the index is not a part of the

Algorithm 2 AlterTsCosts

```
for ts in TimeShiftables do
    if ts.startTime != -1 then
        index = time-ts.startTime
        if ts.startTime == time+ts.consumptionCostList.length then
            ts.startTime = -1
        else
            ts.consumption = ts.consumptionCostList[index]
```

consumption list the *startTime* is set to minus one, indicating that the timeshiftable is no longer active.

CalculateFillValues Calculating the energy consumptions for EVs is done according to the pseudocode given in Algorithm 3. It starts by sorting the EVs by the start time of their current interval, as explained in the design choices. Using this order, the consumptions are calculated one by one, accounting for the consumptions calculated for the EVs that started earlier. First the desired energy consumption is calculated according to the fill limit, which is then constrained between zero and the maximum charge rate. It is checked whether using the desired energy consumption leads to the required charge no longer being reachable, since this would go against the constraints of the system. In case the desired consumption does not conform to the constraints of the system, the energy consumption is calculated by taking the remaining energy needed and dividing it by the remaining time. When the desired value does conform to the constraints of the system, the minimum value between the desired consumption and the remaining energy is taken, ensuring that no more energy is consumed than is required. This ensures that all EVs conform to the constraints of the system and that no more energy is consumed than needed. This method of calculating can create inefficient schedules, but that simply means a suboptimal fill limit was chosen, which the optimization algorithm will then filter out.

Once both the *alterTSCosts* and *calculateFillValues* functions have been executed, a signal is emitted on the *batteryStep* channel. This signal allows the batteries to make a choice. After emitting the signal, the template is in another urgent location. This urgent location allows the committed locations of the battery to finish executing. When all batteries have completed their choice, a transition back to the starting location is taken. This transition executes the *calculateBatteryFillValues* function,

Algorithm 3 CalculateFillValues

```
EVs = sortEVs(ElectricalVehicles)          ▷ Sorts the EVs by start time
for EV in EVs do
    costs = staticCost+timeshiftableCosts+evCosts
    desiredRate = max(ev.fillLimit-costs, 0)
    maxDesiredRate = min(ev.maximumChargeRate, desiredRate)
    rC = ev.requiredCharge
    cC = ev.currentCharge
    resultingRemainingCharge = (rC-cC-maxDesiredRate)
    maxAvailable = ((ev.intervalEnd-time)-1)*ev.maximumChargeRate
    if resultingRemainingCharge <= maxAvailable then
        ev.fillValue = min(maxDesiredRate, rC-cC)
    else
        ev.fillValue = (rC-cC)/(ev.intervalEnd-time)
```

after which the combined energy consumption of all devices is stored in the global *latestCost* variable. The *latestCost* variable is used by the *Battery* template to make its decision. After the transition, the template is in the starting location, waiting until a new signal is emitted on the *step* channel.

CalculateBatteryFillValue

Calculating the energy consumption of the battery is accomplished by using the fill limit set by the battery, the pseudocode of which is given in Algorithm 4. The Algorithm refers to the energy consumption as the *fillValue*. First it is checked whether the fill limit is equal to -2147483648, which is the lowest value an integer in Uppaal can take, hence it is used to indicate no action should be taken. When this is not true, the energy consumption needs to be calculated. This is calculated by subtracting the combined energy consumption from the fill limit. The energy consumption is then constraint such that the schedule will conform to the constraints of the system. The first step is ensuring that the energy consumption does not cause the SoC to exceed the capacity or go below zero. The second and final step is to ensure that the battery is not charging more than the maximum charging rate.

The algorithm works because of the choices made when designing the battery template. Due to the battery only being able to improve the quadratic cost and the battery being forced to stop once the battery's (dis)charging rate reached zero, a lot of information is known based on the difference between the fill limit and the combined costs. Without these restrictions it is required to keep a value indicating whether the

Algorithm 4 CalculateBatteryFillValue

```
if batteryFillLimit != -2147483648 then
    SoC+=batteryFillValue;
    costs = staticCosts+timeshiftableCosts+evCosts
    diff = batteryFillLimit-costs
    if diff >0 then
        batteryFillValue = min(diff, batteryCapacity-SoC))
        batteryFillValue = min(batteryFillValue, maxBatteryRate)
    else
        batteryFillValue = max(diff, -SoC)
        batteryFillValue = max(batteryFillValue, -maxBatteryRate)
```

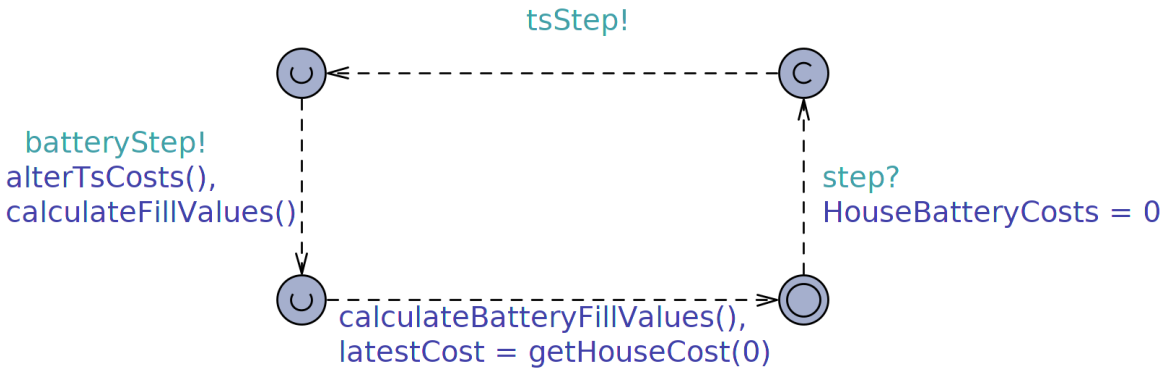

Figure 5.7: MainLoop Template

battery is charging or discharging in order to calculate the correct energy consumption. Most importantly, it is known that, if the difference is less than zero the battery must be discharging, whereas a difference greater than zero means the battery must be charging. Because of this, it is not needed to explicitly keep track of whether the battery is charging or discharging.

5.6 Online algorithm

The offline algorithm can only be used to create schedules for small ALPG situations and for a small number of days. To make the algorithm scale well with time, it had to be changed from offline to online, as explained in Section 2.9. This was not fully possible within Uppaal. Therefore, a Python program has been created that can be found in Appendix C.

The most challenging task for the Python program is to store the state of the model and create a new model that starts in the stored state. This problem is aggravated by multiple instances being created of one template, each with its own state. To be able to create a new

Algorithm 5 Online Algorithm

```
parsed = parse(model)
parsed = duplicateTemplates(parsed)
parsed.save(newModel)
for day in days do
    state=runModel(newModel, start=day, end=day+2)
    setModel(newModel, parseState(state, time=day+1))
    storeFirstDay(state)
```

model that contains the state stored from the previous iteration, each template can have only one instance. Due to the amount of instances of one template being determined by variables obtained from the ALPG data, these templates have to be duplicated by the Python code.

Besides duplicating the templates, the Python code also has to be able to change every variable to its new value. This quickly led to the conclusion that the ideal solution is to parse the entire model, modify the Python object and then write the resulting model into a new file. There exists an open-source parser for Uppaal [36], which is written in *C*. Due to a lack of programming experience with *C* and having done all other programming in Python we decided to write our own parser using the *pyPEG* library [39]. The library was chosen for its simplicity and high amount of customizability. The created grammar is not complete, however, it can parse everything that is required for the model. The grammar can be found in Appendix C.

Using the created parser, the rest of the algorithm becomes quite straightforward. The pseudocode can be found in Algorithm 5. The first step is parsing the Uppaal model using the created grammar. The parsed model is then used to duplicate the templates that are instantiated more than once, the result is saved to a new file. After that, the algorithm loops over the days that need to be addressed in the optimization. In each iteration the schedule for two days ahead is calculated, the result being the state information for both days. All of the variables in the model are set to the values corresponding to the state after one day. This results in the model continuing as if one day has passed the next time it is run. While executing the loop, all of the state information for the first day is stored, which, when combined, results in the state information over all days.

The pseudocode from Algorithm 5 leaves out many small details in favour of simplicity. Most are insignificant, although there is one detail regarding the execution of the model that requires some clarification.

The model can either be executed by running the Stratego algorithm for optimization or by running several simulations. Not using the optimization algorithm is much faster, how much depends on the specific parameters, but gives worse results as will be shown in Chapter 7. Ideally, when running the optimization algorithm there will only remain one possible simulation, however, this is not always the case. In order to ensure that the solution obtained by using the optimization algorithm is as good as possible, the simulation using the control strategy is run several times and the most cost efficient run is chosen. Due to, in both cases, the simulation query being run multiple times, the simulation duration has to be the full duration, rather than until the end of the first day, as is done in [34]. This is an important step, otherwise the algorithm would not take the next day into account, which could result in an inefficient schedule.

Although the algorithm works well, it is not perfect. The problem is that, when Uppaal runs simulations, large numbers are rounded to six figure numbers. This is hardcoded in the implementation of Uppaal. This has several consequences for the implementation. Most notably, all comparisons to the lowest possible value, which is -2147483648, had to be changed to instead compare to -2147480000. This is the only fixable problem. When a fill limit is selected above one million, there is no possibility of obtaining the exact value due to the rounding. The effect of the cost going above one million can be somewhat mitigated by setting it to zero after each interval, which has no effect on the resulting schedule.

5.7 Compositional algorithm

The online algorithm solves the problem of scalability over time, however, the online algorithm does not scale well with the number of devices. A way to overcome this is to change the algorithm into a compositional algorithm, as explained in Section 2.10. This requires the schedule of a device to be optimized by itself and then have other devices optimize themselves, whilst accounting for the devices that have been scheduled before them. To achieve this, a template instance first needs to be optimized. The template then needs to be altered, such that the template will always follow the optimized schedule. This process is hereafter referred to as making the instance *static*. Similar to the online algorithm, this requires each template to have only one instance, since each instance can have a different optimal schedule.

Making an instance static is challenging due to the limited amount

of information that is available. It would have been much easier if the strategies could be extracted, which would dictate how the model should be changed. Making it even more challenging is the fact that not all execution orders can be perfectly controlled, as explained in Section 4.3. Fortunately, due to the way the model is designed this poses as few problems as possible, as most transitions from states that are committed at the same moment can be executed in any order. The only exception to this rule lies in the EV template, because the method that the EVs use to calculate the fill limit depends on the fill limits set by other templates. This problem is mostly fixed by the method in which the template is made static, which is described later.

Due to each template being different, how they are changed from dynamic to static is also different. For each of the three devices templates, namely timeshiftables, EVs and batteries, it is described below how the model is changed. After explaining how each of the device templates is made static, an explanation of how the algorithm is implemented is given.

Finally, it is explained how the battery template has changed with respect to the offline algorithm as described in Section 5.5.

5.7.1 Static timeshiftables

To change the timeshiftable template from controllable to static, the *startTime* variable needs to be set correctly. The controllable timeshiftable template is given in Figure 5.4 and an example static timeshiftable template is given in Figure 5.8. As can be seen, there are no controllable transitions in the static template. When controllable, the *startTime* variable is set when a transition is made from the *Waiting* location to the *Activated* location. This behaviour remains the same for the static template. To make the template static, all existing paths between the *Waiting* location and the *Activated* location are replaced by a set of transitions. One transition is required for each time the timeshiftable needs to activate. Each transition synchronizes on the *tsStep* channel and sets the *startTime* variable. The difference between the transitions is the guard, which ensures that the transition is taken at the intended time. It is not allowed by Uppaal to merge the transitions by chaining all of the guards using *or* expressions, as explained in Section 2.5.2, thus multiple transitions are required.

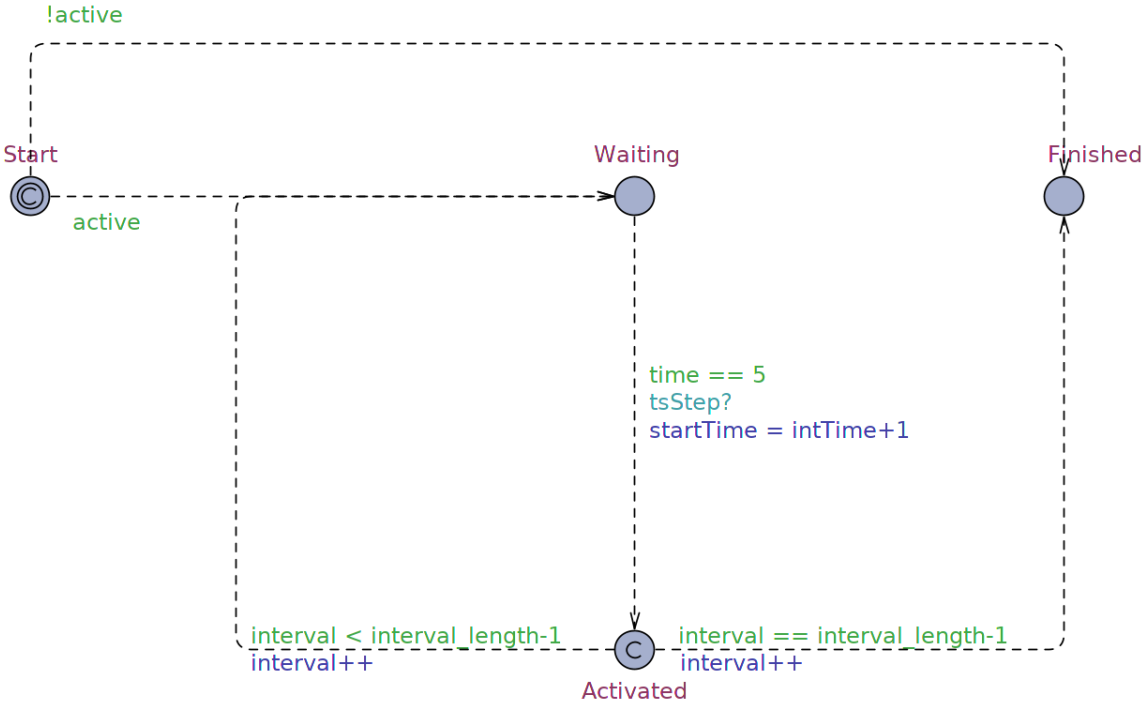

Figure 5.8: Example Static Timeshiftable Template

5.7.2 Static EVs

For EVs the variable of importance is the *fillLimit* variable. The controllable EV template is given in Figure 5.5 and an example static EV template is given in Figure 5.9. The only change that is required is to replace the controllable transition from the *Choice* location to the *Working* location with a set of uncontrollable transitions. Similar to the static timeshiftable template, the set of transitions consists of one transition for every time the controllable model switched to the *Working* location. Each transition has a guard that checks the time and, when taken, sets the fillLimit to the correct value and increments the interval counter.

5.7.3 Static Batteries

For batteries the variable that needs to be set is the *batteryFillLimit*. The controllable battery template is given in Figure 5.6. An example of a static battery template is given in Figure 5.10. The battery template is more challenging to turn from controllable to static. This challenge arises due to how switching from the *Working* location to the *Inactive* location is handled. This switch is never controllable, therefore, it could remain the same in the static template. However, doing so could result in the template transitioning from the *Working* location to the *Inactive*

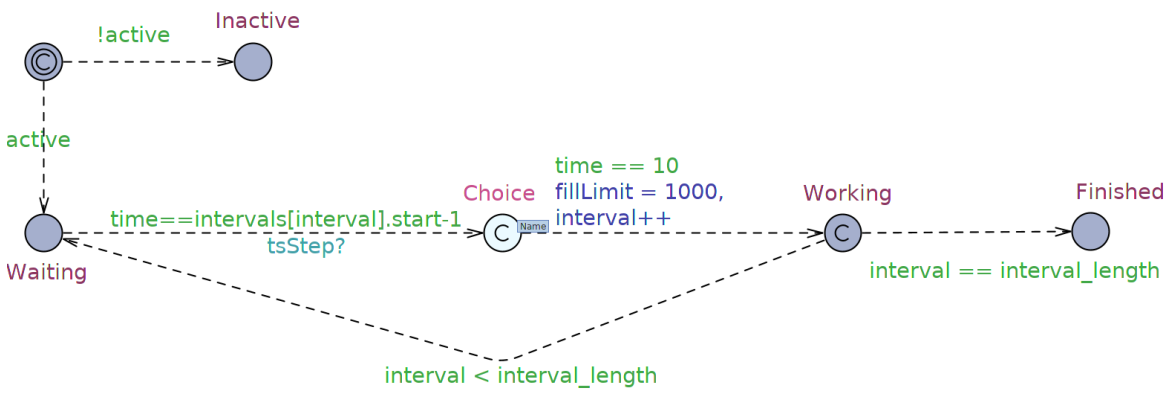

Figure 5.9: Example Static EV Template

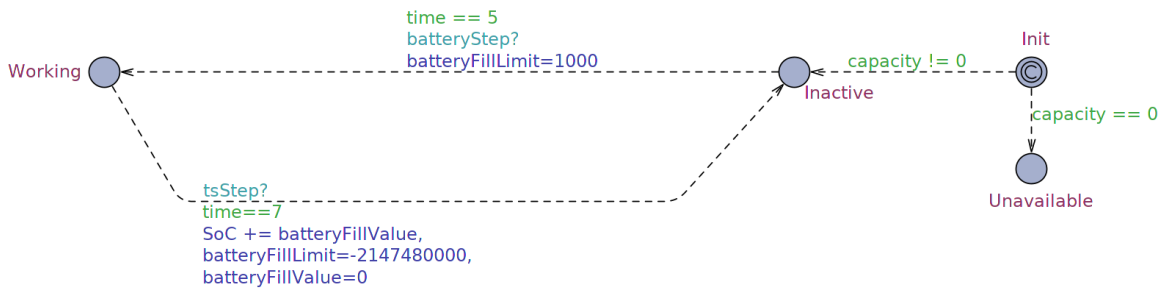

Figure 5.10: Example Static Battery Template

location at a different time then intended, which could result in worse performance. This is possible due to the behaviour of the other templates changing. This problem is especially prevalent in case a switch to the *Inactive* location occurs after the next *batteryFillLimit* should already have been set. To prevent this behaviour, the transitions to the *Inactive* location have been changed to be based on time. Due to how the amount of energy consumed at each point in time is calculated, it is still impossible for the SoC to exceed the battery capacity or to drop below zero. All the template does is transition between the *Inactive* and *Working* locations based on the time, setting the required values to their determined values.

5.7.4 Algorithm

Using the methods for making templates static, we can make a compositional algorithm. The pseudocode for the algorithm can be found in Algorithm 6. The first step is to disable every device, which means having an Uppaal system definition consisting of only the *Time*, *MainLoop* and *CostCalculator* templates. The second step is to group the devices into component groups, which are devices that are optimized together. The simplest case would be to give every device its own component group,

however, the results are better when multiple devices are optimized simultaneously. Once the component groups have been chosen, it is time for the main section of the algorithm, which consists of two nested loops. The outer loop is used to iterate over all of the devices multiple times, thereby improving the outcome. The inner loop iterates over all the component groups. The first step inside the loops is to enable all devices in the component group and then make the devices controllable, which means restoring the template to its original state and adding it to the system definition. The model is then used to optimize the cost. When the algorithm is still in its first iteration, or when the result is more optimized than the *bestResult*, the result is stored as the *bestResult*. Each device in the component group is then made static based on the *bestResult*. This results in the devices always being made static based on the best schedule that has been found, preventing the result from getting worse after the first iteration. Once both loops have completed, the model is simulated once more to get the final result. The model is completely static at this point, hence, there is no need for the optimization algorithm.

During this process, the order in which the component groups are optimized influences the schedule that is created. The order has been chosen such that it is equal to the order in which the *MainLoop* template calculates the energy consumptions. Therefore, the timeshiftables are optimized first, then the EVs and then the batteries. This is done because the schedules of the EVs take the timeshiftables into account and the schedules of the batteries take both the timeshiftables and EVs into account.

5.7.5 Battery template

Due to the improved scalability with the number of devices, there is no longer a need to combine the batteries into one battery, as was done for the offline algorithm described in Section 5.5. We decided it was better to have each battery as a separate template, because this gives the actual battery schedule and cannot create infeasible schedules. The downside is that scheduling one large battery is much faster, therefore it does slow down the optimization.

The only change to the battery template needed was to make the variables references to arrays, rather than accessing the global variables directly. The *calculateBatteryFillValues* function inside the *MainLoop* template had to be changed to calculate one *fillValue* for each battery. This also required the batteries to be sorted by their *fillLimit* before

Algorithm 6 Compositional Algorithm

```
parsed = parse(baseModel)
parsed = duplicateTemplates(parsed)
parsed.save(model)
for device in devices do
    devices.disable()
bestResult = None
componentGroups = createGroups(devices)
for i in range(0, num_iterations) do
    for componentGroup in componentGroups do
        for device in componentGroup do
            device.enable()
            device.makeControllable()
        result = model.run(optimize=True)
        if i==0 or result.cost <bestResult.cost then
            bestResult = result
        for device in componentGroup do
            device.makeStatic(bestResult)
return model.run(optimize=False)
```

calculating their energy consumption, which ensures the profile stays as flat as possible.

5.8 Online Composition algorithm

Both the online algorithm and the compositional algorithm fix one scalability issue. These algorithms can be combined to have good scalability with both time and the amount of devices. The combined Online Compositional algorithm is given in Algorithm 7. The merging of the algorithms is accomplished by replacing the optimization step of the online algorithm with the compositional algorithm. This means that, in every iteration of the online algorithm, the compositional algorithm is executed. The compositional algorithm is used to optimize the time slice given by the online algorithm. The result of the compositional algorithm is then used as the schedule for said time slice, after which the next time slice is optimized, again by using the compositional algorithm.

Theoretically, the complexity of this algorithm could be reduced to $O(n*t)$, in which n is the number of devices and t is the number of days to be scheduled. This is due to the online part of the algorithm scaling linearly with time and the compositional part of the algorithm scaling linearly with the number of devices. However, the current implementation slows down when more devices are added and the duration to calculate becomes

longer. These implementation faults are more thoroughly discussed in Section 5.10.

Algorithm 7 Online-Compositional Algorithm

```
parsed = parse(baseModel)
parsed = duplicateTemplates(parsed)
parsed.save(model)
for day in days do
    for device in devices do
        devices.disable()
    bestResult = None
    componentGroups = createGroups(devices)
    for i in range(0, num_iterations) do
        for componentGroup in componentGroups do
            for device in componentGroup do
                device.enable()
                device.makeControllable()
            result = model.run(start=day, end=day+2, optimize=True)
            if i==0 or result.cost <bestResult.cost then
                bestResult = result
            for device in componentGroup do
                device.makeStatic(bestResult)
    result=model.run(start=day, end=day+2, optimize=False)
    setModel(model, parseResult(result, time=day+1))
    storeFirstDay(result)
```

5.9 Uppaal Cora

As mentioned in Section 2.7.1, Uppaal Cora implements linearly-priced automata and has an algorithm that minimizes the price. This is identical to how this research makes use of the optimization algorithm of Uppaal Stratego. Our first attempts to use timed automata for creating schedules were carried out using Uppaal Cora, however, we quickly ran into problems with the amount of devices that could be scheduled at once. The reason the amount of devices that can be scheduled is very limited is that, in order to find the optimal schedule, Uppaal Cora explores every possible trace. Due to the problem being NP-hard, the amount of possible traces increases exponentially with time and the amount of devices. This problem is magnified by the static energy cost being piecewise-linear, which is explained in Section 5.2. This led us to look elsewhere, namely Uppaal SMC and Uppaal Stratego.

Uppaal Stratego saw a similar problem with scalability, although to

a lesser extent, which was solved by creating an online compositional algorithm. This prompted a second look at Uppaal Cora. During the development of the Uppaal Stratego model, several improvements were made. Therefore, the same models used inside Uppaal Stratego were converted to Uppaal Cora, which required almost no changes. With these new models the amount of devices that can be scheduled has improved, however, it is still more limited than Uppaal Stratego. The major problem occurred when calculating the battery schedule. Even with one battery, using all of the restrictions mentioned in Section 5.5, Uppaal Cora could not find a schedule before running out of memory. Ignoring the fact that the amount of memory runs out, it would never be feasible to calculate a schedule for two days ahead, which is the least required by the online algorithm. To indicate how poorly it scales, when calculating for twelve hours ahead it takes two seconds, calculating twelve and a half hours takes sixteen seconds and thirteen hours gives an out of memory error.

5.10 Discussion

The model that has been created is able to generate good schedules. However, there are several downsides. The most apparent downside is that clock variables are underutilised, which are usually the strength of TAs. Without these clock variables the model is a finite automaton. Our model makes so little use of clocks that it would be possible to remove them completely, turning it into a finite automaton. Using finite automata instead of TAs has not been attempted during this research and is more thoroughly discussed in Chapter 9. Due to the piecewise linearity described in Section 5.2, it is challenging to make active use of clocks. One method by which clocks could find more usage, would be to calculate the quadratic energy consumption at the end, rather than continuously over time. However, this adds the downside of needing to keep track of when devices became active and which values they chose. This is especially problematic for timeshiftables, which would need to store the time they activated as an integer. In Uppaal Cora this is not even possible, since there is no concept of doubles. In Uppaal SMC and Stratego the *fint* function, which is supposed to convert a double into an integer, has not been implemented as explained in Section 4.1. Therefore, it is challenging to convert a double or clock into an integer. It is possible to convert a clock to an integer, due to a comparison between a double and an integer being possible, however, it is computationally expensive.

There are also several areas that can be improved in the current implementation of the online compositional algorithm, which can reduce the calculation time of the algorithm significantly. Currently the algorithm slows down when there are more devices to be optimized. This is due to the static devices, that have already been scheduled, still existing inside of the system and requiring computations. This problem could be mitigated by merging the static templates of each type. This one static template would then set the required values for all devices of its type that have already been scheduled. This can be further improved by adding the energy consumption of a device to the uncontrollable energy consumption and removing the template completely. However, this has the downside that the energy consumption of EVs and batteries is no longer calculated at each time-unit. The recalculation of the energy consumption could be helpful when changing the starting time of a timeshiftable should alter the consumption of an EV.

When calculating over a long duration, the array with uncontrollable energy consumption becomes very large. This slows down the simulations and thereby the optimizations. A solution is to only store the uncontrollable consumptions that are needed during the current interval of the online algorithm. Replacing the array of uncontrollable consumptions after every interval of the online algorithm. An offset could then be used to obtain the correct indices.

Another improvement is integrating Uppaal Cora into the online compositional algorithm. Although Uppaal Cora cannot create schedules for batteries, it can create schedules for both timeshiftables and EVs, doing so more efficiently than Uppaal Stratego. The difference is that Uppaal Stratego can give an approximation of a perfect solution, which means that Uppaal Stratego can schedule much more at the same time. Nevertheless, when running the online compositional algorithm, it is possible to use Uppaal Cora when calculating schedules for timeshiftables and EVs. This could drastically improve the speed at which the algorithm works and would increase the consistency of the output.

Due to a limited amount of time we were unfortunately unable to implement any of these additions. However, we strongly believe that these changes would greatly improve the speed of the algorithm and potentially improve the schedules that are created.

Chapter 6

Experimental set-up

To answer the research questions, as listed in Chapter 1, several tests have been executed. There are three main questions related to testing that need to be answered:

1. How do the schedules created using Timed Automata (TAs) compare to the schedules created by the Decentralized Energy Management Kit (DEMKit)?

2. How well does our algorithm scale?

   (a) How many devices can be scheduled?

   (b) What amount of time can be scheduled?

   (c) Can the computation time be reduced by sacrificing accuracy?

3. What are strengths and weaknesses of using timed automata for solving the Demand Side Management (DSM) scheduling problem?

By testing multiple scenarios, we believe that we can sufficiently answer all of these questions.

In the research questions, TAs are a concept rather than an implementation. During this research we used several versions of Uppaal, each of which implements TAs. To properly answer the research questions, the different versions of Uppaal need to be compared.

The rest of this chapter details how the tests are executed, the design choices and their motivation. Section 6.1 describes the requirements for the tests, after which Section 6.2 describes the testing methodology. Finally, Section 6.3 details how the resulting schedules are validated. Descriptions of the scenarios to be tested are given in Chapter 7, combined with the results of each scenario.

6.1 Requirements

There are several requirements for selecting what tests are to be performed and how they are performed. The tests are split into two main categories, namely, testing the performance of the different versions of Uppaal and testing the difference between Uppaal and DEMKit. For all tests there are three main requirements regarding how they are performed, as follows:

- Fairness and validity.
- Consistency.
- Reproducibility.

Each of these requirements is detailed below.

6.1.1 Fairness and validity

The most important requirement is fairness, which is especially pressing for the comparison between Uppaal and DEMKit. For a comparison between the different Uppaal schedule generation methods fairness is obtained by using nearly identical models, thereby ensuring that the schedules adhere to the same constraints. For the comparison between DEMKit and Uppaal there could be differences in the constraints, due to implementation faults or a different interpretation of the Artificial Load Profile Generator (ALPG) scenario. Therefore, it is important to validate both schedules to ensure they comply with the same set of constraints.

Fairness in the tests is also obtained by using the randomly generated data from the ALPG. The only manually specified part is the ALPG configuration. When creating the configuration, the key settings are the number of houses, length of the schedule and the types of houses. Adjusting these values should give diverse enough scenarios to draw conclusions. The settings for the devices can also be changed slightly, although it is important that the values are still close to realistic. When using unrealistic values, such as an overly large battery, the choices for all other devices will become trivial in comparison. Because changing the device specific settings has only a minor effect on the data, these settings remain at their default values for most tests.

6.1.2 Consistency

When running multiple tests, it is also important that the testing methodology is consistent. This is needed to ensure that the results are also consistent, such that proper conclusions can be drawn. How the consistency is obtained is explained in Section 6.2.

6.1.3 Reproducibility

Reproducibility is important in case anyone wants to reproduce a specific testing scenario. To have the exact same results, the same version of DEMKit and its configuration are required. We cannot yet publish these due to DEMKit not being publicly available.

In order for the Uppaal tests to be reproducible, the model, query, ALPG scenario, random seed and command line arguments need to be stored. Having those five things should result in the exact same schedule. There is no guarantee that the computational time will be the same though, due to this being influenced by many other factors, such as the hardware.

6.2 Methodology

This section describes the methodology behind the testing. The section has been split into two parts, both with different methodologies due to the scope of the comparisons. The first part describes the methodology when doing a comparison between the different Uppaal versions, whereas the second part describe the methodology for comparing the online compositional algorithm against DEMKit.

6.2.1 Uppaal versions

During our research three different versions of Uppaal were used, each using a different method to generate schedules. The goal is to show the strengths and weaknesses of each of these versions with respect to our DSM application. These strengths and weaknesses are quite apparent, therefore we deemed it unnecessary to test multiple scenarios, especially because the models used are nearly identical between all versions. The only differences between the models are the ones necessary due to the restrictions that exist with Uppaal Cora.

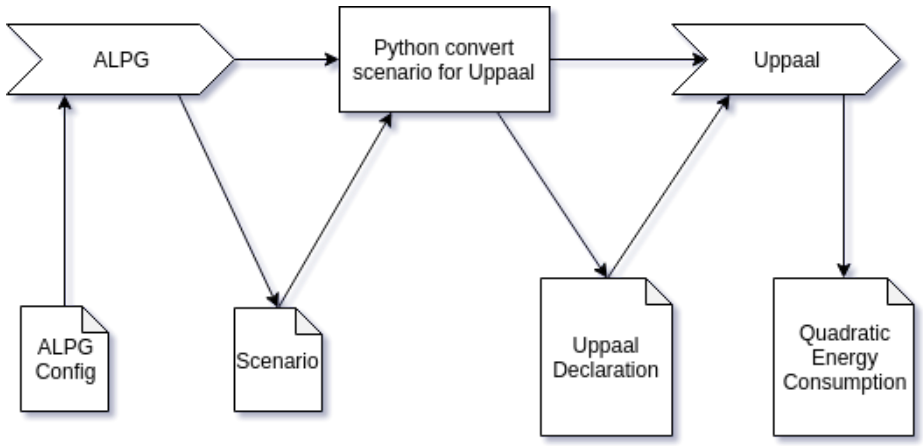

Figure 6.1: Test setup for testing between different versions of Uppaal.

Uppaal Cora has not been implemented as an online compositional algorithm. Therefore, the comparison between the different Uppaal versions needs to be done by having each version calculate the schedule directly, without any external program splitting the task. This limits the size of the testing scenarios. The online compositional algorithm splits the complete scenario into many small scenarios, which are then scheduled using Uppaal. Therefore, the schedules created for small scenarios determine the output of the online compositional algorithm. Although the strength and weaknesses with respect to the Uppaal model can differ between the offline and online compositional variant, these tests can expose some of the strength and weaknesses of using Uppaal for DSM scheduling.

Due to the limited testing that is possible and needed, we decided not to automate this process, instead, we carefully documented the tests by hand. The tool chain that is used to compare the different version of Uppaal is given in Figure 6.1. Although not the entire test is automated, rewriting the ALPG data into a format usable by Uppaal, as explained in Section 5.1, is automated. The declaration created as a result of this rewrite needs one small tweak to be compatible with Uppaal Cora, namely, the removal of the *cost* variable definition. This is needed since Uppaal Cora has an implicit *cost* variable.

6.2.2 Uppaal and DEMKit

The main focus of the tests is to compare the online compositional algorithm against DEMKit. These tests show how well the algorithm currently works and the potential that timed automata have to model DSM sce-

narios. All of the schedules need to be validated for compliance with the constraints set by the ALPG data, this goes for both the schedules generated by our algorithm as well as DEMKit. The tests need to range from small to large, short to long and with different types of devices, in order to get an accurate understanding of the differences between the algorithms.

For the online compositional algorithm, only the variant that makes use of the Stratego optimization algorithm is compared against DEMKit. Running several simulations instead of using the optimization algorithm should be faster. This is due to both methods running simulation, however, when not using the reinforcement learning algorithm the amount of simulations will give diminishing returns. In contrast, the reinforcement learning algorithm will give a bad strategy when too few simulations are run. Therefore, running simulations without the optimization algorithm works well for scheduling problems with few choices, whereas the optimization algorithm will keep improving the more simulations it gets. Running simulations without the optimization algorithm results in worse schedules and inserts additional randomness to the algorithm, thereby resulting in inconsistencies between tests. It is possible to specify a seed, as described in Section 4.5. However, because each scenario changes the model and thereby the behaviour of the random number generator, the inconsistencies between tests remain. Therefore, we decide not to include the results of running simulations.

The entire testing process has been automated. This is done to ensure that the results are consistent and to ensure the validation of the schedules is handled properly. The entire testing procedure is displayed in Figure 6.2. Each test only requires an ALPG configuration file, based on which the scenario is generated. Based on the generated scenario, the schedules are optimized, the schedules are validated and finally the outcome is written to a file. The validation process is described in Section 6.3 and ensures that both schedules are validated in the same manner. The result of each test contains the following:

- The parameters needed to recreate the test.
- The computation time of each test.
- The cost of each schedule.

The cost is calculated after the validation is complete using the formula given in Equation 6.1, ensuring the calculation happens in exactly the same fashion for both schedules.

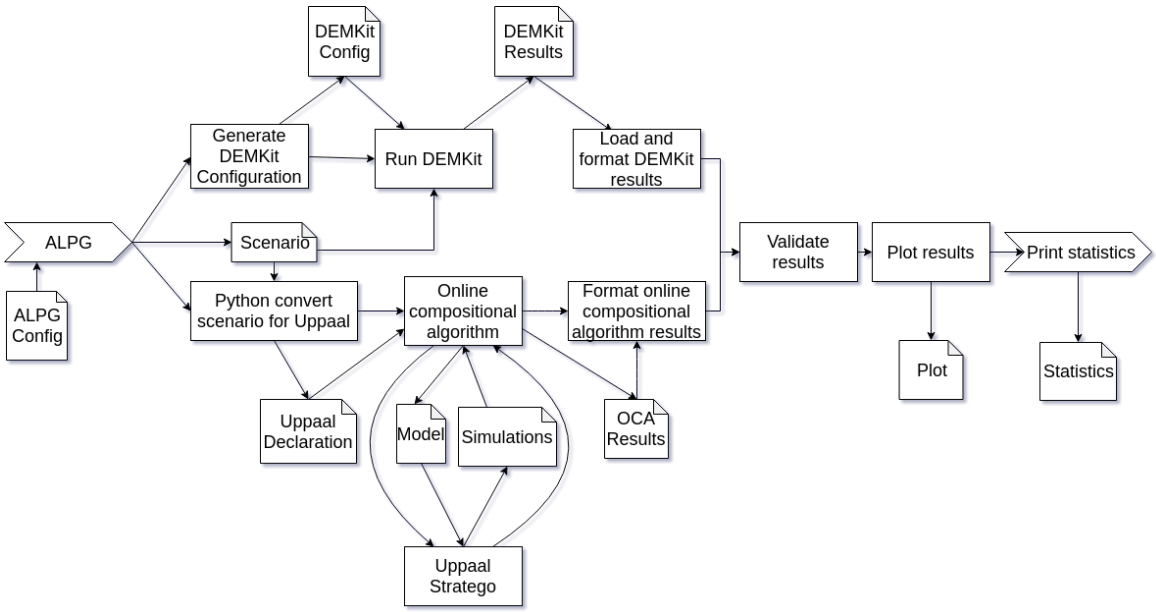

Figure 6.2: Test setup for testing between DEMKit and Uppaal.

$$cost = \sum_{i=start}^{end} load_i{}^2 \tag{6.1}$$

Uppaal

There are several parameters that influence the speed and performance of the Stratego algorithm. The reasoning behind the *learning filter* and *learning method* were described in Section 4.6.6. The parameter for the resolution, as explained in Section 4.4, is set to $duration + 1$, which gives exactly one data point at each time-unit. Because the state of the system can only change at the end of each time-unit, this will give perfect precision.

The only remaining settings are the learning parameters, which determine how many simulations the optimization algorithm is allowed to run in order to find the strategy. These parameters are as follows:

- The number of successful runs.
- The maximum number of runs.
- The number of good runs.
- The number of runs to be evaluated.

The effect of each parameter is explained in Section 4.6.6. It is difficult to determine optimal parameter values, due to the randomness involved

	Successful	Max	Good	Eval
Timeshiftables	5	5	1	1
EVs	3	3	1	1
Batteries	50	50	1	1

Table 6.1: Parameters for the Stratego online optimization

and because each scenario could have different optimal values. More determinism in the resulting strategy is better and, due to the reasons given in Section 4.6.6, this led to the use of the value *one* for both the number of good runs and the number of runs to be evaluated. To decide the values of the number of successful runs and the maximum number of runs, we tested with several large scenarios and tried to find at which point there was very little improvement when allowing more simulations. The resulting values can be found in Table 6.1. These are all the values when optimizing one device at a time. With these values it is likely that the strategy is good in at least one of two iterations of the compositional part of the algorithm. It can be seen that batteries require many more simulations to find a suitable solution, even with the restrictions mentioned in Section 5.4. Therefore, each battery takes much longer to schedule.

DEMKit

When testing with DEMKit there are many settings that can be altered. However, there are several settings that are the most important for creating a fair comparison. The first setting is that of perfect predictions, meaning that DEMKit knows everything that will happen. This should be set since the Uppaal model assumes perfect predictions as well. Because we want to compare our approach to profile steering, it is also needed to enable profile steering as the optimization algorithm. The optimization should only take active energy consumption into account, as explained in Section 5.1.5. It is also important to optimizes for a neighbourhood, rather than for a single house. The final settings that are important are the timebase and the control timebase, both of which should be set to fifteen minutes, since this is what the model uses as well. All other settings should be quite self-explanatory and simply need to be set such that the outcome matches the ALPG data. Due to the settings consisting of a Python code project, which unfortunately cannot be shared due to copyright protection, it is not feasible to discuss every little detail. Most

settings have been kept at the default, which was recommended by a prominent DEMKit developer.

6.3 Validation

Validation of the schedules is an important aspect to ensure the algorithm is working correctly and to ensure the comparison is done fairly. To ensure that both schedules adhere to the same set of constraints, both schedules have been rewritten into the same format. The validation consists of four parts:

- Timeshiftables
- Electrical Vehicles (EVs)
- Batteries
- Uncontrollable costs

Each part is discussed below. Once the validation has been completed, the quadratic cost of the load, as given in Equation 6.1, is calculated. This measure is then used as the performance metric for the schedule. Calculating the quadratic cost at the end of the validation ensures that the calculation happens in the same fashion for both optimization techniques. This is important because it is one of two values used in the comparison, therefore it needs to be correct in order to achieve a fair comparison. The results of these validation tests are given in Section 6.3.5.

6.3.1 TimeShiftables

The validation of the timeshiftables is done one timeshiftable device at a time. First, it is checked whether the device has been activated in each interval. Next, it is checked whether the activation time and the deactivation time of the device are both within the interval. Finally, it is checked whether the costs for each time unit from the time of activation correspond to the costs as defined in the scenario. This ensures that all constraint on the timeshiftable are met, assuming that the device is not interruptible, which is also assumed within the Uppaal model and within DEMKit.

6.3.2 EVs

For each EV it is checked whether the sum of the energy consumption, within each interval, is equal to the required amount of energy for said

interval. It is also checked whether the energy consumption outside of the intervals is zero. We made the assumption that the EVs cannot charge beyond its required amount of energy, which is not necessarily true due to the capacity being larger than the required charge. We made this assumption because DEMKit adheres to this assumption, at least when using the default configuration. It also seems unlikely that an EV would charge more energy than required, due to the additional costs involved. We also made the assumption that EVs can only charge electricity and cannot discharge electricity. This assumption is made to limit the complexity and to be consistent with DEMKit. Together, these validations ensure that the EV always reaches its target and that the EV does not charge while not available to be charged.

6.3.3 Batteries

The batteries are validated by checking that the State of Charge (SoC) remains between zero and the maximum battery capacity. It is also checked whether the amount being charged or discharged never exceeds the maximum charging rate. This ensures the battery adheres to the constraints of the scenario. The only assumption made is that the battery has a perfect conversion ratio, which is the assumption made by DEMKit. Implementing a more complex battery model within Uppaal would be possible without any major loss of performance. This is due to Uppaal having to calculate the amount of energy consumed at each time unit. Therefore, adding a conversion loss has little impact on the computation time.

6.3.4 Uncontrollable load

Validating the uncontrollable load could be skipped. Instead, the correct uncontrollable load could be used when calculating the performance. However, we deemed it important to ensure that both optimizations were using the correct values. Therefore, it is checked at each time-unit whether the uncontrollable load is equal to the values given by the ALPG.

6.3.5 Results

Using this validation revealed several bugs and mistakes in DEMKit, the DEMKit configuration and the implementation of our algorithm, as described in Chapter 5. All of these mistakes and bugs have been confirmed and fixed. All results shown in Chapter 7 have passed the validation,

which can be checked by redoing the tests. The only downside is that it is not possible to test whether either of the optimization strategies have constraints that are more restrictive than the constraints tested for by the validation tests. This is challenging because more restrictive constraints result in valid. The reason it is still important to know when there are more restrictive constraints is because the comparison needs to be as fair as possible. When one of the two scheduling algorithms is more restricted, it gives an unfair advantage to the other algorithm. Because it is challenging and very time consuming to create an automated test for this, we manually looked at the schedules created by both algorithms. We observed that both algorithms made use of the lower and upper bounds on intervals and charging rates. Therefore, we assumed both constraints to be equal to the validation test.

Chapter 7

Results

This chapter describes the specific experiments that have been executed and what their results are. The chapter is split into three parts, starting with Section 7.1, in which we compare different Uppaal versions. Then, Section 7.2 details everything about the comparison between the Decentralized Energy Management Kit (DEMKit) and the algorithm developed in this research. Afterwards, Section 7.3 discusses how these tests have answered the questions stated at the start of Chapter 6. Finally, a conclusion is drawn in Section 7.4

7.1 Uppaal comparisons

During this research, three different versions of Uppaal were used. This section shows the scheduling differences between the versions. To explain the differences between the Uppaal versions, and to show their strengths and weaknesses, a testing scenario is scheduled by the different versions of Uppaal. The exact testing scenario can be found in Appendix C. The parameters of this testing scenario can be found in Table 7.1. The testing scenario has been chosen as a rather small and short scenario, such that a schedule can be calculated without the need for splitting the problem into multiple smaller problems. This was a requirement due to Uppaal Cora not having been implemented as an online compositional algorithm.

All of the Uppaal versions use a nearly identical model, although a slight adjustment was needed for Uppaal Cora. This adjustment was required due to guards not being allowed to contain a clock comparison on synchronizing transitions with Uppaal Cora. The behaviour of all models is identical. The Uppaal Cora model could have been improved by moving much of the code directly onto the transitions, rather than inside a function. However, we valued identical behaviour over a slight performance increase.

7.1.1 Evaluation

The results of running the different Uppaal versions on the aforementioned scenario can be found in Table 7.2. Each type of device has been optimized independently to show the strengths and weaknesses and also because running everything at once would have required an even smaller scenario. The *Score* column displays the sum of the quadratic energy consumption, therefore, lower is better. For Uppaal Statistical Model Checker (SMC) and Stratego the *Score* is rounded to six digits, due the rounding done by Uppaal in those versions. It should also be noted that the quadratic cost is rounded down to integers before taking the square. This is due to our model completely making use of integers, because Uppaal Cora does not have *doubles* and the *double* implementation in the other versions reduces the performance. Because all three versions do this, it should not have any impact on the comparison.

The results are interesting due to the difference in results for each device type. For timeshiftable devices it is clear that the Uppaal Cora branch is far superior. Uppaal Stratego can also come up with a good, but not perfect, schedule. However, Uppaal Stratego requires much more time and whether the output is good depends on the randomness. With a large amount of time Uppaal SMC could also come up with a good answer, however, we decided to limit the number of simulations such that the time is close to the time it took Uppaal Stratego. The result is that Uppaal SMC is unable to find a good schedule, let alone a perfect schedule.

For Electrical Vehicles (EVs) it is a completely different story. The results of all three Uppaal versions are very close to each other. Uppaal SMC has the advantage that it can potentially find a very good solution in very little time. At the same time, Uppaal SMC has the potential to find nothing for a long duration. The seed we had chosen was actually able to find the same solution in 0.3s, however, this was a rare coincidence, therefore we decided to change the time such that it will have roughly a 50% chance of finding a similar solution. Uppaal Cora is still the most reliable, due to always giving the perfect solution, and does not take any longer than the other Uppaal branches. The reason the solution is always perfect is due to its exhaustive evaluation of the model, as explained in Section 2.7.1.

The battery results are, again, completely different. Uppaal Cora cannot calculate a schedule due to running out of memory. However, even with unlimited memory it would still not have found a schedule

Parameter	Value
Duration(days)	2
Houses	2
TimeShiftables	4
TS intervals	5
EVS	2
EV intervals	2
Batteries	1

Table 7.1: Test scenario description.

Device type	Uppaal	Time	Score	Notes
TimeShiftables	Cora	0.093s	1859146	
	SMC	12.329s	2009120	
	Stratego	11.334s	1895040	Another seed resulted in 2027230
EVs	Cora	2.269s	5131128	
	SMC	2.387s	5131610	
	Stratego	2.484s	5131610	
Batteries	Cora	-	-	Memory ran out after about half a day
	SMC	33.749s	1144090	
	Stratego	31.109s	895128	

Table 7.2: Uppaal version comparison.

within a reasonable amount of time. This is because Uppaal already ran out of memory after calculating the schedule for a little over half a day, which is one fourth of the entire period to be scheduled. Calculating the schedule for half a day already took several seconds, combined with the exponential increase in the amount of states, this means it would most likely take years to calculate the entire schedule due to each state needing to be calculated. Uppaal SMC and Stratego are both able to find schedules. Once again, Uppaal SMC has been limited in time, such that it is close to Uppaal Stratego. Uppaal Stratego clearly has a much better schedule than Uppaal SMC, however, neither of them is able to find a good schedule quickly.

7.1.2 Discussion

As mentioned above, Uppaal Cora outperforms both Uppaal SMC and Uppaal Stratego when scheduling EVs and, especially, timeshiftables. At the same time Stratego outperforms Cora when it comes to batteries.

The cause for this difference lies in how both versions optimize the cost. Uppaal Stratego always runs complete simulations, which takes time and results in a lot of duplicate work. Uppaal Cora on the other hand creates the entire state space and traverses all of those states, thereby never calculating the same state twice. Uppaal Cora does not take the *cost* variable into account for its states. Instead the *cost* for a state is set to the lowest *cost* of any path leading to the state. Consequently, all paths but the one with the lowest cost are removed. This makes it so that, when there are no devices inside of an activation interval, there can only be one state due to the only distinguishable difference being the *cost*. This is especially beneficial for timeshiftables, which are only active for a short time. When a timeshiftable is not active, its variables are always reset to their default state, except for the interval counter. An example of the complete state space of a timeshiftable can be found in Figure 7.1. In this example the timeshiftable has three possible starting times. The different paths that can be taken are indicated by using different colours. As displayed, all paths eventually lead to the same state, due to the *cost* variable not being included in the states. This results in having to calculate seventeen states, rather than the twenty-seven states that would be needed when completely following each path. This effect is amplified when there are more states that do not fall inside of the interval, which is the case in a real scenario. This results in a rather small state space. However, the amount of states does increase exponentially when more overlapping intervals are added. This is the complete opposite of Uppaal Stratego, which wants to try as many paths as possible, of which there are often many, each time simulating all states.

When scheduling EVs it can already be seen that Uppaal Cora starts to take much longer than for timeshiftables. This is the opposite of Uppaal Stratego and SMC, for which the required time goes down. This is because, during the entire interval an EV is charging, two values are set, namely the *fillLimit* and *State of Charge (SoC)*. This makes it so that there is very little chance of multiple paths leading to the same state whilst the EV is charging, thereby increasing the amount of states that need to be calculated. Because the time an EV is charging is much longer than the time a timeshiftable is active, there are more states and therefore it takes longer. Uppaal Stratego and SMC do not suffer from this downside due to running simulations. In reality there are only several choices to be made for each EV, namely, thirty in this example. This means all options can be simulated in a reasonable amount of time.

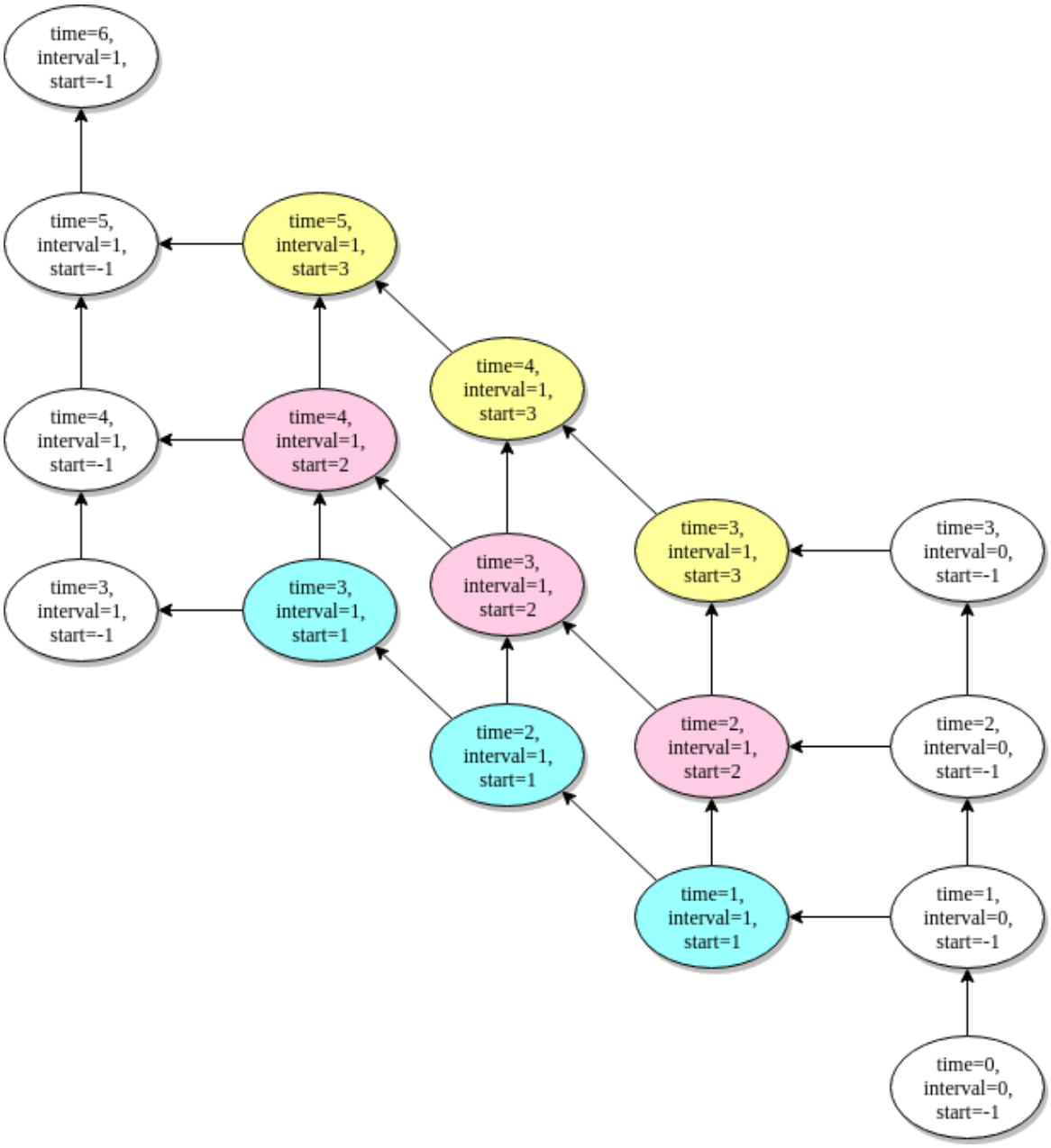

Figure 7.1: Uppaal Cora states for timeshiftable.

Scheduling batteries exacerbates the problem Uppaal Cora has with scheduling EVs. There are multiple variables that keep changing for the entire duration, thereby ensuring that many paths lead to a unique state. EVs are only active for a small period, whereas batteries can switch to active at any point, thereby creating more states than Uppaal Cora can handle. Uppaal Stratego and SMC also struggle to find good solutions due to the number of possible paths. This is where the strength of Uppaal Stratego shines, which can use its reinforcement algorithm to find a decent solution even when having only explored a limited set of possibilities. How good the battery solutions provided by Uppaal Stratego actually are will be shown in Section 7.2.

These results show that there is not a “one size fits all” solution when using Timed Automaton (TA) for the Demand Side Management (DSM) scheduling problem.

We have made the choice to use only Uppaal Stratego in the online composition algorithm, as describe in Section 5.8, due to a limited amount of time. The algorithm could have benefited from using both Uppaal Cora and Uppaal Stratego, however, more about this is presented in Chapter 9.

7.2 Uppaal and DEMKit

To determine how good Uppaal is at scheduling a DSM scenario, we compared it against the state of the art in DSM scheduling, which is currently DEMKit. The comparison is done between our online compositional algorithm, as described in Section 5.8, and DEMKit. This is done because both are able to handle longer schedules and many devices. We have mentioned multiple techniques that can improve our algorithm’s speed. Therefore, it is our expectation that DEMKit generates its schedule much faster. However, the schedules created by both algorithms are expected to be fairly similar. It is anticipated that for batteries Uppaal will perform less than DEMKit, due to the restrictions that were placed on the model, as explained in Section 5.4. Both timeshiftables and EVs are expected to be scheduled similar to their DEMKit counterparts.

In order to compare the two algorithms, several scenarios have been created using the Artificial Load Profile Generator (ALPG). Most tests use a combination of timeshiftables, EVs and batteries to find out how good the schedules are. There are also four tests that focus on one type of device, in order to compare how efficient each device can be scheduled.

#	Days	TS	EVs	Bat	Algorithm	Time	Score	Norm
1	5	12	4	1	DEMKit	53s	3322416	100.0
					Uppaal	1562s	3357305	101.1
2	5	14	4	0	DEMKit	58s	4505242	100.0
					Uppaal	628s	4516096	100.2
3	48	3	1	0	DEMKit	93s	54813029	100.0
					Uppaal	1064s	54127640	98.7
4	3	40	18	6	DEMKit	413s	1487495	100.0
					Uppaal	13078s	1807698	121.5
5	3	7	4	2	DEMKit	13s	353418	100.0
					Uppaal	1045s	365800	103.5
	3	7	0	0	DEMKit	10s	2048285	100.0
					Uppaal	99s	2015465	98.4
	3	0	4	0	DEMKit	6s	1471153	100.0
					Uppaal	46s	1469557	99.9
	3	0	0	2	DEMKit	6s	74913	100.0
					Uppaal	506s	58283	77.8
6	3	0	0	1	DEMKit	6s	454253	100.0
					Uppaal	329s	449121	98.9

Table 7.3: Results comparing DEMKit against Uppaal.

7.2.1 Evaluation

The results of the tests comparing Uppaal against DEMKit can be found in Table 7.3. Six different scenarios have been tested. For each scenario the length of the schedule that needs to be created is given in the number of days. For each scenario the number of timeshiftables, EVs and batteries is also given. There are still many specifics that are not listed, however, the complete scenarios as output by the ALPG can be found in Appendix C.

The scores represent the sum of the quadratic energy consumption. However, in order to reduce the size of the numbers, the scores have been divided by sixty multiplied by the number of houses. Therefore, the scores should not be compared between different scenarios. A column with the normalized scores has also been given. In the normalize column, the score from DEMKit is always set to 100.0. For Uppaal a lower normalized score means a lower cost and, therefore, a better schedule.

Scenario 1

The first scenario calculates a schedule for five days and does so for twelve timeshiftables, four EVs and one battery. This represents a neighbourhood of ten houses, although the number of houses should have little impact besides deciding the number of devices. It can be concluded that DEMKit and our algorithm create a schedule with similar performance. The resulting schedules for DEMKit and Uppaal can be found in Figures 7.2 and 7.3 respectively. In these figures the y-axis displays the energy consumption in Wh, which is 3600 Joule, and the x-axis displays the energy consumption over time, per fifteen-minute interval. There are four lines, each representing one type of device, such that it can be observed when a certain device makes worse decisions. The uncontrollable or static energy consumption is indicated by the purple line. The timeshiftable, EV and battery energy consumptions are indicated by the red, blue and cyan lines respectively. The solid green area displays the combined energy consumption of all devices, meaning it is the sum of the four lines. The solid green area is what is used to calculate the quadratic energy consumption indicated as the *Score* in Table 7.3.

There are slight differences between the schedules. However, the schedules follow more or less the same trend. This is expected due to the steps in our algorithm resembling those of profile steering. One noticeable difference is that the EV and battery schedules provide flatter energy consumptions in the schedules created by Uppaal. This is due to the valley-filling implementation. The schedule created by DEMKit has flat profiles due to valley-filling as well, although there are minor variations, whereas Uppaal creates a completely flat plateau. This is not to say that the DEMKit battery and EV schedules are worse, in fact some instances can be found where they appear to be better.

Although the scores of the schedules are close, the durations it took to calculate the schedules are widely different. The algorithm using Uppaal was thirty times as slow as DEMKit. This is a huge, anticipated, difference. The other tests partially reveal why this difference is so large.

Scenario 2

The second scenario is created using the same ALPG configuration file as the first scenario to validate that the previous result was not a fortunate accident. Although the configuration file is the same, the created scenario is different, whilst still similar to the first scenario. Just as for the first scenario, a slight difference between the schedules exists. Unlike the

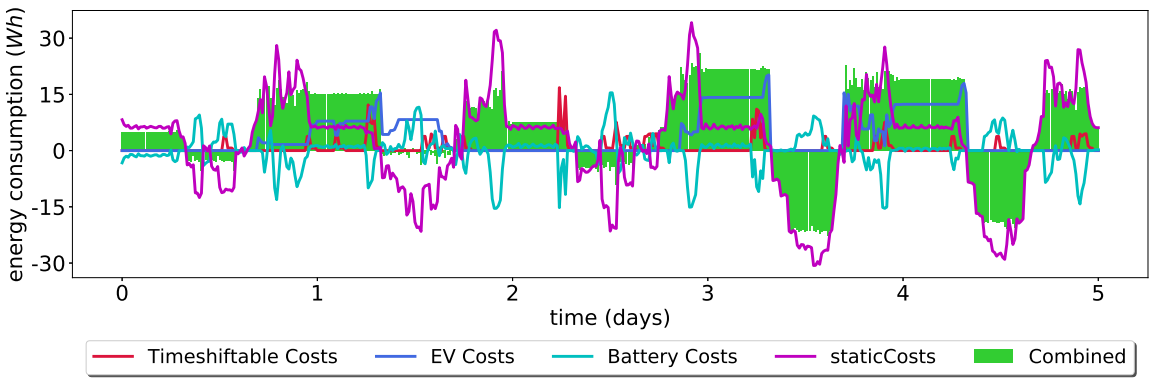

Figure 7.2: DEMKit schedule for scenario 1.

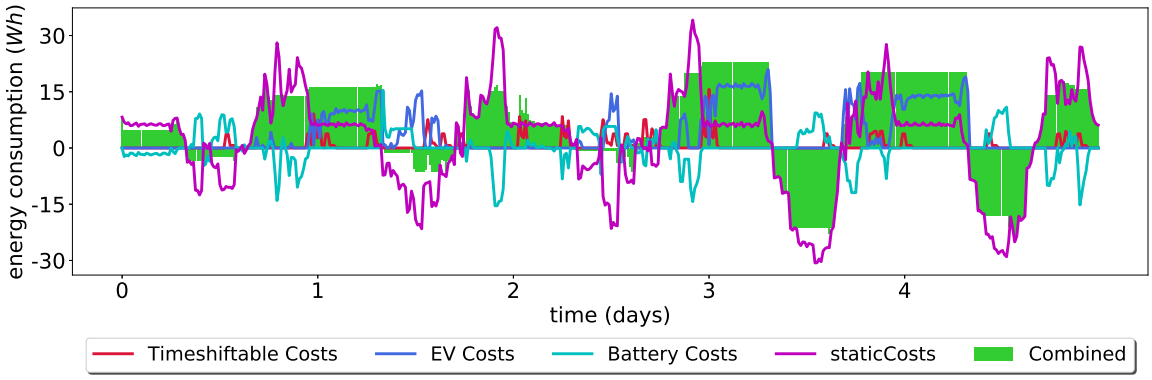

Figure 7.3: Uppaal schedule for scenario 1.

first scenario, it took Uppaal eleven times longer to create a schedule than it did DEMKit, which is less than in the first scenario. This difference is due to the first scenario having a battery, whereas this scenario does not. As shown in Section 7.1, the battery takes a long time for Uppaal to optimize, which explains why it takes much less time to generate a schedule without a battery.

Scenario 3

The third scenario creates a schedule for forty-eight days, having to schedule only a few devices to keep the duration reasonable. Again the score between DEMKit and Uppaal is very close, this time in favour of the Uppaal schedule. Creating the schedule took Uppaal about eleven times longer than it did DEMKit, which is similar to scenario two. Similar to scenario two, this scenario also does not have a battery. This indicates that our algorithm scales about as well with time as DEMKit.

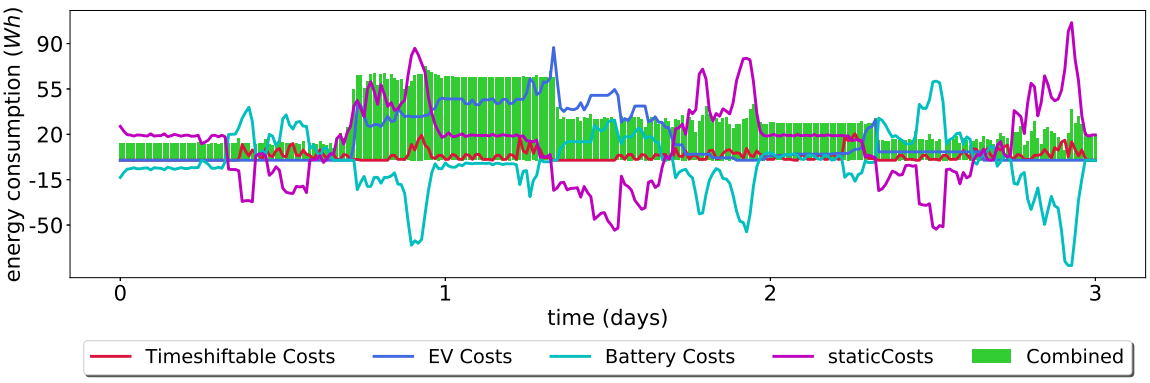

Figure 7.4: DEMKit schedule for scenario 4.

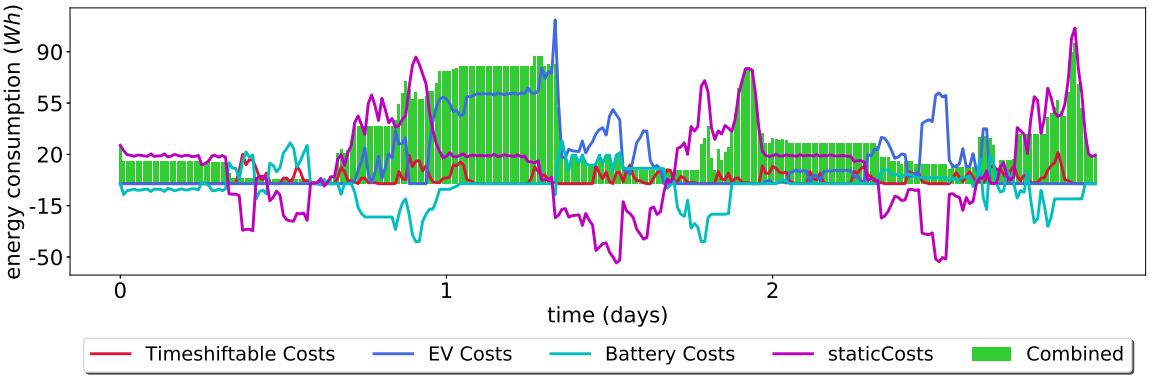

Figure 7.5: Uppaal schedule for scenario 4.

Scenario 4

The fourth scenario has been created to schedule a large amount of devices. This time the schedule created by Uppaal is much worse than the schedule created by DEMKit. The resulting schedules can be seen in Figures 7.4 and 7.5. From those schedules it can be concluded that the EV and battery schedules are the biggest contributing factors to the discrepancy in performance. The reason why the schedule is this much worse when there are many devices has to do with the Uppaal Stratego parameters, which is further discussed in Section 7.2.2. The duration to calculate the schedule is about thirty times longer for Uppaal, which is consisted with the result of scenario one[1].

Scenario 5

The fifth scenario consists of four different tests. The first test is calculating the schedule when all devices are taken into account. The other

[1]The test had to be paused for roughly six thousand seconds due to having to move location, which has been subtracted from the reported time.

three tests each take only one device type into account, in order to see how each device type compares between Uppaal and DEMKit. When scheduling all devices, the result is comparable to scenario one and two, which is as expected. The difference is that the amount of times Uppaal is slower than DEMKit has risen to a hundred. This rise can be explained by the fact that a higher percentage of the devices is a battery.

For both timeshiftables and EVs the schedules are very similar between Uppaal and DEMKit, both having slightly more efficient schedules than DEMKit. Timeshiftables took ten times as long to calculate and EVs took about eight times as long. These durations are consistent with scenario two and three, in which there did not exist batteries.

The difference in battery schedules between the two algorithms is interesting, because the Uppaal schedule is more efficient. Both battery schedules can be found in Figures 7.6 and 7.7. Due to both batteries having quite a large capacity and initial SoC, the combined energy consumption, indicated by the solid green area, is very low in both case. It is difficult to pinpoint exactly why the Uppaal schedule is more efficient. However, the difference probably seems more meaningful than it actually is, due to small differences being interpreted as a relatively large difference when the scores are this low. Calculating the more efficient schedule took a long time, roughly eighty-four times as long as DEMKit.

Scenario 6

Scenario six has been added to test the schedule created for a single battery with a smaller capacity than the ones in scenario five. This should be a more realistic comparison. The relative difference between the scores is much smaller than in scenario five, however, the Uppaal schedule remains more efficient. The Uppaal schedule did take about fifty times as long to calculate, which is less than with two batteries, but still a large difference.

7.2.2 Overall comparison

Throughout all the scenarios there is one consistent factor, that the current Uppaal algorithm is much slower than the DEMKit algorithm. Although this was expected, the actual difference is definitely an indication that the current algorithm needs some major adjustments to be comparable in both duration and performance to DEMKit. The main reason why it takes this long to calculate good schedules lies in the battery template. As mentioned in Section 5.4, batteries have many possible

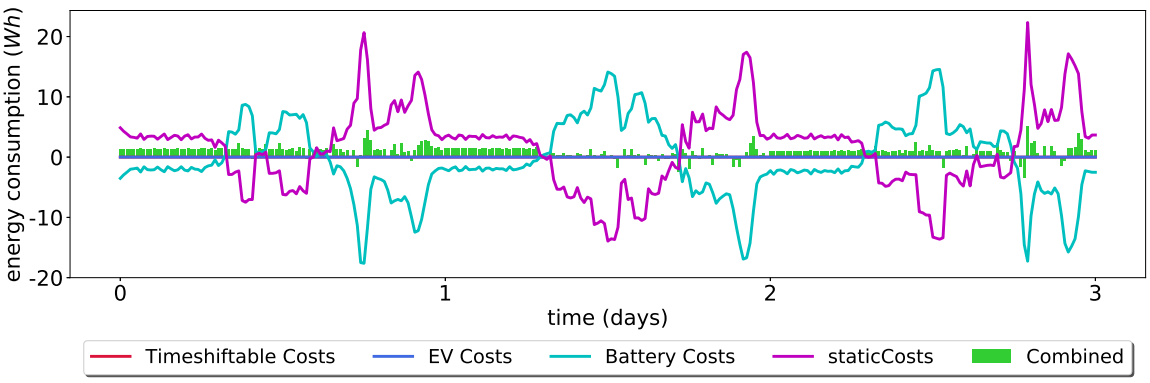

Figure 7.6: DEMKit battery schedule for scenario 5.

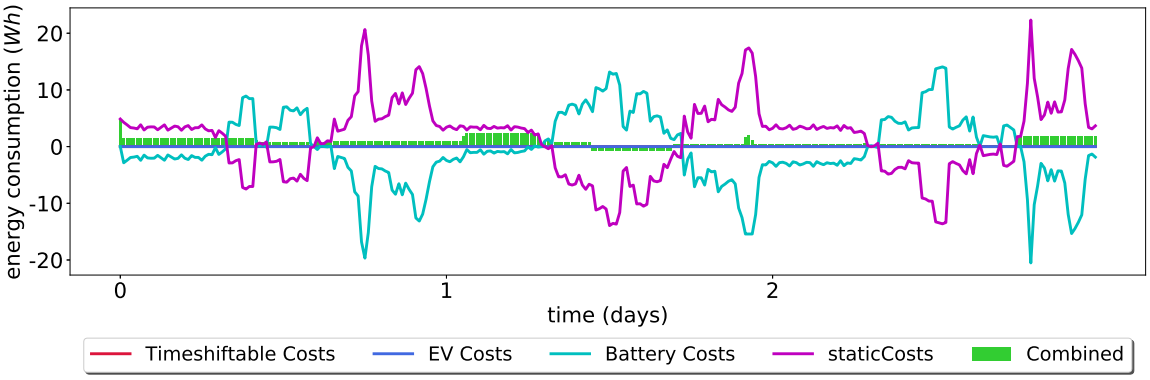

Figure 7.7: Uppaal battery schedule for scenario 5.

schedules, resulting in Uppaal not being able to find a good schedule without simulating many of these schedules. All of the restrictions added to the battery model to limit the amount of possibilities did not negate this problem.

It has been shown, using scenario three, that calculating a schedule over a longer duration has roughly the same scaling factor as DEMKit. This means the online part of the algorithm is working as intended. The compositional part of the algorithm has been less successful with its scaling. Although the durations in the results follow a pattern, this is caused by using the same learning parameters for all tests, which is not always optimal. Scenario four revealed that the learning parameters should be adjusted to the number of devices in the scenario. Learning parameters that are set below the required value will result in less good schedules, or even bad ones when the learning parameters are set far too low.

The reason why the amount of devices influences the required values for the learning parameters, is that each device that is made static, as explained in Section 5.7, induces additional randomness into the system.

This randomness comes from committed states that are executed in a random order, as explained in Section 4.3. Although the model is made such that the order of these committed states has no influence on the outcome, the Stratego algorithm does not consider this. Therefore, all of this randomness increases the number of possibilities that the learning algorithm needs to simulate to be able to give a good schedule.

Whilst the Uppaal algorithm is much slower than DEMKit, it is able to calculate equally good schedules, sometimes even slightly better. However, the output of DEMKit could also be slightly improved by changing the amount of iterations and the minimum difference between iterations before the algorithm stops. Therefore, in its current state, our algorithm would never be considered for any real world application. What the algorithm does show is that a DSM scenario can be modelled inside of TAs and most likely even in finite automata, as suggested in Section 5.10. It has also been shown that both the number of devices and the length of the schedule can be managed using TAs. This does require the use of an additional program, in this case our Python program, to make the algorithm iterative.

7.3 Discussion

We set out to answer the following three main questions with these tests:

1. How do the schedules created using TAs compare to the profiles created by the existing solutions?
2. How well does our algorithm scale?
   (a) How many devices can be scheduled?
   (b) What amount of time can be scheduled?
   (c) Can the computation time be reduced by sacrificing accuracy?
3. What are strengths and weaknesses of using timed automata for solving the scheduling problem?

7.3.1 Schedules

The first question is how the schedules created using TAs compare to the profiles created by the existing solutions, which is the profile steering implementation inside of DEMKit. This comparison is focussed on the schedules created and what their quadratic energy consumption is. The

outcome is that the both schedules are similar, assuming enough time is given. There are minor differences between the schedules, mainly between the EVs and batteries. For EVs the difference comes from Uppaal using the *fill limit* in order to precisely calculate the amount of energy consumed at each point in time, thereby creating intervals in which the energy consumption is completely flat. The schedule created by DEMKit has these same flat intervals, however, they contain slight deviations. The battery schedules are different due to the restrictions placed on the Uppaal model, which prevent certain schedules that DEMKit creates. Even with these differences, the efficiencies of both schedules are very close and the energy consumption profiles generally follow the same trend.

7.3.2 Scaling

The second question to be answered is about how the TAs scale. This question is split into three parts, namely, the amount of devices that can be scheduled, the amount of time that can be scheduled and whether the speed can be increased by sacrificing accuracy. The amount of devices that can be scheduled has a limit, due to memory restrictions. However, the limit is quite high as shown in scenario four of the comparisons. The amount of time for which schedules can be created is theoretically without a limit. However, that is when considering the devices can be scheduled one at a time and that the schedule can be created two days at a time. When considering how many devices can be scheduled with using nothing but Uppaal, it is rather limited and highly dependent on the devices. In Uppaal Cora, the answer is roughly four timeshiftables or two EVs over a duration of two days. For Uppaal Stratego the answer is more complex. It depends on the amount of time given and the desired quality of the schedule. This is related to the second sub-question, whether performance can be gained by sacrificing accuracy. The answer is a definite *yes* for Uppaal Stratego, simply by adjusting the learning parameters. For Uppaal Cora this is more difficult, however, it is possible to add extra restrictions that limit the state space, thereby giving a, most likely, worse schedule in less time.

Although we have not implemented a scalable algorithm using Uppaal Cora, it should be noted that Uppaal Cora was a hundred times faster than Uppaal Stratego when it comes to timeshiftables, thereby beating DEMKit with a large margin. Uppaal Cora can also calculate multiple timeshiftables at once, which can potentially result in better schedules

than DEMKit. However, determining exactly how Uppaal Cora compares to DEMKit when it comes to scheduling timeshiftables is left for future research.

7.3.3 Strengths and weaknesses

The last question is: what are the strengths and weaknesses of using TAs for solving the DSM scheduling problem? Only the strengths and weaknesses relating to the performance are described here; the complete set of strengths and weaknesses is described in Chapter 8. A major strength is how Uppaal Cora is able to calculate the schedule for timeshiftables. This strength comes from the fact that the state of timeshiftables is determined by when they are activated and what the next interval is. This ensures that the *cost* is the only distinguishable feature for many states, which is not included in the state space. Therefore, many states are identical and are merged during the state space reduction. Unfortunately, this strength does not apply to EVs and batteries, due to both these templates requiring a variable that keeps track of the SoC. The SoC is unlikely to be identical in two states, therefore, the state space cannot be reduced as much. Having a large state space means that the calculation of an optimal schedules takes longer and consumes more energy, which, unfortunately, makes Uppaal Cora less suitable for such devices.

The main weakness is that Uppaal Stratego is rather slow compared to DEMKit. The reason Uppaal Stratego is slow is due to the amount of simulations needed to give a good answer. For timeshiftables it is inefficient to run complete simulations, which is why Uppaal Cora is much better for those. For both EVs and batteries a good *fill limit* needs to be calculated. For EVs this can be done efficiently in Uppaal, considering there are only static costs involved. However, this can be done just as efficiently, if not more efficiently, outside of Uppaal, due to the entire calculation being done using code inside of Uppaal. The *fill limits* for batteries can be calculated much more quickly outside of Uppaal, this is mainly due to having more capabilities in a traditional programming language and because the *fill limit* can be adjusted without needing a completely new simulation.

7.4 Conclusion

Although TA are not an ideal solution to the DSM scheduling problem, by creating these schedules it has been shown that scenarios can be modelled

inside of Uppaal. It has also been shown that the created schedules can be as good as the schedules created by DEMKit, which means the problem is the computation time. There are several improvements that can be made to improve the computation time required to calculate the schedules. The main improvement is using Uppaal Cora for calculating the schedules for the timeshiftable devices. The problem of calculating a good battery schedule remains. However, it would be possible to create the battery schedule using an object-oriented programming language, rather than using TA, thereby creating a hybrid algorithm. Not only would these improvements increase the speed of the algorithm, it could even be possible to create a better schedule in less time than the current DEMKit implementation. These improvements would mainly be due to Uppaal Cora calculating optimal timeshiftable schedules in very little time. All of these improvements can be attempted in future research, which appears to be a promising direction for optimizing DSM schedules.

Chapter 8

Conclusion

During this thesis we set out to answer the following research question: *Can Timed Automata (TAs) correctly and efficiently model and simulate a residential Demand Side Management (DSM) scenario?* To answer this question, three research questions needed to be answered. These questions are the following:

1. Are timed automata suitable for solving the demand side management scheduling problem?
   (a) How do the profiles compare to profiles created by existing solutions?
   (b) Can the models be adapted to new situations?
2. How do the created timed automata scale?
   (a) What amount of data can be handled, expressed in the amount of days that need to be scheduled, devices and households?
   (b) Can performance be increased by sacrificing accuracy?
3. What are strengths and weaknesses of using timed automata for solving the scheduling problem?

All of these questions are answered in the sections below, after which Section 8.4 answers the main research question.

8.1 Suitability

During this research, we have created an algorithm that relies on Uppaal for scheduling the devices. The schedules this algorithm creates have a similar performance to the Decentralized Energy Management Kit (DEMKit) schedules. This indicates that TAs are in fact suitable for

solving the DSM scheduling problem. The models can also be adapted to new situations, although this does require knowledge about Uppaal and how the model is made. Depending on the adjustment, it can be easy or extremely difficult to implement. However, due to how generic the modelling language is, many alterations can be made with relative ease. Therefore, we conclude that TAs are in fact suitable for solving the DSM problem. How suited they are still depends on the performance, which is discussed in the next section.

8.2 Scalability

When discussing the scalability, both the model itself and the online compositional algorithm can be considered. The model itself scales rather poorly with both devices and time. The number of households has no influence, except resulting in higher numbers for the static costs. How poorly the model scales depends on the version of Uppaal that is used. Uppaal Stratego scales the best due to not needing to calculate the optimal schedule, whereas Uppaal Cora is very limited in the amount of devices it can schedule. Scaling over time is a larger issue for Uppaal Stratego than it is for Uppaal Cora. This is due to Uppaal Cora splitting the problem when, at a certain time unit, there is only one state the model can be in. This is the case when no devices are in an activation interval, which happens regularly with smaller neighbourhoods. The exact amount of data that can be handled is difficult to determine.

When considering the online compositional algorithm, the current implementation can handle almost any amount of time. The amount of devices that can be scheduled is limited with the current implementation of the algorithm. The exact maximum has not been discovered, because this depends on the devices that are being scheduled. We estimate that fifty devices is roughly the maximum that can be scheduled, whilst maintaining a schedule with an accuracy similar to a schedule from DEMKit. When adding more devices, the learning parameters need to increase in order to generate a good schedule, which can cause Uppaal to run out of memory.

It is definitely possible to increase the speed of the algorithm by sacrificing accuracy. In Uppaal Stratego this is done by lowering the learning parameters. For Uppaal Cora it is possible to add restrictions to the model, thereby reducing the state space and thus lowering the amount of time it takes to generate the schedule.

Overall, the scaling of the models themselves is as one would expect when solving an NP-hard problem, which is not good. The online compositional algorithm mitigates the NP-hardness to a certain extend, however, the current implementation still has its limitations. We do expect that it is possible to create an algorithm that scales well with both time and the number of devices. However, when using only TAs it is unlikely that an algorithm will ever be as fast as DEMKit for batteries.

8.3 Strengths and weaknesses

The strengths of using TAs are as follows:

- Uppaal Cora can efficiently schedule timeshiftables due to its state space reduction techniques.
- The modelling technique is generic.
  - It can be easier to understand an Uppaal model than an algorithm such as DEMKit.
  - The model provides flexibility in the optimization criteria.
  - Alterations to the model can be made with relative ease.
- TAs allow for more than only scheduling.
  - The model can be used to verify properties of the system.
  - TAs are able to find the odds of causing a grid overload.
  - The model is able to generate statistics about the behaviour of the system.

The key strength regarding optimizing a schedule is that timeshiftables can be scheduled extremely quickly using Uppaal Cora. The other strengths are all caused by the modelling technique being generic and by allowing for more than schedule generation. Although the modelling technique being generic is useful, in a real life scenario performance and speed are more important than whether people can understand what is happening. There can definitely be use cases where the added functionality of TAs, which allows for verification and the generation of statistics, can be useful. However, this is mostly aimed at researching and testing, and is less important when DSM is implemented in the real world.

The weaknesses of using TAs are as follows:

- TAs are slow when optimizing devices that often change their state, such as batteries and Electrical Vehicles (EVs).
- Small mistakes in the model are difficult to find and fix due to the inability to debug the model. At least this is the case in Uppaal.
- Algorithms making use of TAs are limited to what can be expressed as a TA.
- Uppaal has a steep learning curve.
  - When using Uppaal it is challenging to collaborate with people new to Uppaal and TAs.

When purely focussing on the scheduling aspect, TAs are relatively slow at scheduling devices that often change their state. This applies to buffer devices, such as EVs and especially batteries. EVs are able to be scheduled in a reasonable amount of time, batteries on the other hand are not. The other issues have to do with how challenging it can be to work with TAs and Uppaal in general. Small mistakes are easily made and can be challenging to locate when it happens inside of a function. The only method of discovering the mistake is by looking at how the state of the system changes over time, which can be especially challenging for large models. It is also not possible to express everything within TAs, therefore, there are limitations to what is possible. In contrast, an object oriented programming language has almost no limit to what is possible. Lastly, it can be difficult for people new to TAs and Uppaal to learn the details of how it works. Initially everything seems intuitive, however, there are many small details that work counter to how people expect. We experienced this learning curve first hand. Even after working with Uppaal for over half a year we still learned about new behaviour. It does not help that the documentation for Uppaal is not always consistent and often does not explain all details.

8.4 Verdict

The question we set out to answer is whether TAs are able to correctly and efficiently model and simulate a residential DSM scenario. It has been shown that the scenario can be modelled correctly. Whether it can be done efficiently is another matter. The answer is that it can be simulated efficiently. However, the scheduling based on the simulations is not always efficient. This is due to batteries taking a long time when being scheduled. When considering the different devices, we conclude that both

timeshiftables and EVs can be simulated and scheduled efficiently. EVs take longer to schedule than timeshiftables, however, we are confident that a good implementation of the online compositional algorithm can bring down the durations of both these devices to the level of DEMKit.

Overall, we can imagine TAs being used in some shape or form for DSM. The method in which timeshiftables are scheduled is especially interesting. There could also arise use cases in which statistics about a scenario are needed, such as the odds of causing an overload. For such a purpose, TAs can provide a good solution. Uppaal Stratego could also be useful when using auction based steering, as is explained in Chapter 9.

Chapter 9

Limitations and future work

This chapter discussed some areas that require further research. Section 9.1 discusses several improvements that can be made to the algorithm to reduce the computation time and improve the results. Then, Section 9.2 discusses the possibility of using finite automata instead of Timed Automata (TAs). Section 9.3 discusses the addition of different devices, after which, Section 9.4 discusses the usage of Uppaal to generate statistics about Demand Side Management (DSM) scenarios. Section 9.5 argues about the addition of randomness into the model to create schedules that are more robust against mistakes in the predictions. Finally, Section 9.6 discusses the possibility of using TAs in an auction-based DSM scenario.

9.1 Improving the scheduling algorithm

As discussed in Chapter 7, the algorithm developed during this research gives results that match closely with the results of the Decentralized Energy Management Kit (DEMKit). The major difference lies in the computation time. Throughout this thesis several improvements for the speed of the algorithm have been suggested. The most prominent improvement would be to have Uppaal Cora calculate the schedules for timeshiftable devices, which was shown to be much faster (in Section 7.1). The next improvement is to add the timeshiftable costs directly onto the static costs, rather than changing the template into a static template, which requires much less calculating time. With the timeshiftable costs added onto the static costs it also becomes possible to calculate the optimal fill limit for the Electrical Vehicle (EV), therefore there is no longer the need to try different fill limits. The speed for longer scenarios can also be improved by only having part of the static costs array inside of the model during each interval of the online algorithm. Having a long array slows down the loading of the model as well as the lookups inside of

the array, of which there are many. Finally, the battery optimization could be done using Python instead of Uppaal. This change is prompted by the calculation of battery fill limits being much faster inside of DEMKit.

With these optimizations in place it also becomes feasible to have the compositional algorithm schedule more than one device at a time. Currently each component group, which is a set of devices that are optimized together, contains only one device in order to lower the computation time. With a more optimized algorithm the results could potentially be improved when component group consists of multiple devices. It should then be researched whether larger component groups improve the results. The composition of these component groups can also be researched. It could prove beneficial to schedule devices of different types simulations, such that they can properly compensate for each other.

9.2 Finite automata

As discussed in Section 5.10, the models that have been created to optimize the schedules make little use of clocks. The model can be adjusted such that it does not use clocks, turning the model into a finite automaton. In future research it can be investigated whether there is another tool that makes use of finite automata and can create a schedule. This could prove to be more efficient, although it is our assumption that the difference is minimal. We assume this based on the fact that the clocks add little computation time to the system. The only reason we can imagine that finite automata might be significantly more efficient is that additional mathematical techniques can be applied.

9.3 Adding additional device types

Currently the model contains three types of devices, namely, timeshiftables, EVs and batteries. Due to time restrictions we did not manage to include a smart thermostat device, which could be added in future research. Keeping the temperature of the room within bounds is one of the more computationally challenging problems, therefore it could turn out to be challenging to add a thermostat to the model. It would also have to be investigated which version of Uppaal best suits such a model; our hypothesis is that Uppaal Stratego is the most suited.

Besides adding more devices, the current devices could also be expanded upon. There could be timeshiftables that can be interrupted

during parts of their routine. Another example would be EVs that can function as a battery whilst they are charging, meaning they can discharge to the grid as well.

There are many more possibilities for extending the model, some might not be feasible due to their complexity, whereas others could be surprisingly cheap to implement.

9.4 Generating statistics

This research has proven that an Artificial Load Profile Generator (ALPG) scenario can be modelled using TA. However, the models have been used to optimize the schedules, rather than perform checks on the model. An example would be, given a schedule and the standard deviations for the predictions, “what are the odds that the grid is overloaded”? Instead of using schedules there could also be predefined controllers, for which statistics could then be obtained.

Due to TAs being a very generic modelling technique, there are many possibilities for obtaining insight into different situations. The advantage of TAs being generic is that additional restrictions or requirements can be added fairly easily. The downside is that, when a mistake is made, it can be challenging to find out what the mistake is.

9.5 Account for wrong predictions

Instead of creating a schedule based on static data, Uppaal Stratego could, theoretically, also be used to optimize the schedule such that prediction errors have as little impact as possible. This can be accomplished by adding random variations to the data, which the optimizer has to account for. The aforementioned variations could be used to optimize the schedule for a single device, for which the parameters are known, that has an imperfect prediction of everything else inside of the environment. Such a schedule could then be created for every device independently, resulting in a complete schedule without having exact knowledge of other devices inside of the systems. This is a scenario in which dynamic dispatching of instances could be of interest to semi-randomly add devices to the system.

9.6 Auction-based steering

The last future research suggestion we have is to use TAs to optimize an auction-based market, in which the energy provider alters the prices based on the amount of energy being consumed. The model could then be used to optimize the energy prices over time, based on the expected behaviour of residents and devices. In such a scenario randomness could be added as well, in order to adequately model humans that are less predictable. Such a model comes much closer to being usable in reality to optimize the energy prices. The model would still require some knowledge about the behaviour of residents and devices, however, the model would ensure that deviations from their expected behaviour do not endanger the energy grid.

In our opinion this appears to be an interesting application, due to energy providers already using the price of energy to attempt to steer the energy consumption of a neighbourhood. Using Uppaal Stratego could definitely be beneficial, however, there is also a risk that such a task is too large to be calculated at once. Furthermore, it is more challenging to split a scenario with randomness into multiple optimization steps than it is to split a scenario in which everything is predetermined. The challenge arises due to the randomness making it so that the humans would react differently depending on the randomness. Therefore, the behaviour of one simulation would not make sense for another simulation with different random values. Thus, the schedule of a device cannot be made static, as is done in a compositional algorithm.

References

[1] Oxfordlearnersdictionaries.com. (2019). Electrification, [Online]. Available: https://www.oxfordlearnersdictionaries.com/definition/english/electrification?q=electrification (visited on 29 Jan. 2019).

[2] H. Toersche, "Effective and efficient coordination of flexibility in smart grids", PhD thesis, University of Twente, Netherlands, Oct. 2016, ISBN: 978-90-365-4197-8. DOI: 10.3990/1.9789036541978.

[3] H. Farhangi, "The path of the smart grid", *IEEE Power and Energy Magazine*, vol. 8, no. 1, pp. 18–28, Jan. 2010, ISSN: 1540-7977. DOI: 10.1109/MPE.2009.934876.

[4] M. E. T. Gerards, H. A. Toersche, G. Hoogsteen, T. van der Klauw, J. L. Hurink and G. J. M. Smit, "Demand side management using profile steering", in *2015 IEEE Eindhoven PowerTech*, Jun. 2015, pp. 1–6. DOI: 10.1109/PTC.2015.7232328.

[5] A. Mohsenian-Rad, V. W. S. Wong, J. Jatskevich, R. Schober and A. Leon-Garcia, "Autonomous demand-side management based on game-theoretic energy consumption scheduling for the future smart grid", *IEEE Transactions on Smart Grid*, vol. 1, no. 3, pp. 320–331, Dec. 2010, ISSN: 1949-3053. DOI: 10.1109/TSG.2010.2089069.

[6] W. Saad, Z. Han, H. V. Poor and T. Basar, "Game-theoretic methods for the smart grid: An overview of microgrid systems, demand-side management, and smart grid communications", *IEEE Signal Processing Magazine*, vol. 29, no. 5, pp. 86–105, Sep. 2012, ISSN: 1053-5888. DOI: 10.1109/MSP.2012.2186410.

[7] H. Chen, Y. Li, R. H. Y. Louie and B. Vucetic, "Autonomous demand side management based on energy consumption scheduling and instantaneous load billing: An aggregative game approach", *IEEE Transactions on Smart Grid*, vol. 5, no. 4, pp. 1744–1754, Jul. 2014, ISSN: 1949-3053. DOI: 10.1109/TSG.2014.2311122.

[8] Uppaal.org. (Jun. 2015). UPPAAL, [Online]. Available: http://www.uppaal.org/ (visited on 4 Dec. 2018).

[9] A. David. (Feb. 2014). UPPAAL Cora, [Online]. Available: http://people.cs.aau.dk/~adavid/cora/ (visited on 4 Dec. 2018).

[10] ——, (Jul. 2011). UPPAAL SMC, [Online]. Available: http://people.cs.aau.dk/~adavid/smc/ (visited on 4 Dec. 2018).

[11] Uppaal.org. (Nov. 2015). UPPAAL Stratego, [Online]. Available: http://people.cs.aau.dk/~marius/stratego/ (visited on 31 Jan. 2019).

[12] M. E. T. Gerards and J. L. Hurink, "On the value of device flexibility in smart grid applications", in *2017 IEEE Manchester PowerTech*, Jun. 2017, pp. 1–6. DOI: 10.1109/PTC.2017.7981170.

[13] G. Hoogsteen, "A cyber-physical systems perspective on decentralized energy management", CTIT Ph.D. thesis series no. 17-449, PhD thesis, University of Twente, Netherlands, Dec. 2017, ISBN: 978-90-365-4432-0. DOI: 10.3990/1.9789036544320.

[14] A. David, D. Du, K. Guldstrand Larsen, A. Legay and M. Mikučionis, "Optimizing control strategy using statistical model checking", in *NASA Formal Methods*, G. Brat, N. Rungta and A. Venet, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 352–367, ISBN: 978-3-642-38088-4. DOI: 10.1007/978-3-642-38088-4_24.

[15] W. Ahmad and J. van de Pol, "Synthesizing energy-optimal controllers for multiprocessor dataflow applications with uppaal stratego", in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2016, pp. 94–113, ISBN: 978-3-319-47166-2. DOI: 10.1007/978-3-319-47166-2_7.

[16] G. Behrmann, K. G. Larsen and J. I. Rasmussen, "Priced timed automata: Algorithms and applications", in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf and W.-P. de Roever, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 162–182, ISBN: 978-3-540-31939-9. DOI: 10.1007/11561163_8.

[17] T. van der Klauw, "Decentralized energy management with profile steering: Resource allocation problems in energy management", CTIT Ph.D. thesis series no. 17-424, PhD thesis, University of

Twente, Netherlands, May 2017, ISBN: 978-90-365-4301-9. DOI: 10.3990/1.9789036543019.

[18] G. Hoogsteen. (2019). Alpg, [Online]. Available: https://github.com/GENETX/alpg (visited on 29 Jan. 2019).

[19] M. R. Jongerden, "Model-based energy analysis of battery powered systems", PhD thesis, University of Twente, Dec. 2010, ISBN: 978-90-365-3114-6. DOI: 10.3990/1.9789036531146.

[20] A. Rubinstein, "Finite automata play the repeated prisoner's dilemma", *Journal of Economic Theory*, vol. 39, no. 1, pp. 83–96, 1986, ISSN: 0022-0531. DOI: 10.1016/0022-0531(86)90021-9.

[21] R. Alur and D. L. Dill, "A theory of timed automata", *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994, ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90010-8.

[22] K. G. Larsen, P. Pettersson and W. Yi, "Uppaal in a nutshell", *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1, pp. 134–152, Dec. 1997, ISSN: 1433-2779. DOI: 10.1007/s100090050010.

[23] A. L. Alexandre David Kim G. Larsen. (Jan. 2018). Uppaal smc tutorial, [Online]. Available: https://www.it.uu.se/research/group/darts/papers/texts/uppaal-smc-tutorial.pdf (visited on 2 Feb. 2019).

[24] G. Behrmann, K. G. Larsen and J. I. Rasmussen, "Optimal scheduling using priced timed automata", *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 4, pp. 34–40, Mar. 2005, ISSN: 0163-5999. DOI: 10.1145/1059816.1059823.

[25] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn and F. Vaandrager, "Minimum-cost reachability for priced time automata", in *Hybrid Systems: Computation and Control*, M. D. Di Benedetto and A. Sangiovanni-Vincentelli, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 147–161, ISBN: 978-3-540-45351-2. DOI: 10.1007/3-540-45351-2_15.

[26] A. David, K. G. Larsen, A. Legay, M. Mikučionis and D. B. Poulsen, "Uppaal smc tutorial", *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, Aug. 2015, ISSN: 1433-2787. DOI: 10.1007/s10009-014-0361-y.

[27] A. David, P. G. Jensen, K. G. Larsen, M. Mikučionis and J. H. Taankvist, "Uppaal stratego", in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 206–211, ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_16.

[28] A. David, P. G. Jensen, K. G. Larsen, A. Legay, D. Lime, M. G. Sørensen and J. H. Taankvist, "On time with minimal expected cost!", in *Automated Technology for Verification and Analysis*, F. Cassez and J.-F. Raskin, Eds., Cham: Springer International Publishing, 2014, pp. 129–145, ISBN: 978-3-319-11936-6. DOI: 10.1007/978-3-319-11936-6_10.

[29] Y. Mou, H. Xing, Z. Lin and M. Fu, "Decentralized optimal demand-side management for phev charging in a smart grid", *IEEE Transactions on Smart Grid*, vol. 6, no. 2, pp. 726–736, Mar. 2015, ISSN: 1949-3053. DOI: 10.1109/TSG.2014.2363096.

[30] G. Hoogsteen, A. Molderink, J. L. Hurink and G. J. M. Smit, "Generation of flexible domestic load profiles to evaluate demand side management approaches", in *2016 IEEE International Energy Conference (ENERGYCON)*, Apr. 2016, pp. 1–6. DOI: 10.1109/ENERGYCON.2016.7513873.

[31] G. Hoogsteen, "Demkit: A flexible smart grid simulation and demonstration platform written in python", in *Proceedings of the 'Energy-Open' workshop, WP 17-01*, 2017, pp. 26–27. [Online]. Available: https://energy-open.nl/local/abstracts/001_hoogsteen.pdf.

[32] A. David, J. Rasmussen, K. Larsen and A. Skou, "Model-based framework for schedulability analysis using uppaal 4.1", Nov. 2009. DOI: 10.1201/9781420067859-c4.

[33] W. Ahmad, P. K. F. Hölzenspies, M. Stoelinga and J. v. d. Pol, "Green computing: Power optimisation of vfi-based real-time multiprocessor dataflow applications", in *2015 Euromicro Conference on Digital System Design*, Aug. 2015, pp. 271–275. DOI: 10.1109/DSD.2015.59.

[34] K. G. Larsen, M. Mikučionis, M. Muñiz, J. Srba and J. H. Taankvist, "Online and compositional learning of controllers with application to floor heating", in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds., Berlin,

Heidelberg: Springer Berlin Heidelberg, 2016, pp. 244–259, ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_14.

[35] It.uu.se. (2012). Uppaal web help, [Online]. Available: http://www.it.uu.se/research/group/darts/uppaal/help.php?file=System_Descriptions/Expressions.shtml (visited on 9 Jan. 2019).

[36] M. Mikučionis. (2016). Uppaal timed automata parser library, [Online]. Available: http://people.cs.aau.dk/~marius/utap/ (visited on 7 Jan. 2019).

[37] Bugsy.grid.aau.dk. (2018). Bug list, [Online]. Available: https://bugsy.grid.aau.dk/bugzilla/buglist.cgi?product=UPPAAL%20Stratego (visited on 13 Dec. 2018).

[38] Uppaal.org. (Jul. 2014). UPPAAL Tiga, [Online]. Available: http://people.cs.aau.dk/~adavid/tiga/ (visited on 21 Feb. 2019).

[39] V. Birk. (2014). Pypeg, [Online]. Available: https://fdik.org/pyPEG/ (visited on 7 Jan. 2019).

Appendices

A Uppaal SMC functions

In Table A a list of SMC functions can be found. These functions are all supposed to be implemented according to the documentation [35]. This list has been created by looking at the parser source code available at [36] and splits the functions in two groups, the ones that have been implemented and the ones that are not implemented. Not every function on this list has been tested, however, assuming there are no differences between the parser used by the *server* and the provided parser, this list is correct.

B Textual representation of the structures

See Listing 1 for the textual definitions of the structures that are used inside of Uppaal.

C Source code

All of the source code can be found at our github page. The repository consists of five folders. The contents of each folder are listed below.

C.1 ALPG-Configs

The folder contains the Artificial Load Profile Generator (ALPG) configurations that are used to generate the scenarios used in Section 7.2. It should be noted that, although a seed is included, the results did vary when running the ALPG. Therefore, the actual scenarios used are included in another folder.

```
typedef struct {
int start;
int end;
} Interval_t;

typedef struct {
int length;
Interval_t intervalList[longestIntervalList];
} IntervalList_t;

typedef struct {
int length;
int data[longestConsumption];
} ConsumptionProfile_t;

typedef struct {
ConsumptionProfile_t consumption;
IntervalList_t data;
} TimeShiftable_t;

typedef struct {
UInt32_t maximumChargeRate;
UInt32_t capacity;
UInt32_t initialSoc;
} Battery_t;

typedef struct {
UInt32_t maximumChargeRate;
UInt32_t capacity;
UInt32_t requiredCharge[longestIntervalList];
Int32_t maxTsCosts[longestIntervalList];
IntervalList_t intervals;
} ElectricalVehicle_t;

typedef struct {
int timeShiftableLength;
TimeShiftable_t timeShiftables[maxNumTimeShiftables];
Battery_t battery;
ElectricalVehicle_t ev;
} House_t;
```

Listing 1: Uppaal structure declaration

Functions				
Implemented	Unimplemented			
ceil	abs	acos	acosh	asin
cos	asinh	atan	atan2	atanh
exp	cbrt	copysign	cosh	erf
fabs	erfc	exp2	expm1	fint
floor	fma	fmax	fmin	fmod
ln	hypot	ilogb	ldexp	lgamma
log	log10	log1p	log2	logb
pow	nextafter	normal	round	signbit
random	sinh	tan	tanh	tgamma
sin	trunc			
sqrt				

Table A: Uppaal SMC functions

C.2 ALPG-Scenarios

This folder contains the ALPG scenarios that have been used in Section 7.2. These can be used to reproduce the tests as closely as possible.

C.3 Algorithm

The *Algorithm* folder contains the entire code of the online compositional algorithm that has been created, including the Uppaal model used. It should be noted that the Uppaal model has had a slight adjustment since the tests in Section 7.2. This adjustment does not change the behaviour of the model, however, it does change the randomly generated numbers. Therefore, the tests cannot be run exactly the same. However, the results include the precise schedule that was generate, therefore, it is still possible to validate the result. These results are discussed in the next section. There are four python files:

- *Grammar.py* is used to parse the Uppaal model.
- *StrategoDeclaration.py* parses an ALPG scenario into the Uppaal declaration.
- *StrategoOnline.py* runs the online compositional algorithm.
- *DEMKitUppaalComparison.py* runs the tests, including the validation, as described in Section 6.2.2.

C.4 ResultsDemkit-Uppaal

This folder contains the results of the test scenarios. Each sub-folder contains five files.

- Two images, which are the plots created by the test. Note that these are not the image included in this thesis.
- Two pickle files that can be parsed using Python to get the exact results from the tests.
- The *result.txt* file, which contains the parameters used and the results in a textual form.

The folder also contains a python script named *remake_image.py*. This script was used to regenerate the plots based on the pickle files in each folder. Those are the plots that are included in this thesis.

C.5 UppaalVersionComparison

This folder contains three sub-folders, one for each device type that has been tested. Each sub-folder contains four files. The three models that have been used with the different versions and a textual file that contains the results and the commands to redo the test. The main folder also contains a query file that is used by Uppaal Cora, because it does not support queries inside of the model file.