Optimizing Modular Factory Configurations

- Using Timed Automata and Tabu Search -

Project Report
d701e16

Aalborg University
Department of Computer Science
Selma lagerløfs vej 300
DK-9220 Aalborg

Department of Computer Science
Selma lagerløfs vej 300
http://www.cs.aau.dk/

AALBORG UNIVERSITY

STUDENT REPORT

Title:
Optimizing Modular Factory Configurations

Theme:
From reality to models

Project Period:
Fall semester 2016

Project Group:
d701d16

Participant(s):
Alexander Brandborg
Mathias Claus Jensen

Supervisor(s):
Søren Enevoldsen

Copies: 0

Page Numbers: 77

Date of Completion:
December 21, 2016

Abstract:

Formålet med denne rapport er at udvikle et system, hvormed vi kan genere valide og effiktive fabrikskonfigurationer baseret på en mængde løse fabriksmoduler samt en ordre bestående af produkter, der skal produceres. Vi udvikler en model I UPPAAL, der gør at vi kan simulere produktionen af ordre på en konfiguration. I forlængelse kan vi bruge denne til at vurdere, hvor hurtig en konfiguration er til at producere sin ordre. Herefter definerer vi formelle regler, der bruges til at generere valide fabrikskonfigurationer fra et simpelt udgangspunkt. Disse regler implementeres i python og bruges, i tandem med vores UPPAAL model, i en implementeret tabu search, hvis formål er at søge konfigurationer igennem og finde den hurtigste. Ved sammenligning med en menneskeskabt konfiguration, finder vi at vi kan genere en ikke ens, men alligevel fornuftig konfiguration til at producere en ordre.

Contents

Preface

This report is written by project group d701e16 at Aalborg University as part of the 1st semester of the Master of Computer Science. The group has two members consisting of Alexander Brandborg and Mathias Claus Jensen. The topic of the semester was *Reality to Models*, and the chosen project proposal was *Model and Optimization of Production*. In this report we describe, how we develop a model of modular factories, and use it find the optimal factory configuration for producing an order of items.

We would like to thank our supervisor Søren Enevoldsen for his support and advice. Fellow student Martin Kristjansen gets our thanks for sparring with us during the semester. Thanks goes to Kim G. Larsen for his initial input to the project. Lastly we would like to thank Ole Madsen and Casper Schou from Aalborg University's *Department of Mechanical and Manufacturing Engineering* for answering our questions, and allowing us to inspect the department's CP Learning Factor set-up.

Both the UPPAAL model and python scripts refered to in the report can be found at `https://github.com/mathiascj/d701e16.git`.

Aalborg University, December 21, 2016

Alexander Brandborg
<abran13@student.aau.dk>

Mathias Claus Jensen
<mcj13@student.aau.dk>

Chapter 1

Introduction

We stand in the middle of a new paradigm shift within manufacturing known as Industry 4.0. While the term has no official definition, the European Parliamentary Research Service (EPRS)[1] defines it as entailing the transformation of traditional manufacture using modern digital technologies. In part, these include wireless communication, cyber-physical systems, simulation and modular factory systems.

Modular factory systems are made up of individual production modules, each performing some kind of job in a production line. These systems are flexible as their modules can be rearranged into a new production line. This is opposed to more traditional lines, which do not change often after being installed. The added flexibility of these systems allows manufacturers to quickly change the product, which they manufacture, to meet fast-changing market demands. The production of small product batches also becomes more affordable. This is beneficial when trying to reach niche markets, or when producing product prototypes as part of an iterative development process.

However, with these benefits comes the challenge of constantly re-planning the design of production lines. Traditional manufacturing concerns such as maximizing throughput of course have to be considered. Yet, we must also take care of challenges unique to the modular factory, such as the cost of moving from one production line to the next. This leads us to the project's initial problem:

How may it be possible to optimize the throughput of individual production lines in a modular factory system, while keeping down the overall cost of production?

We will narrow down this problem in the following problem analysis, as to arrive at a specific goal for this project.

Chapter 2

Problem Analysis

In this chapter, we delve deeper into the problem stated in chapter 1. While the eventual solution is aimed at the area of modular factories as a whole, we use the FESTO modular factory system [2] as a point of reference. This system was chosen specifically as Aalborg University's *Department of Mechanical and Manufacturing Engineering* have an existing set-up, which we were allowed to inspect.

2.1 FESTO

The FESTO system accessible to us is the Cyber-Physcial(CP) Learning Factory. This is not intended for full fledged production, but rather it is used in education and research. While not an industrial system, we deem that it will aid in a general understanding of the field and its related problems. The public literature about the CP Learning factory is rather light. The most reliable sources come from FESTO themselves, as in the case with [3]. To supply the written material, we also use our own observations about the local set-up, in addition to statements made by staff working the factory.

An example of a CP-factory set-up can be seen in fig. 2.1. Each module in a set-up is made up of the same base module. This includes a control panel as well as two conveyor belts moving in opposite directions. Modules can be connected either at the narrow ends or on the face opposite the control panel. On each module new equipment may be fitted. This can include a robot arm, a drill or a camera. A module works only on one item at a time. Yet several items may be located on the same module, being queued up and waiting to be worked on or pass through the module.

The factory control tasks are handled through a MES (Manufacturing Execution System) known as MES4. This controls production order dispatch, scheduling, resource management etc. During production the MES4 system signals individual modules to perform certain actions. These instructions are encoded into the module's own controller as KRL code.

Figure 2.1: A CP-factory setup

The factory also uses a large amount of networking technologies. RFID technology is used to monitor the location of individual items running along the conveyor belts. Individual modules may communicate by field-bus, while overall communications in the factory occurs over a WLAN. The factory may also be connected to an Enterprise Resource Planning (ERP) system or other factories through MES4. A production run is initiated by activating each module through its control panel, then launching from a PC.

2.2 Problems in Modular Factories

Through our discussion with staff at Aalborg University we identified some problems relating to use of the CP Learning Factory. Additional problems concerning modular factories in general were found by looking into the literature. In this section we will examine each of these issues.

2.2.1 Calculating an Optimal Factory Configuration

Aalborg staff relayed to us that they would like to generate optimal factory configurations according to some order of items, which must be produced. A configuration is here the same as the layout of a factory, which considers where modules are placed,

and which other modules they connect to.

Product Lifecycle Management (PLM) systems such as Siemens' Technomatix [4] have this sort of functionality included. Technomatix is able to simulate factories and their throughput before implementation. Yet, the system does not focus on rapidly changing factory configurations. It concerns itself with getting the configuration right before implementation. When new orders come in, Technomatix may suggest how to optimize production parameters for an implemented configuration. This may entail an alteration in how production is scheduled. A production schedule informs us, how we should produce an order of items using a specific configuration. Manufacturers are interested in finding the schedule, which delivers the the highest throughput of items in an order on a given configuration.

As Technomatix is more about altering parameters of current factory configurations, it is not a good fit when working with modular factories. In a modular factory system, the factory configuration may be altered in response to a new order of items. The problem is not only about finding the fastest schedule on an already implemented configuration. It also concerns looking at alternative configurations, and seeing if any of these have an even better schedule. We may also deal with the cost of moving from one configuration to the next. Even though one configuration may be very efficient, it may be expensive to reconfigure the current configuration in order to set it up.

2.2.2 Manual Software Reconfiguration

On the issue of reconfiguration, Aalborg staff informed us that the difficult part of setting up a new factory configuration is not the moving and connecting of the physical modules. Instead it is the alterations made to individual modules. As mentioned in section 2.1, each module has a set of instructions encoded in KRL code, which the MES4 system may signal for them to accomplish. As a factory is reconfigured this code will most likely have to be changed. This is no light task, as there must be an instruction for each case in which MES4 system may signal the module. If a red phone has to be treated differently from a black phone, there needs to be a separate instruction. Programmers have to handle each mundane case, leading to a lot of boilerplate code.

The staff here at Aalborg themselves, suggested that they would like a way to generate the KRL code. This problem is of course not only relevant in the case of FESTO. Any production line has to update its machine controlling code, if it reconfigures its layout.

2.2.3 Security

Like with the rest of Industry 4.0, modular factories rely heavily on networking technologies to interconnect every part of production. Through this network a lot of security and safety-critical data is send. According to Sadeghi[5], security is often not adequate and modular factories are thus at risk of infiltration from many different

angles. This is an issue, as an attack on a modular factory risks both monetary, but also human loss. Yet, it also seems that many modern security methods do not work for the modular factory. This is in part because a factory has a high requirement for availability. Thus a registered attack can not simply be countered by halting production. In addition, modules are often required to have low power consumption and minimal cost. Again, security features will clash with these requirements.

2.2.4 Big Data

Modular factories store a lot of data about their production. Each individual module, item and piece of equipment may be a source of information. Thus there is a vast amount of heterogeneous data to deal with. We must concern the field of Big Data, the challenges it brings along. Yin [6] mentions that the size of data itself puts significant requirements on both hardware and software. He also states that there is an issue of how to use the data for real time analysis and detection. Not all data is interesting either, so the cleaning of raw data has to be handled as well.

2.3 A Running Factory Example

In this section we set up an example of a factory, which we will refer to throughout the report. It can be seen on fig. 2.2. It depicts a specific factory configuration for producing various children's toys. We use this example to present some of the core concepts used in this report. In the final part of this report, we will try to generate this configuration, given only its modules and an order of items it needs to produce. In this section we will also discuss, what parts of the real world we choose to model.

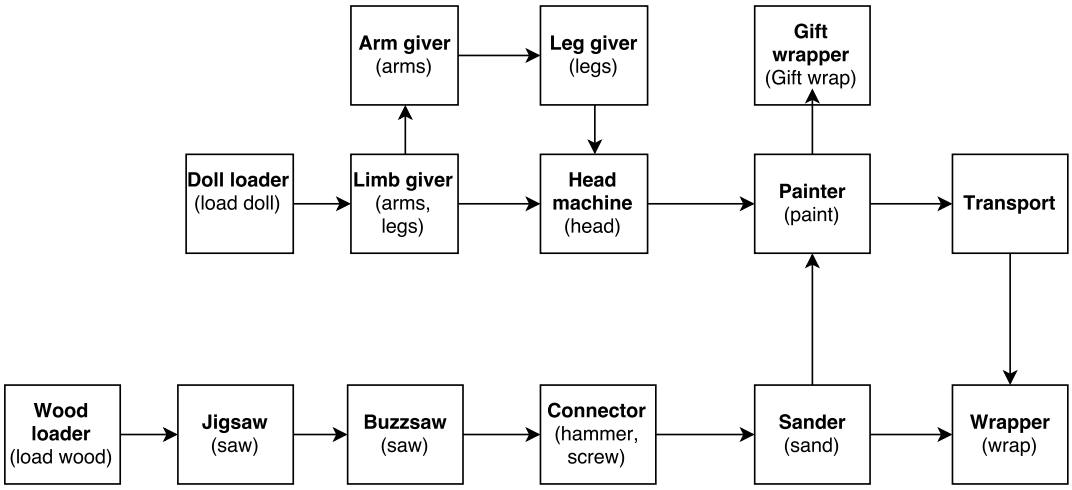

Figure 2.2: An example of a toy factory layout

2.3.1 Modules

On the diagram, we have chosen to depict a module as a square, which may connect to other modules. Each module has a name, shown in bold. A module may perform one or more types of work upon an item, these are shown in parenthesis. Each work type takes a certain amount of time to complete. It also takes a certain amount of time for an item to be transported through a module. A module may only work on one item at a time. These are all properties, which we discovered by inspecting the CP learning factory.

A connection signifies, how we may pass items between modules. There are real world physical restrictions upon these connections, in that each module may only connect to at most four other modules. At the same time, a module can only connect to another module, if they are placed right next to each other. On the diagram, we have chosen to move the modules away from each other to fit the arrows, but in the physical layout connected modules lay side by side. Yet, just because modules are placed next to each other, does not mean there needs to be a connection.

We choose to restrict connections in this way as modules tend to be rectangular in shape as in the case of the CP Learning Factory. We also allow only one neighbour per side, restricting us to four connections total. In reality the modules in the CP Learning Factory leave one side open for a control panel, so only three neighbours are allowed. We choose to ignore this requirement, as this may not be the case for all real life modules. CP Learning Factory modules also have two separate conveyor belts on each module. We instead imagine a single conveyor per module from which we may guide items to all sides of the module. In addition, we choose to view our modules as square, while the CP Learning Factory modules are only rectangular. Modules from different systems may have many different shapes, so we choose the one, which is easiest to work with, when we later have to deal with modeling the physical placement of modules. If we need to model non-square modules, we must add their properties to the rule set, which guides placement.

If we wish to connect two modules, which are not neighbours, we may connect them through one or more transport modules. This is a type of module that does not perform any type of work on items, other than ferrying them along a conveyor. On the diagram, a transporter is used to connect the *Painter* and *Wrapper* modules. Yet, using transport modules we can not freely connect any two modules, as the transporters have to be physically placeable. That is, they may physically not intersect any other module. This is the case when placing down any type of module. We view the factory layout on a 2-dimensional plane, where each module has the same height. This means that two modules intersect, if they take up the same space on this plane. We choose to disregard the third dimension, as modules tend to be of equal height.

2.3.2 Items

In our factory, various items may be produced. The top line is used to produce dolls, while the bottom is used to produce toys made of wood, such as wooden swords.

A doll needs to have its base loaded onto the factory. Then it needs to have arms, legs and head attached, followed by a layer of paint and finally a gift wrapping. This set of linearly dependent instructions is called a recipe. It describes, how we may create items of a specific type. Several identical items may be constructed to the same recipe. A recipe also indicates, the module at which items should begin production. In this case it would be *Doll loader*. To complete a doll according to this recipe, we may have the doll base worked on by the following sequence of modules *Doll loader*, *Limb giver*, *Head machine*, *Painter* and *Gift wrapper*. This sequence provides the types of work needed in the order specified by the recipe. Once an item has fulfilled its recipe it is removed from the factory.

A wooden sword may have a recipe, which says that it first needs to have wood loaded onto the factory. Followed by sawing, hammering, sanding, painting and finally regular wrapping. It needs to start in *Wood Loader*. To make a sword, we may work using the following sequence of modules: *Wood*, *Loader*, *Jigsaw*, *Connector*, *Sander*, *Painter* and *Wrapper*. Notice that the sword will have to use the transport module to get wrapped. We could also have connected the *Wrapper* directly to the *Painter*. However that would require wooden items that do not require painting, such as a rocking horse, to pass through the *Painter* module in order to get wrapped. The best placement depends on many factors such as the speed of individual modules, and how many wooden swords need to be produced in proportion to the amount of rocking horses.

2.3.3 Parallelism

Many items can obviously be worked on in parallel by the many modules of the factory. However we would like to focus on a specific type of parallel production, which increases throughput. We want items made according to the same recipe and having reached the same point in production to be worked on concurrently by different modules.

On the bottom line, we do not need both *Jigsaw* and *Buzzsaw* modules to produce wooden items according to the shown recipes. One is enough. However, in the case that we need to produce many wooden swords, it may be beneficial to have both modules, as we can then double the rate, at which we can saw our wood. This of course requires that wood may pass through the *Jigsaw* module, while it is sawing, in order to reach the *Buzzsaw*.

On the top line we have an off-branching line consisting of *Arm giver* and *Leg Giver*. Again, these are not required to produce dolls. Yet, by adding this branch, we allow for two doll bases to have arms and legs added at the same time. This increases the throughput of dolls. We may choose to parallelize execution in this manner, as opposed to the previous way, if we do not wish to lengthen the current

production line. It may also be used, if we want to parallelize a module's work, but we may not pass through that module, while it is working, as we can with *Jigsaw*.

Aalborg staff have not set up their factory to enable this type of parallel production. Yet they state that they would do so to increase throughput, if they had the additional modules to do so.

2.4 Problem Definition

This report started by bringing up Industry 4.0 in addition to the modular factory. This lead to the following initial problem:

How may it be possible to optimize the throughput of individual production lines in a modular factory system, while keeping down the overall cost of production?

Afterwards we looked deeper into the field of modular factories, using FESTO's CP Learning Factory as a starting point. From this we found a few challenges that modular factories are facing. Some relate to industry 4.0 as a whole, such as handling security and Big Data. Looking at the issues facing modular factories in particular, we see that the main problem lies in reconfiguration, as it requires a lot of resources. This is somewhat surprising, since modular factories are mainly based on frequent reconfiguration. In addition we have provided a running example of a factory configuration, and already discussed what properties of the real world modular factories, we wish to model.

For this project, we choose to concern ourselves with the issue of producing optimal factory configurations, given some order of items that needs to be produced and a set of avaliable modules. We define an optimal configuration to be the factory layout, which has the fastest schedule, and thus the highest throughput, among all candidate configurations. We formulate the goal of this project with the following project statement:

How may we, given some order of items and set of avaliable modules, be able to generate a factory configuration which has the fastest schedule of any other candidate configuration?

This problem is solved in two parts. The first will concern itself with the actual modelling and simulation of a modular factory. This should allow us to simulate any factory configuration, and allow us to generate its fastest schedule for producing a given order. We rate a configuration according to the time taken to execute its fastest schedule. The second part of the solution will concern, how we pick the optimal configuration by generating candidates and comparing their ratings, as to identify an optimal choice.

Chapter 3

UPPAAL model

In this chapter, we describe, how we model modular factory configurations. In addition, we will explain, how we may rate a configuration by producing its fastest schedule through simulation. We want to be able to model and simulate configurations, so that we may compare the ratings of many different configurations. This would not be practical, if we had to set up each configuration physically.

To model modular factories and simulate individual configurations, we use the UPPAAL model-checker[7]. This is an integrated development environment used for modelling, simulating and verifying real-time systems. It allows us to model our system using timed automata in a combined graphical and programmatic manner. Once we have designed our model, we may use it to instantiate a configuration and a set of items. UPPAAL can then simulate the production of these items on the given configuration. In addition, the model-checker included in UPPAAL, allows us to query different properties about the setup. To us, the most interesting property is one of reachability; *Is it possible to reach a state, where all items in an order have been produced?* If this is the case, then UPPAAL may produce a fastest timed trace of transitions needed to reach this state. This sequence of system states evaluates to the fastest schedule for the given configration. Looking at the global clock of the final state in the trace gives us the time taken to execute the trace, and thus our configuration rating.

UPPAAL has been used to model many different real-life systems, such as a lacquer[8] and an industrial printer[9]. From this research we take inspiration for our model. In these examples however, UPPAAL is used to model systems with only a single configuration. Our work differs in that we attempt to model a system of which there are many possible configurations, which need to be compared. Therefore we focus on keeping the module flexible and avoiding hard-coded values. Thus, many different configurations and items may be instantiated.

When modeling the real world, we need to take care to pick an appropriate abstraction level. Too high, and the model will not fit well with the real world. Too low and the model becomes very complex. The more complex a model is, the more likely it is that the search space will be large. This severely increases the runtime,

when searching for a best schedule. Therefore, we naturally aim to hit a sweet spot between abstraction and complexity.

In this chapter we will explain how we modeled configurations and items in UPPAAL, so that we may simulate production. This will be a look at both the technical implementation, but also a discussion on when we have abstracted from the real world, and when we have not. We will also extend the design decision made in section 2.3.

3.1 Item

As mentioned in section 2.3 a group of identical items to be produced by a factory are described by a recipe. This is a linear sequence of work, we need to perform on an item to complete it. Yet, we realize that a recipe need not be described as linear. In fact, in some cases we do not care about the order of work.

Let us look back to the linear recipe, describing how a doll should be produced on fig. 2.2. It states that the doll must have arms, legs and head attached in the given order. Yet, we may not care for the order in which we attach limbs. Therefore we model a recipe not as a sequence, but an acyclic dependency graph as seen in fig. 3.1. If node A has an edge going to another node B, A is said to depend on B. In our case, it means that the work, which B represents, should be accomplished before A.

The doll recipe now states that after a doll base has been loaded onto the factory, we may attatch arms, legs and head in any order. Yet we may not paint the doll before all limbs have been attached, and we can not gift wrap the doll before it has been painted. This lets us set up more complex recipes that we may produce items according to.

Since the graph is acyclic, we may not model items, where their recipes require the same work performed twice. This may happen in the real world, yet we look past it, as many types of recipes pass this restriction. As an example, while the inspected CP-learning factory configuration has physical cycles in its layout, but no item produced on it is made according to a recipe that requires the same work performed twice.

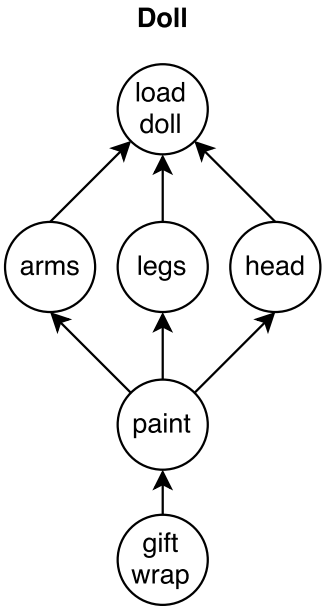

Figure 3.1: Acyclic dependency graph describing order of actions

Neither do we model items, which distribute their production. This means that an item may have different parts located on different modules at the same time. Going back to the doll, this would allow us to paint the arms, legs head and base of the doll before putting it all together. To simplify our model we look past this. Thus, an item can only be located on a single module at one point in time. At the same time, no recipe may require that an item begins production from more than one specific module. At the local CP-learning factory we see no distributed production of any item. Therefore we can still describe production of real life items by modeling a recipe with an acyclic dependency graph.

In UPPAAL we may simulate the process of production by composing several smaller processes to run in parallel. Each process is instantiated through the use of a template. Templates are similar to classes in object oriented programming. Yet they consist not only of local variables and functions, but also a graphically designed timed automata. When a template is instantiated it evaluates to a process defined by initial parameters as well as the initial location of the timed automata. The restrictions set up by local variables, functions and timed automata describe how this process may transform into other processes.

To simulate production on a configuration, we must have several processes running in parallel. Some of these processes are of different types, which means that we need to set up different templates. In the following subsections we describe, how we model items in UPPAAL with the *Item* template, so that we may simulate their production. In addition, we describe how we enforce the order in which items begin production using the *ItemQueue* template.

```
typedef struct {
        wid_t work;
        int number_of_parents;
        int children[NUMBER_OF_WORKTYPES];
        int number_of_children;
} node;
```

Code 3.1: Node struct

3.1.1 Item Template

We could choose to implement a new template for each recipe, using it do instantiate items, which are described by that recipe. This would require no local functions in the template, as we could just translate the acyclic dependency graph, which describes the given recipe, to a timed automata. However, as revealed ichapter 5, we want to instantiate our configurations in UPPAAL through python. To do this we alter the XML file, which describes our templates as well as details on the configuration to be run. Because of the structure of this file, it is easier to change local functions and variables of a template than the graphically designed timed automata. Therefore, we instead create a single template *Item*, which is used to instantiate every item process. This template takes a recipe as an input parameter, to describe the work that the item requires.

A recipe is given as an array of nodes. For this we have developed a *Node* struct as shown in code 3.1. A node knows what type of work it represents, the number of parents it has, its children's indexes in the node array and the number of children. Node A is parent of node B, if B depends on A, and the other way around for children. In addition to the recipe, we also give each item a unique id and the id of the module at which it begins production.

The timed automata within the *Item* template can be seen in fig. 3.2, initial location marked with a circle within a circle. From the initial location we may go to the *InProgress* location, by placing the item onto its start module. In addition the transition requires a call to made to *get_upper_nodes*. This instantiates the local *current_nodes* array to contain all nodes in the recipe, which do not depend on any other node. These describe the work, which we may initially be performed.

After this, the item is moved along the factory. Each time it meets a module, where it wishes to have work done, it will handshake on its own private channel with the module to identify itself. Afterwards, it will synchronize with the module again on one of the allowed *work* channels. A work channel can be synchronized on, if one of the nodes in the *current_nodes* array, represent the work corresponding to that work channel. This check is done by the *is_callable* function.

Before going back to the *InProgress* location, a call to *update_current_nodes* is made. The code of this function can be seen in code 3.2. It will first collect all current nodes in the *new_nodes* array, except for *called_node*, as this is the

node just worked on. It then runs over each of *called_node*'s children to decrement their *number_of_parents* field by 1. If this field reaches 0, it means that the node no longer has any dependencies and can be worked on. Because of this, it is added to the *new_nodes* array. Once all children have been updated, the contents of *new_nodes* are used to update *current_nodes*. Thus we update the work, which we may perform according to the recipe.

After the call to *update_current_nodes* is finished, we call *no_more_nodes*. This will set the local *done* boolean to *True*, if *current_nodes* nodes is empty. This means that the item can not be worked on any further. It has now been produced and may be removed from the factory.

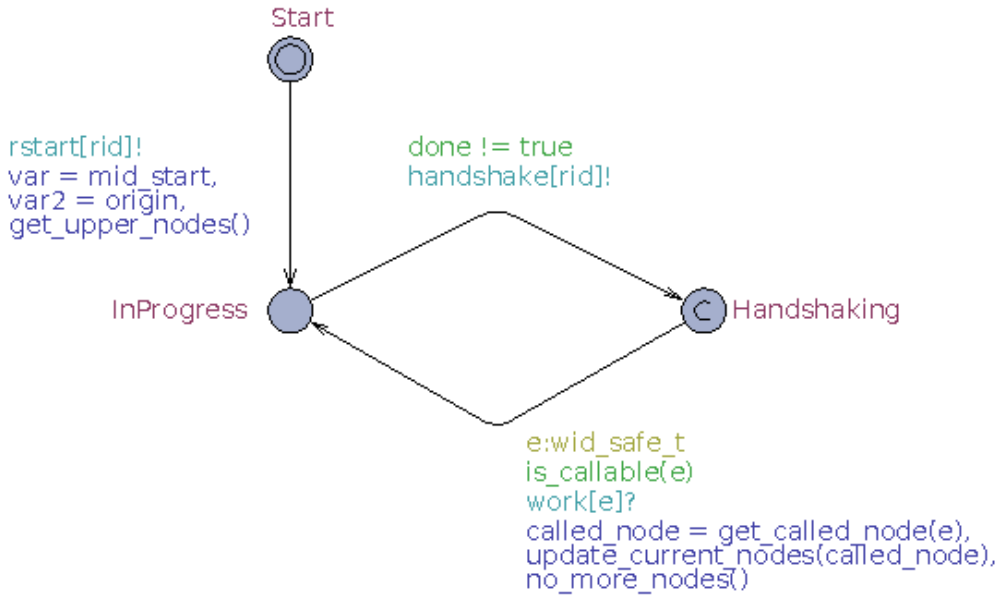

Figure 3.2: *Item* template

3.1.2 ItemQueue Template

Having all item processes compete for a place on the factory at first, leads to a large state-space. This must be searched, when we ask UPPAAL to find the shortest timed trace. In order to get around this, we enforce a certain order, in which items may be placed onto the factory. This is done using a queuing system, which we implement using the *ItemQueue* template as seen in fig. 3.3.

The template is instantiated with an array of item ids, indicating the order, in which we wish the items added. Once instantiated, the queue may begin dequeuing items, so that they may be placed onto the factory. This reduces the overall state space, as items are to begin in a specified order.

```
// Updates the current_nodes array,
// to reflect that work has been done.
void update_current_nodes(){
    node new_nodes[length];
    int i;
    int j = 0;

    current_works[rid][called_node.work] = false;

    // Collects all elements of current_nodes,
    // except for the node just worked on.
    // Decrements the size of array by 1
    for(i = 0; i < c_length; i++){
        if(current_nodes[i] != called_node){
            new_nodes[j] = current_nodes[i];
            j++;
        }
    }
    c_length--;

    // Finds all children on the node that has been worked on
    // and decrements their number of parents by 1
    for(i = 0; i < called_node.number_of_children; i++){

        node child = func_dep[called_node.children[i]];
        int res;
        res = --child.number_of_parents;

        // If child has lost all it's parents :-(
        // It is added to the array of current nodes
        if(res == 0){
            new_nodes[c_length] = child;
            current_works[rid][child.work] = true;
            c_length++;
        }
    }

    // Overwrite the old current array with new one
    for(i = 0; i < c_length; i++)
        current_nodes[i] = new_nodes[i];
}
```

Code 3.2: The update_current_nodes function local to the *Item* template

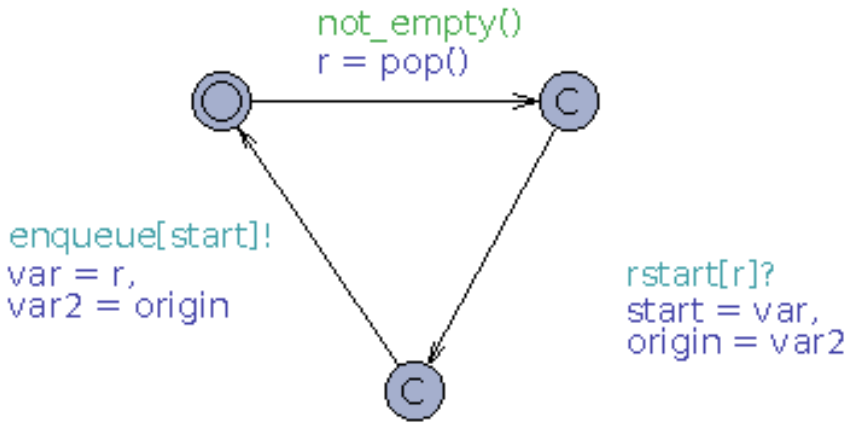

Figure 3.3: *ItemQueue* template

3.2 Module

As a configuration is made up of individual modules, it seems natural to model it as a set of synchronizing module processes. In this section we will first show off the initial design of the *Module* template. Then discuss, how we got more concrete with the final version.

3.2.1 Early Module Template

In our early design work, we tried to implement most everything brought up in section 2.3. A module has an identifying name, and it may only work on one item at a time. It also takes a certain amount of time to perform work and to transport an item across it. However, we start by allowing modules to only perform one type of work. There are also no physical restrictions, on which modules may be connected, other than the fact that a module may not be connected to more than four other modules. These basic ideas turned into our first version of the *Module* template, which can be seen in fig. 3.4.

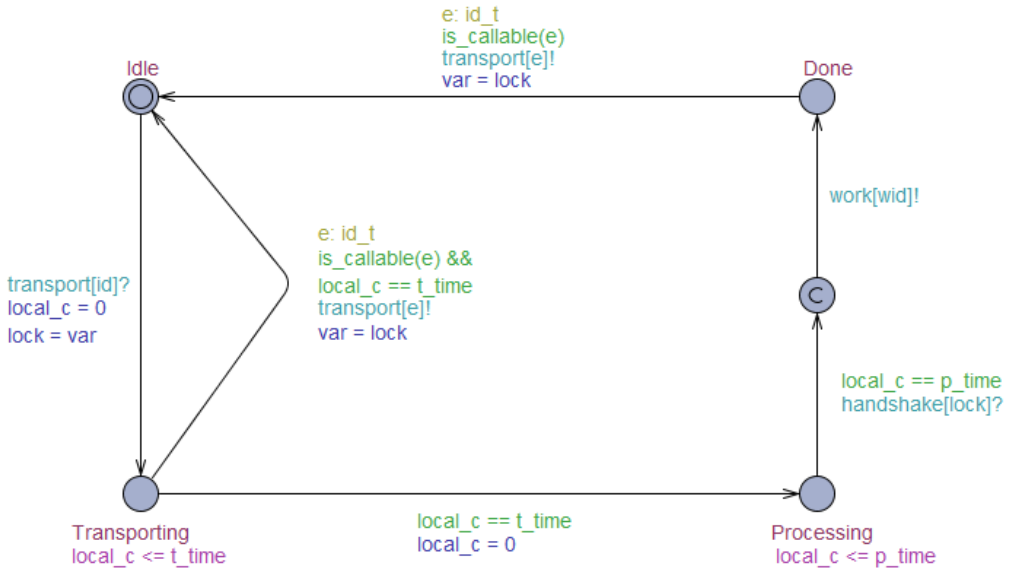

Figure 3.4: Early version of the *Module* template

A module starts in the *Idle* location and resides there, when not processing an item. It leaves this location, when one of its up to four neighbouring modules passes on an item by synchronizing on the *transport* channel identified by the modules's id. Once an item has been received, the module waits in the *transporting* location for *t_time*. This simulates the time taken to transport the item across the module. Once the time has passed we may send the item to a neighbouring module. The local guarding function *is_callable* makes sure, we only send to neighbours. This set-up allows us to implement modules, which are only used for transporting item.

Instead of passing an item along, we may perform work on it by moving into the *Processing* location. Here we wait for *p_time*, to simulate the time it takes to work the item. Before we can update the item to have its work performed, it must identify itself. This is done by synchronizing on the *handshake* channel given by the value stored in the local *lock* variable. This variable contains the unique id of the item, which most recently entered the module. It was received from the previous module over the global *var* variable. This ensures that after a handshake, moving us to a committed location, the synchronizing on the *work* channel will be with the correct item. Once work has been performed, we may pass the item onto another module.

Some modules may also allow for items to be removed from the system by transporting on the reserved *transport[0]* channel. No actual module has the id 0. Instead a process instantiated from the *Remover* template is constantly calling on the *transport[0]* channel. If a synchronization is established the item is removed from the module and not passed on to any other module. A module can only synchronize with the remover process, if it has one of its neighbouring modules set to 0.

3.2.2 Final Module Template

From further observations of the local CP Learning Factory configuration, we conclude that the above implementation is too abstract. In an actual factory, each module may perform several different types of work on an item, as stated earlier. While only one item is worked at a time, several items may queue up on the module, waiting to be worked on or pass through. A module has a fixed limit of items, which may queue up. This is in accordance with the safety property of the system. If this is not upheld, items may fly off the modules, or the modules themselves may be damaged. Some modules do not allow for items to pass through them, while they are working on another item. This feature is not enforced in the above implementation. In addition, the time to transport an item across a module depend on the dimensions of the module as well as the directions, from which a item enters and leaves.

These features lead to some drastic changes in the module template. The biggest is that the template now gets split into three. *ModuleQueue*, which controls the enqueuing and dequeuing of items. Then *ModuleWorker* which performs work upon a item. Finally *ModuleTransporter*, which transports items between modules. To get a complete module, we need a process of each of these templates sharing the same module id. In the following we will explain each of these new templates.

One aspect that we ultimately do not try to model in UPPAAL are the more extensive physical rules for placing down modules. We choose to enforce these later, when generating and comparing configurations.

ModuleQueue Template

One of the most important characteristics observed was the queuing nature of the system. Several items may be located on a module, but only one is being processed at a time. Some of the items queued may need to be processed, others just want to be passed along to a neighbouring module. Based on this behavior, we construct the *ModuleQueue* template, which can be seen on fig. 3.5. A process of this template has a fixed size of modules, which it may hold. Other modules may be able to use the *enqueue* channel to move an item onto the module. In addition, items may be removed from it through a synchronization on its *dequeue* channel.

As in the earlier version, an item is moved between instances by proxy of its id. The id is always stored in the local *lock* variable.

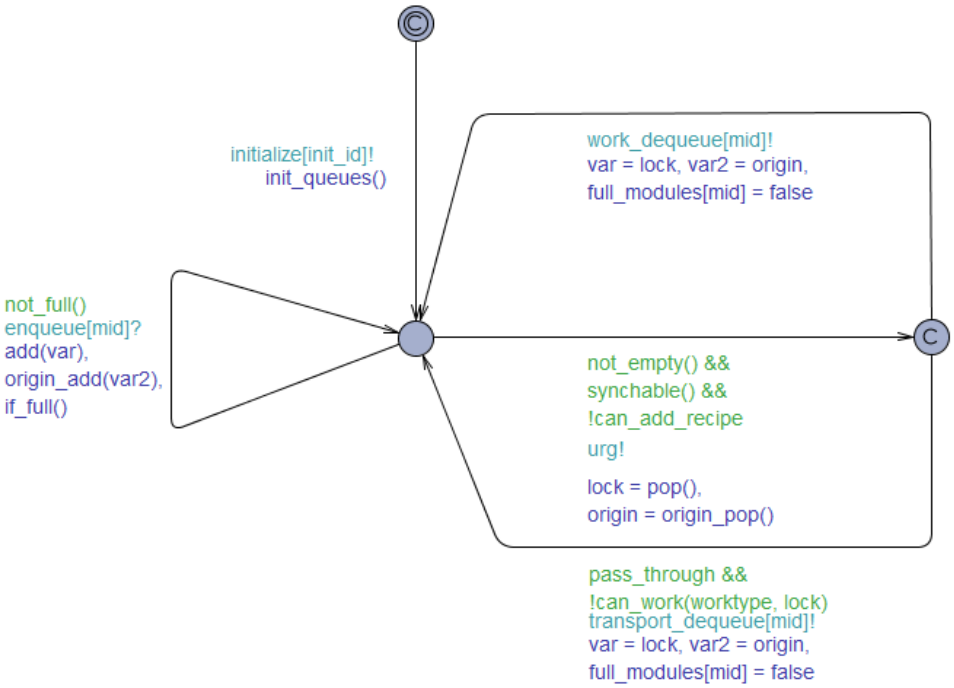

Figure 3.5: The *ModuleQueue* template

ModuleWorker Template

The *ModuleWorker* template can be seen in fig. 3.6. An instance of this template may apply several different types of work upon an item. In the *Idle* location, the worker waits to synchronize with the module's *ModuleQueue* instance allowing it to enter the *Done* location. If the module can perform some work, it will move into the *Working* location. Here it will wait for *p_time*, symbolizing the time it takes to perform the work. When the wait is over, it will try to synchronize with the item through the *handshake* and *work* channels as in the earlier version. Arriving at the *Done* location, we may choose to work the item further if possible. Otherwise we perform a synchronization on the module's *intern* channel. This passes the item over to the module's *ModuleTransporter* instance. Synchronization on the intern channel may also occur, if the module can not perform any work on the item. This models the case, where an item has to pass through a module without having worked performed, while the model does not allow for pass through while it is working on other items.

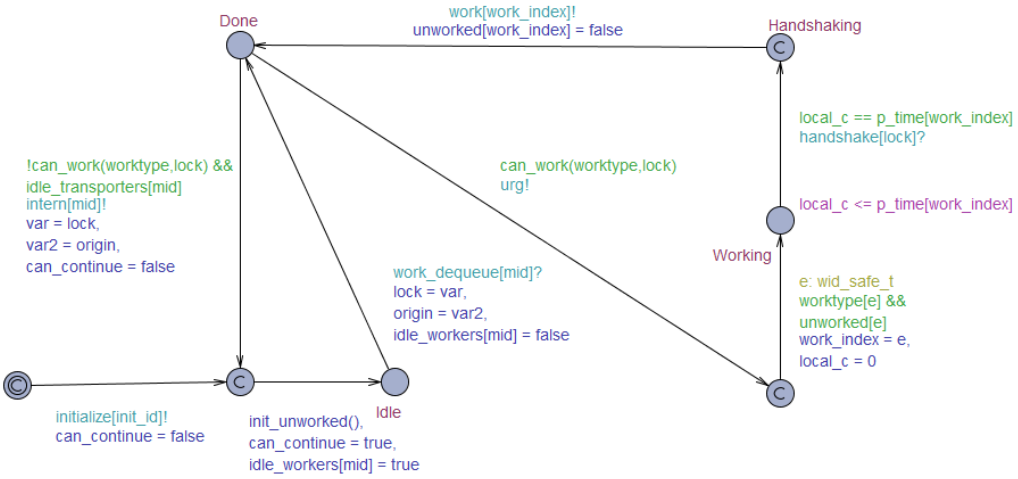

Figure 3.6: The *ModuleWorker* template

ModuleTransporter Template

The *ModuleTransporter* template can be seen in fig. 3.7. A process of this template allows a module to transport an item onto a neighbouring module. If the item comes from the module's *ModuleWorker* instance, then we move into the *Selector* location by synchronizing on the module's *intern* channel. We may however also move directly to the *Selector* location by synchronizing with the module's *ModuleQueue* instance over the *dequeue* channel. This can however only be done, if the fig. 3.7 module allows us to pass an item through a module, while it is working on another item. This is indicated by the boolean *pass_through* variable, which is given at process instantiation. To create a module only used for transportation, we set this variable to *true* and do not create a *ModuleWorker* instance for the module.

Once in the *Selector* location a item may go to *Idle* or *Transporting*. If the item is done, it has to go back to *Idle* and be removed from the production line by synchronizing on the *remove* channel, as opposed to the *transport[0]* in the earlier version. This also means that we do not specify a specific module to remove items from, they are just removed when done.

If the item is not done, it may look for a neighbouring module to be passed onto. A module may have up to four neighbours, one on each side. These are stored in the local *next* array, their index indicating their placement relative to the module. When moving to *Transporting* location, the item will choose a possible neighbour. In *Transporting*, we wait to simulate the time taken to transport an item over the module. This time varies according to where the item enters the module, and where it leaves. The different transport times are stored in the 4 by 4 multidimensional *t_time* array. Given that we at this point know the direction, which the item entered from *origin*, and the direction on which it will leave *succ*, we can look up the exact

time to wait in *t_time*. By implementing this feature, we ensure that the time take to pass over a module becomes more accurate to how an item may actually be transported over a module.

Once the wait is over we move to the *Queuing* location. If the neighbour's queue is full, we wait here until there is room. When possible we synchronize with the neighbours instance of *ModuleQueue* using the *enqueue* channel. At the same time we use the local *inverse* function to calculate, from which side the item enters the neighbouring module. This is sent with the global *var2* variable and is later saved into the *origin* variable of the neighbour's *ModuleQueue* process.

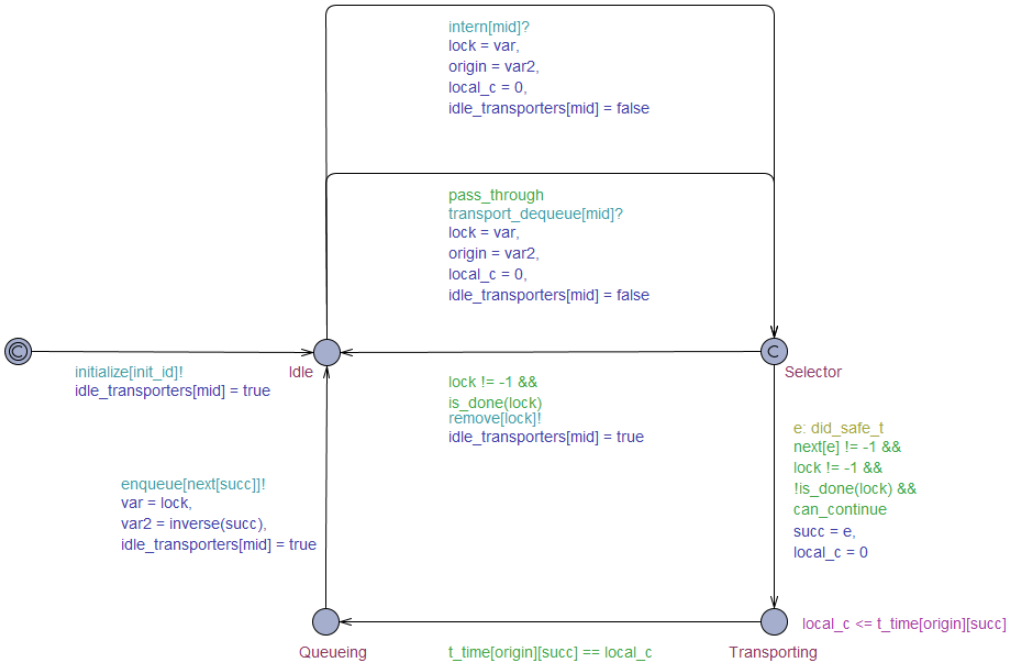

Figure 3.7: The *ModuleTransporter* template

3.3 Efficiency

Section 3.1.2 illustrates how we make search for the shortest timed trace more efficient, by ordering how items are placed onto a factory. By applying such order, we are trying to keep down the global state space that needs to be searched, like how the bourgeois tries to keep down the proletariat. The lower the state space, the faster the search for the shortest timed trace.

This is not the only part of the model, where we combat the state space. We have strayed from going deeper into this until now, as it does not impact the overall

functionality of the model. Yet it does greatly impact runtime. In the following subsections, we explain, how we have decreased our state space.

3.3.1 Urgency

The biggest positive impact on the state space has come through the use of committed locations and urgent channels.

When a process enters a committed location, it ensures that globally no transition may be taken and no time may pass before the process leaves the location. This ensures that some processes stay atomic, such as handshake and work on item processes as described in section 3.1.1. This guards us from the interference of other processes, but also greatly reduces the state space, as we can not wait by making timed transitions in a committed location.

Urgency is an attribute that can be applied to different channels. Having a channel be urgent means that no time may pass, if we can transition from one process to another by synchronizing on the channel. In *ModuleWorker* on section 3.2.2 the *Intern* channel is urgent. This means that, when the guards evaluate to true, then we transition by synchronizing on *intern*. Again, not allowing the system the possibility of waiting reduces the state space.

There are transitions going between locations that we wish to make urgent, such as from *Done* to *Working* in the *ModuleWorker* template, where there is not already some way of synchronizing on a channel. We solve this by creating a new simple template called *Urgent*. A process instantiated from this will continuously try to synchronize on the urgent *urg* channel. Thus we can add a synchronization on this channel to any non-urgent transition to make it urgent.

A big difficulty with working with committed locations and urgent channels is that we sometimes need to be able to wait. This is the case when passing from *ModuleWorker* to *ModuleTransporter* through the intern channel as mentioned above. We can not safely make *intern* urgent, if we may synchronize on it from *ModuleWorker* but not *ModuleTransporter*. This will lead to unwanted deadlock. Therefore, the guard on this particular transition states that it must only be taken, if the *ModuleTransporter* is in idle and can communicate. We use this technique throughout the model, applying guards so that we may safely make our channels urgent.

3.3.2 Priority

In order to shave more off the state space, we have also create priorities between channels using the built-in UPPAAL priority feature. This can be seen in code 3.3. By placing this order, we do not allow the model checker to synchronize on a certain channel, if a channel of a higher priority may be synchronized on. This helps guide us when there is an ambiguous choice of which channel to communicate on.

However, if we set up the same configuration in a version of the UPPAAL model without priorities and one with, then their initial parallel processes will not be bisimilar. This is because the model with priorities is restricted from transitions that the

```
chan priority transport_dequeue < work_dequeue < \
                              intern < handshake < work < enqueue < \
                              default < rstart < remove < urg;
```

Code 3.3: Channel priorities

un-prioritized model may take. Yet, we find that this does not impact the fastest schedule found between the two different set ups. By design of the templates, none of the processes that make up the larger parallel system process will be able to make another transition, if they are waiting to communicate on a channel. Because of this, if two processes waiting to communicate are bypassed because of prioritization, they will stick around and wait. Resulting from the nature of the system, they will be able to communicate before the global clock is counted up, which only occurs as the result of transport and work on items. Because of this, our shortest time schedules end up having their final global clock values be equal, whether we prioritize or not.

An issue is that even with channel priorities, we may have two transitions communicating on the same channel type, which again leaves us with an ambiguous order of execution. To get around this, we also prioritize system instances. If we ever get into the above situation, we pick the transition involving instances with the highest priorities.

3.3.3 Initial Order

A feature which UPPAAL templates lack is a user-defined constructor, which would allow us to execute code right when a process is made. We need this feature in templates such as *ModuleQueue* in section 3.2.2, where we need to set up an array implementing the local queue. To get around this, we add an initial location to the template. To transition from this location, the instantiating functions have to be called.

The issue here is that with N processes having to be instantiated in this manner, there are N! ways of performing instantiation. This greatly increases our state space. To get around this, we use a queueing system similar to *RecipeQueue* in section 3.1.2. This is implemented with the *Initializer* template as seen in fig. 3.8. Each process, which needs to have code run upon instantiation, is given an instantiation id. When creating a processes from the *Initializer* template, we give it an array, which orders all these ids. The *Initializer* process then starts to instantiate the instances in the order given to it by synchronizing on the *initialize* channel. As the *Initializer* process's main location is committed, nothing may occur until all needed instances have been synchronized with. Once all instantiations have been done, the *Intanializer* is forced to move into a dead state, and we need not to worry about it anymore. From this point on the remaining processes can transition as before.

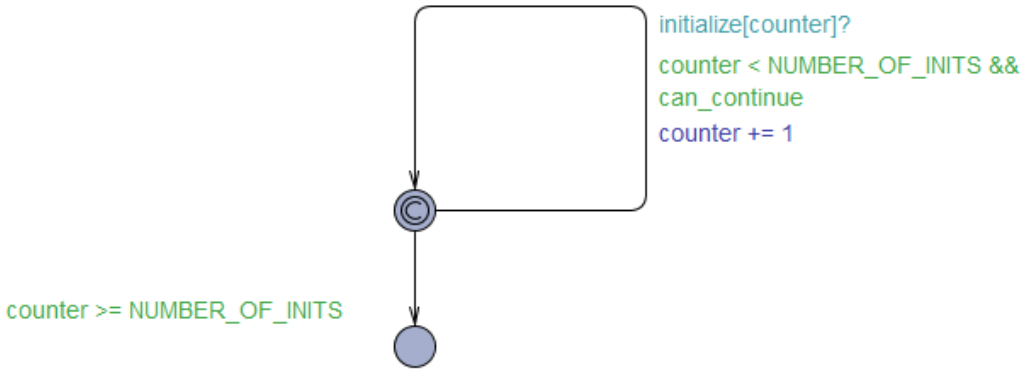

Figure 3.8: The *Initialzier* template

3.4 Evaluation of Model

Having now defined all needed templates as to model a factory configuration in UPPAAL, we would like to evaluate our design. We will evaluate, how well our model is capable of emulating reality. This is done by comparing the time it takes for a real life factory configuration to produce an order items, to the rating we give the same configuration when simulated in UPPAAL.

3.4.1 The Model Compared to Reality

In order to compare our model to reality, we set up the configuration of the actual CP Learning Factory mentioned earlier using our UPPAAL templates. We then run different orders on both the actual factory and the model. Afterwards, we compare the time it takes for both to produce their orders.

The configuration, which we wish to simulate, is used to produce faux smartphones. 3 different recipes were set up, which described different types of smartphones that the configuration may produce. These can be seen in fig. 3.9. Each are named after, how many fuses are put into them.

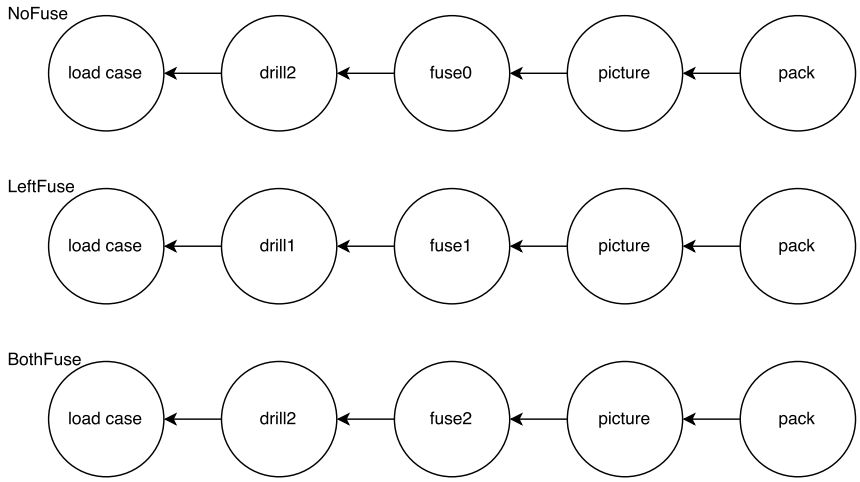

Figure 3.9: Dependency graphs representing the three recipes the CP-Factory could produce

Inspecting the configuration, we find the different modules and the types of work that they may perform, as well as how they connect. This is shown in fig. 3.10. We choose to disregard that these modules each have two conveyor belts, as both are not used by any item produced. We also disregard the *Transport1* module, as it is not used in the production of any item.

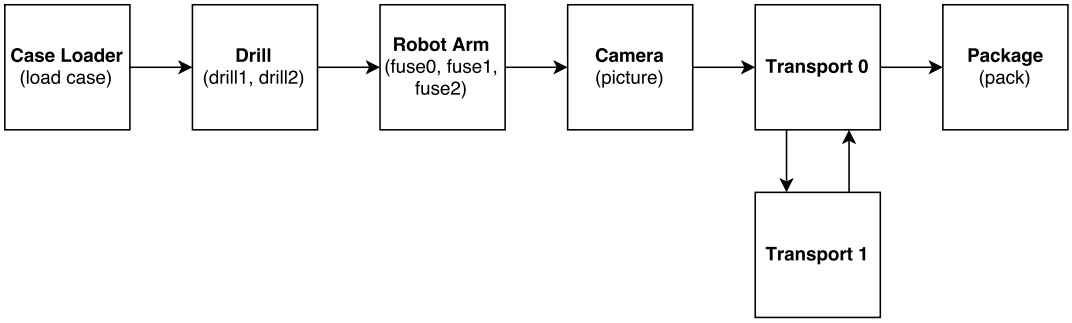

Figure 3.10: Graphical representation of the CP-Factory configuration at Aalborg University

Each item is placed on the configuration in the *Case Loader* module and are removed at the *Package* module. We were allowed to try and produce some items using the factory. Here we timed how long it takes for items to pass over modules, and how long it takes for certain works to be done. Averages of these times are listed in table 3.1.

In total we ran four different orders on the configuration. On table 3.2 each order is defined by the item types it needs to produce, as well as the amount of each type required.

Module Name	Transport Time	Work Time
Case Loader:	11.7	load case: 6
Drill	10.7	drill1: 5.3 drill2: 10.6
Robot Arm	16.4	fuse0: 58.2 fuse1: 75.2 fuse2: 85
Camera	11.2	picture:2.0
Transport	11.2	/
Packaging	0	pack:6.8

Table 3.1: For each module, shows time taken to transport through it and time to perform each of its works. Time is given in seconds.

Order Name	Description
SingleNoFuse	NoFuse: 1
SingleLeftFuse	LeftFuse: 1
SingleBothFuse	BothFuse: 1
AllTypes	NoFuse: 1 LeftFuse: 1 BothFuse: 1

Table 3.2: Describes each order by the item types to produce as well as the amount of each.

With all this in place, we are able to set up the configuration in UPPAAL using our templates and running each of the four orders on it. In appendix A is shown how we set up the system to run the *AllTypes* order. Please note that the *Item* template is here called *Recipe* and similarly *ItemQueue* is called *RecipeQueue*. The variables *recipe0*, *recipe1* and *recipe2* refer to the 3 processes, which represent the three items that need to be produced for the *AllTypes* order. At the bottom of the example, we see how we parallelize all the smaller processes into one big system process. For this case the following reachability query is given to the UPPAAL model checker:

$E <>\ recip0.done\ and\ recipe1.done\ and\ recipe2.done$

It asks: *"From the initial state, can we reach a state where all three items have been completed"*. A similar query is used for the three simpler orders. In all cases it evaluates to true. Because of this we can produce the shortest timed trace for each order, and on the last state read the value of the global clock. This is the configuration's rating given the specific order, which describes how long it takes to run the fastest schedule. Having extracted this from each of the four orders, we compare them with the time it takes to complete the orders on the actual factory configuration. The results of this can be seen in table 3.3.

Order Name	Actual	Simulated	Difference
SingleNoFuse	144.8	144.9	0.1
SingleLeftFuse	156.5	156.6	0.1
SingleBothFuse	171.6	171.7	0.1
AllTypes	305	311	6

Table 3.3: Comparison of actual and simulated times. Time is in seconds.

Looking at the results, we are almost spot on for the very simple orders. Yet the simulated and actual time drift a bit apart as the orders becomes more complex, though still to a small degree in our case. These results will be discussed further in section 6.1.

Chapter 4

Transformation of Configurations

We have now implemented a UPPAAL model, which we can use to find the shortest time it takes for a configuration to produce a given order. We wish to use this tool in order pick the optimal configuration to perform some kind of order, when we have access to a given set of modules. In this chapter we will formally describe, how we from a simple initial configuration may produce other configurations. Being able to generate different configurations, we are able to compare them, and pick the candidate with the best rating.

4.1 Naive Example

Looking back at fig. 2.2, this example was created by human hands. In its design there is an attempt to maximize throughput by creating seperate lines for different types of items and to increase throughput by using additional modules. Now, how would a computer given a set of modules and an same order arrive at the same configuration?

To start off, we need some initial configuration, from which we can start the search towards the optima. We choose this configuration to be naively constructed, as it will be easy to generate. Let us say that we are given the same modules as in fig. 2.2. In addition, we are supposed to produce three types of items, a doll, a rocking horse and a wooden sword as described in fig. 4.1. Obviously, these items can be produced by the running example. The same is true for the naivley constructed configuration shown in fig. 4.2.

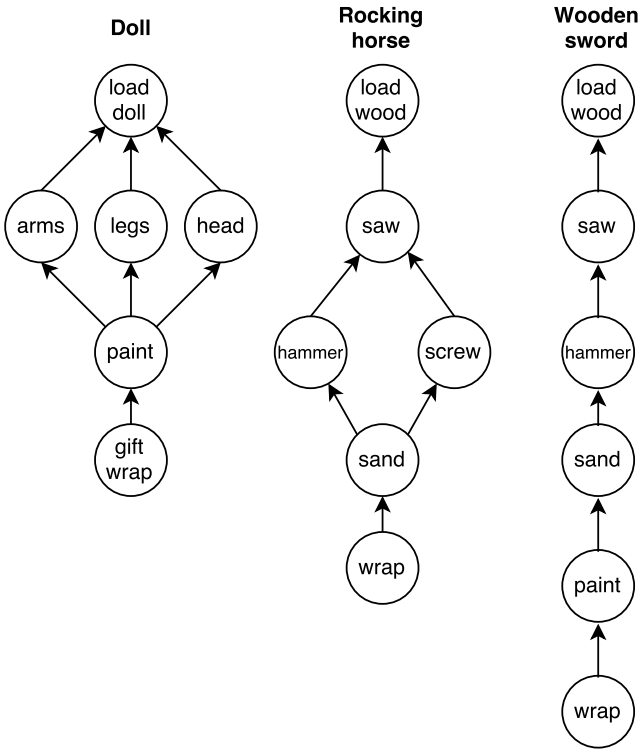

Figure 4.1: Acyclic dependency graphs describing three item types.

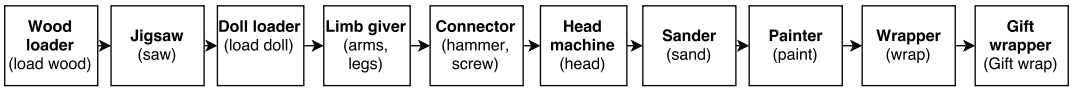

Figure 4.2: A trivial configuration of a toy factory layout

This naive configuration is obviously a poorer choice. In the man-made configuration no item has to pass through a module, where it is not worked upon. At least, when excluding the transport module. We originally got around this issue by producing items on separate lines. In comparison, the trivial configuration is made of a single line.

In addition the trivial example uses the minimal amount of modules needed to process our orders. This means that it does not try to increase throughput by adding additional modules to the configuration.

Yet, the advantage of the trivial example is that it is very simple to generate using a computer. This can not be said for the man-made configuration. To produce the naive example, we have to compose the three graphs in fig. 4.1. On the resulting dependency graph, we then perform a topological sort. This gives us an ordered list of works, which we can then use as a blueprint to place down modules in a line, connecting them from left to right.

A dependency graph may of course have several topological sorts. Because of this we could just generate a configuration for each of these sorts. Each of these candidate solutions can then be set up in our UPPAAL model, and we can find the execution time of their fastest trace. The configuration with the fastest trace is then deemed the best fitting candidate. Yet, again this configuration would most likely still be poor compared to the one we created ourselves.

Because of this, we want to define ways in which we from a naive single line configuration are able to transform into more complex configurations, which may be better at producing the given order. In the rest of this chapter we will formally define tranformation rules, which allow us to do this.

4.2 Formal Configuration Description

In this section we formally describe the parts making up a configuration. We do this by first describing the keystones of a configuration, namely recipes, modules and lines. We then describe a configuration as a tuple of sets. This tuple is what we can perform transformations on in later sections.

---

definition 4.1 Basic elements in a configuration

recipe : *Set of all work required to complete a concrete recipe.*

module : *Set of all work that a module may perform on an item*

line : *Set of modules totally ordered on the relation* $\prec$

In definition 4.1 we define some basic elements, which make up a configuration. We treat a recipe as a set of works, since our initial configuration will always be made up of a single line of ordered modules able to complete all given recipes. As none of the transformation rules described later will change this order, we do not need to enforce an order on the works that make up a recipe. A module is described as the set of works that it may perform on an item. A line is a set of modules totally ordered on $\prec$. The relation orders modules according to their placement in the line from left to right. If $m_1 \prec m_2$ for modules m_1 and m_2 in a line, then there is a horizontal path from m_1 to m_2 traveling rightwards.

In definition 4.2, we have defined a factory configuration as a tuple made of sets and functions. R and M are never changed by our transformation rules. Thus they are constant, going from an initial configuration forwards. As stated before, each configuration starts as a single initial line. For an initial configuration, Γ_0 represents this line. In subsequent configurations Γ_0 represents a special line, from which a subset of our transformations will branch out production.

AW is a function, which for a module $m \in M$ returns the set of works, which

definition 4.2 Formal definition of a configuration

$Configuration : (R, M, \Gamma, \Gamma_0, AW, Start, End)$
Where:
R : A set of recipes
M : A set of modules
Γ : A set of lines
Γ_0 : Main line
$AW : M \to \bigcup_{r \in R} r^2$
$Start : \Gamma \rightharpoonup M$
$End : \Gamma \rightharpoonup M$

m performs actively on at least one recipe $r \in R$. *Start* is a partial function, which maps a line $\gamma \in \Gamma$ to a module $m \in M$ such that $min\gamma'$ where $\gamma' \in \Gamma$ and $\gamma \neq \gamma'$. This represents that there is a vertical path between m and the first element of γ. Similarly the partial function *End*, which maps a line $\gamma \in \Gamma$ to a module $m \in M$ such that $min\gamma'$ where $\gamma' \in \Gamma$ and $\gamma \neq \gamma'$. This represent that there is a vertical path from the last element of γ to m.

Furthermore we only want each *module* to appear in a single *line*. To enforce this we set up the following rule for each *line* γ, in Γ:

$$\texttt{if } \gamma \in \Gamma \texttt{ then } \forall m \in \gamma \wedge \forall \gamma' \in \Gamma \wedge \gamma \neq \gamma', m \notin \gamma'$$

We also set up the following rule for each *module* m in M. This is done to ensure that the module can not perform work not within a recipe, and that the work which it actively performs should be a part of the total set of works:

$$\texttt{if } m \in M \texttt{ then } AW(m) \subseteq m.mW \wedge m.mW \subseteq \bigcup_{r \in R} r$$

In the following subsections we will describe a set of transformation rules, which are a set of functions, each mapping a configuration to another configuration.

4.3 Anti-Serialization

In this section we will describe a set of transformation rules known as Anti-Serializations. In fig. 4.2 all items made according to the three recipes in fig. 4.1 will have to pass through modules, which do not work on them. We would like to minimize this as to reduce the total length that each item must be transported, in addition to combating bottlenecks in the line. Thus it would be beneficial if items could bypass modules that do not work on them. We get around this by branching out production on more lines, where items only go through modules that work on them.

4.3.1 AS_0: Branch Between Common Modules

We start by defining the most common type of anti-serialization with the AS_0 transformation rule given in definition 4.3. Informally the rule states that if we have a set of modules $B_{r,s,e}$, which works the recipe $r \in R$ exclusively, lying between the modules s and e on Γ_0 then we may branch out all modules in $B_{r,s,e}$ to form a new line. This new line is connected to Γ_0 at s and e. We will explain this more in depth, by going over the sets brough up in the rule.

---

definition 4.3 Formal definition of the AS_0 transformation rule

$$[\text{AS}_0]\ \frac{0 < |B_{r,s,e}| \wedge s <_k e}{(R, M, \Gamma, \Gamma_0, AW, Start, End) \rightarrow_{AS} (R, M, \Gamma', \Gamma'_0, AW, Start', End')}$$

Where:
$r \in R$
$s, e \in K_{\Gamma_0,r}$
$\Gamma'_0 = (Pre_s \cup \{s\} \cup A_{r,s,e} \cup \{e\} \cup Pos_e, \prec)$
$\Gamma' = \Gamma \cup B_{r,s,e}$
$Start' = Start \cup \{(B_{r,s,e}, s)\}$
$End' = End \cup \{(B_{r,s,e}, e)\}$

We initially define a special set on r, $\bar{r}$, which contains all types of work which are not unique to r:

$$\bar{r} = \bigcup_{r' \in R} r', \quad \texttt{if}\ \ r' \neq r$$

Next we define, what we refer to as common modules. These are modules in Γ_0, which r needs to use along with at least some other recipe. These are important to identify, as they may not be branched out, since that would make them inaccessible to the other recipes than r. Given r and Γ_0 we define the set of common modules $K_{\Gamma_0,r}$ as follows:

$$K_{\Gamma_0,r} = \{m | m \in \Gamma_0 \wedge \exists \rho \in AW(m),\ \{\rho\} \subseteq r \wedge \{\rho\} \subseteq \bar{r} \wedge r \in R\}$$

From this set we can then define $\alpha_{\Gamma_0,r}$, which is the set of modules in Γ_0 which are not used by r:

$$\alpha_{\Gamma_0,r} = \{m | m \in \Gamma_0 \wedge \forall \rho \in AW(m),\ \{\rho\} \nsubseteq r \wedge r \in R\}$$

Along with $\beta_{\Gamma_0,r}$, which is the set of modules in Γ_0 used only by r:

$$\beta_{\Gamma_0,r} = \{m | m \in \Gamma_0 \wedge \forall \rho \in AW(m),\ \{\rho\} \subseteq r \wedge \{\rho\} \nsubseteq \bar{r} \wedge r \in R\}$$

Next we define the binary relation relation $<_K$ as:

$$a <_K b = \begin{cases} tt \ \texttt{if} \ \ a, b, c \in K_{\Gamma_0, r} \wedge a \prec b \wedge \neg(a \prec c \wedge c \prec b) \\ \texttt{else} \ \ ff \end{cases}$$

We also define a total order of all modules in between two modules, s and e, in some line $\gamma \in \Gamma$:

$$M_{s,e} = (\{m | m \in \gamma \wedge \gamma \in \Gamma \wedge s \prec m \wedge m \prec e\}, \prec)$$

We can now define the total order of all modules appearing between s and e, which are not used to work r, $A_{r,s,e}$ as follows:

$$A_{r,s,e} = (\{m | m \in M_{s,e} \wedge m \in \alpha\}, \prec)$$

Similarly we can define the total order $B_{r,s,e}$, which contains all modules that appear between s and e which work exclusively on r

$$B_{r,s,e} = (\{m | m \in M_{s,e} \wedge m \in \beta\}, \prec)$$

We define Pre_s which is the set of all modules that come before a module s in line $\gamma \in \Gamma$:

$$Pre_s = (\{m | m \in \gamma \wedge \gamma \in \Gamma \wedge m \prec s\}, \prec)$$

And lastly we define Pos_e which is the set of all modules that come after a module e in line $\gamma \in \Gamma$:

$$Pos_e = (\{m | m \in \gamma \wedge \gamma \in \Gamma \wedge e \prec m\}, \prec)$$

Having defined these sets the rule in definition 4.3, shows that in the case where $|B_{r,s,e}| > 0 \wedge s <_K e$ then we may update Γ with a new lines defined by the total order $B_{r,s,e}$ in addition to an update of Γ_0, which removes $B_{r,s,e}$ from the line. This describes a branch out from Γ'_0, which exclusively works upon r before rejoining with the old line.

While this rule is formally sound, it can be difficult to read. As such we have tried to illustrate the transformation from configuration to configuration visually in fig. 4.3. This visualization uses a homemade notation. Square boxes refer to individual modules. Boxes with rounded corners are total orders of sets of modules. A graphical line of boxes flowing from left to right is a line in Γ ordered as shown. If a module s and a line γ are connected by a vertical upward going edge, then *Start* contains the ordered pair (γ, s). At the same time, if the edge goes downwards from a line γ to a module e, then *End* contains the ordered pair (γ, e).

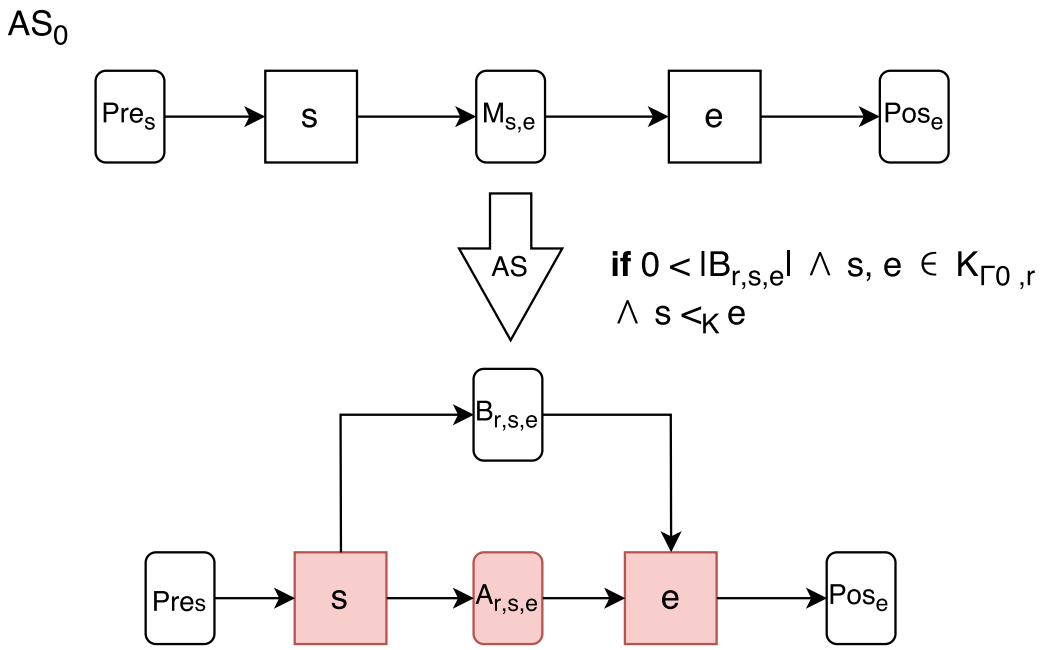

Figure 4.3: A visual representation of the AS_0 transformation rule

For the rest of this chapter, we will use these visualization to explain our transformation rules, instead of setting up transition rules. If any new derived sets are shown in a new transformation rule, we will describe them as well. Note that while we always show new lines appearing above an old line, a new line may be placed at either side in either of the rules described in this chapter.

In the rest of this section we will describe two special cases known as AS_1, branch in, and AS_2, branch out.

4.3.2 AS_1: Branch In

It may be that we have modules that precede the first module of $K_{\Gamma_0,r}$ that we would like to branch out, but can not with the rule described in definition 4.3, as it requires two distinct $K_{\Gamma_0,r}$ modules. In this case, we describe the transformation rule AS_1, which can be seen in fig. 4.4.

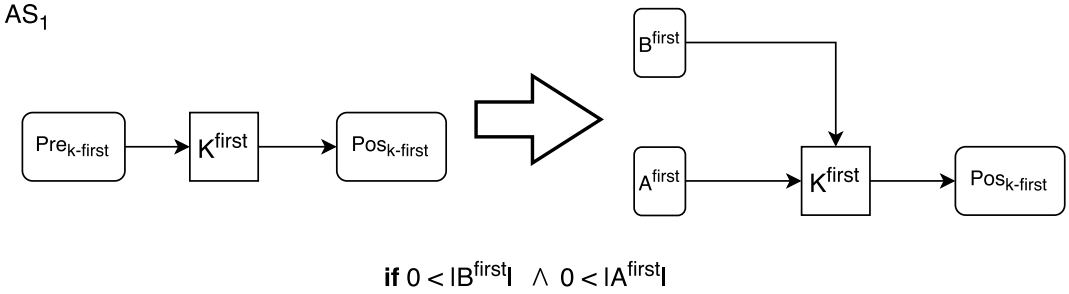

Figure 4.4: A visual representation of the AS_1 transformation rule. This is used for the case where we which to branch out modules prior to the first element of $K_{\Gamma_0,r}$

We define the first module in $K_{\Gamma_0,r}$ as:

$$K^{first}_{\Gamma_0,r} = m \ \texttt{where} \ \forall m' \in K_{\Gamma_0,r} \wedge m \neq m', m \prec m'$$

Similarly for the AS_0 transformation rule, we define two total orderings. $A^{first}_{\Gamma_0,r}$ that describes all modules before $K^{first}_{\Gamma_0,r}$ which do not work on r. And $B^{first}_{\Gamma_0,r}$ which describes all modules before $K^{first}_{\Gamma_0,r}$ which exclusively works on r.

$$A^{first}_{\Gamma_0,r} = (\{m | m \in \alpha_{\gamma,r} \wedge m \in Pre_{K^{first}_{\Gamma_0,r}}\}, \prec)$$

$$B^{first}_{\Gamma_0,r} = (\{m | m \in \beta_{\gamma,r} \wedge m \in Pre_{K^{first}_{\Gamma_0,r}}\}, \prec)$$

We also again use the Pre_s and Pos_e sets here, but used on $K^{first}_{\Gamma_0,r}$ instead of s and e as in the case of the previous rule.

4.3.3 AS_2: Branch Out

Similarly to AS_1 we have the case, where the last element in $K_{\gamma,r}$, is proceeded by a set of modules that we would like to branch out. For this we describe the transformation rule AS_2, which can be seen in fig. 4.5.

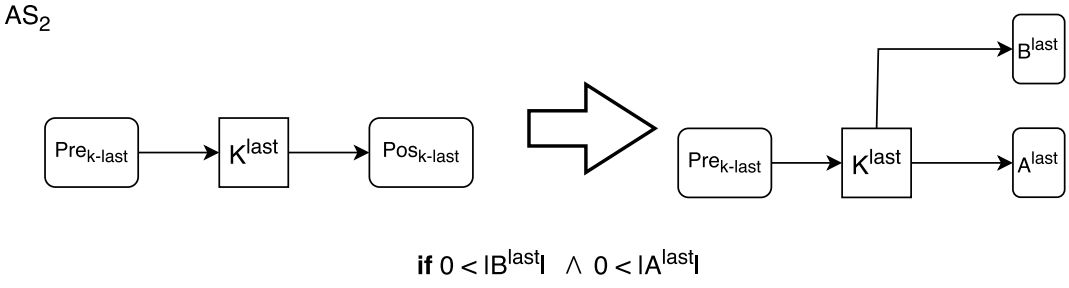

Figure 4.5: A visual representation of the AS_2 transformation rule. This is used for the case where we which to branch out modules proceeding the last element of $K_{\Gamma_0,r}$

We define the last module in $K_{\Gamma_0,r}$ as:

$$K^{last}_{\Gamma_0,r} = m \texttt{ where } \forall m' \in K_{\Gamma_0,r} \wedge m \neq m',\, m' \prec m$$

Again we also define two total orders. $A^{last}_{\Gamma_0,r}$ that describes all modules after $K^{last}_{\Gamma_0,r}$ which do not perform any work on r. And $B^{last}_{\Gamma_0,r}$ which describes all modules after $K^{last}_{\Gamma_0,r}$, which exclusively works on r.

$$A^{last}_{\Gamma_0,r} = \{m | m \in \alpha_{\gamma,r} \wedge m \in Pos_{K^{last}_{\Gamma_0,r}}\}$$

$$B^{last}_{\Gamma_0,r} = \{m | m \in \beta_{\gamma,r} \wedge m \in Pos_{K^{last}_{\Gamma_0,r}}\}$$

4.4 Parallelization

In this section we will describe a set of transformation rules known as Parallelizations. In fig. 4.2 we use only the minimal number of modules necessary to produce the three recipes given in fig. 4.1. It could easily be imagined that we had more modules capable of doing the same work as the ones in the given example. As such, we produce a set of transformation rules that allow for possibly greater throughput by parallelizing free modules with similar modules already in a line.

4.4.1 $Para_0$: Parallelization Between Common Modules

We start by defining the most common type of parallelization with the $Para_0$ transformation rule. Informally the rule states that if we have the total order $M_{s,e}$ of modules in between the modules s and e. Then we can add a parallel line starting at s and ending at e, if we can find an existing total order of free modules $P_{s,e}$ that can perform the same work as the modules in $M_{s,e}$. The visual representation of the $Para_0$ transformation rule can be seen fig. 4.6. Here we expand the visual representation of our rules with the square box with a T in it. This box is just represents a single module that can perform no work, commonly referred to as a transport module.

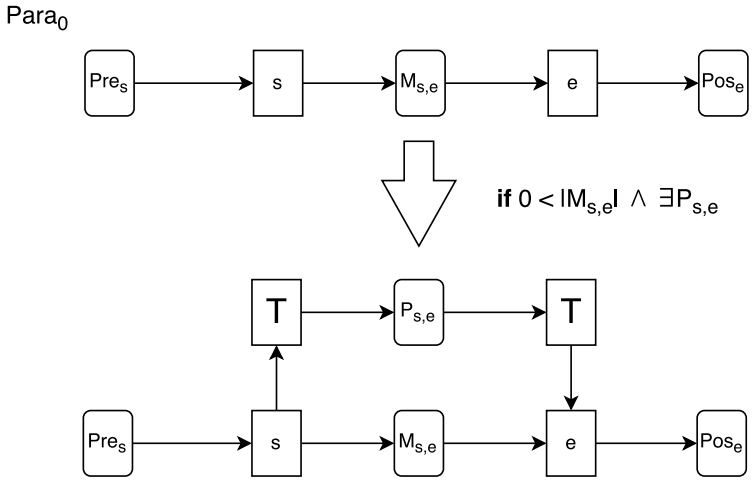

Figure 4.6: A visual representation of the $Para_0$ transformation rule

In order to find the total order, $P_{s,e,}$, that can perform the same work as $M_{s,e}$. We first define the set of all free modules, FM. We say that the set of all free modules is the relative complement of all modules and all modules in a line $\gamma \in \Gamma$.

$$FM = M \setminus \{m | m \in \gamma \wedge \gamma \in \Gamma\}$$

We describe a set of pairs, where free modules are paired with modules in a total order X, if the free modules can at least do the same work as the work currently being done by the modules in X.

$$ParaMap_X = \{(m, m') | m \in FM \wedge m' \in X \wedge AW(m') \subseteq m\}$$

We then describe all sets of pairs that could be a possible parallel line for the modules in X. Note that this set could be the empty set, as the modules needed for creating a possible line might not available in FM.

$$ParaMapPaths_X = \{p \in ParaMap_X^2 | (m, m') \in p \wedge (n, n') \in p \wedge |p| = |X| \wedge \forall m' : m' \neq n'\}$$

After this we define $s[1]$ as the operation that given a set of pairs s, gives the set of all the first elements of the pairs in s.

$$s[1] = \{m_1 | (m_1, m_2) \in s\}$$

Using this we define the total order $P_{s,e}$ as:

$$P_{s,e} = (p[1], \prec), \quad \texttt{where } p \in ParaMapPaths_{M_{s,e}}$$

This definition means that there might not exist any $P_{s,e}$ at all, or that there might be multiple candidates for $P_{s,e}$. Note that we replaced the set X in $ParaMapPaths$ with the total order $M_{s,e}$.

Not shown in fig. 4.6 is that we also update the function AW of our new configuration to AW'. We define AW' as:

$$AW' = AW \cup p, \quad \texttt{where} \quad p \in ParaMapPaths_{M_{s,e}}$$

As with the rules for anti-serialization we also have two special cases for parallelization. Namely $para_1$, branch in, and $para_2$, branch out. The rest of this section will describe these two rules.

4.4.2 $Para_1$: Branch in

Again, similarly to AS_1, it may be that we wish to parallelize everything up to a certain module, without having branched out from some starting module. For this we describe the transformation rule $Para_1$, which can be seen in fig. 4.7.

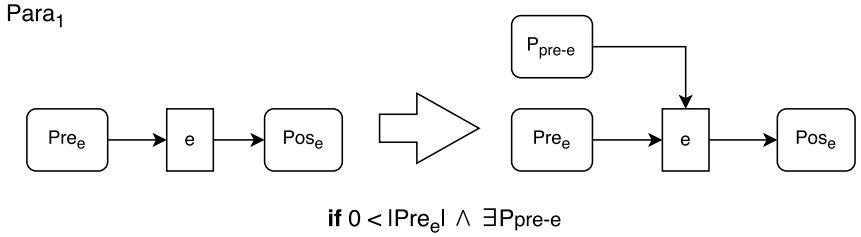

Figure 4.7: A visual representation of the $Para_1$ transformation rule

We here define the total order as P_{Pre_e} as:

$$P_{Pre_e} = (p[1], \prec), \quad \texttt{where} \quad p \in ParaMapPaths_{Pre_e}$$

Not shown in fig. 4.7 is that we also update the function AW of our new configuration to AW'. We define AW' as:

$$AW' = AW \cup p, \quad \texttt{where} \quad p \in ParaMapPaths_{Pre_{s,e}}$$

4.4.3 $Para_2$: Branch out

We then, similarly to AS_2 describe the case in which we would like branch out parallelizing line, but not join it again. For this we describe the transformation rule $Para_2$, which can be seen in fig. 4.8.

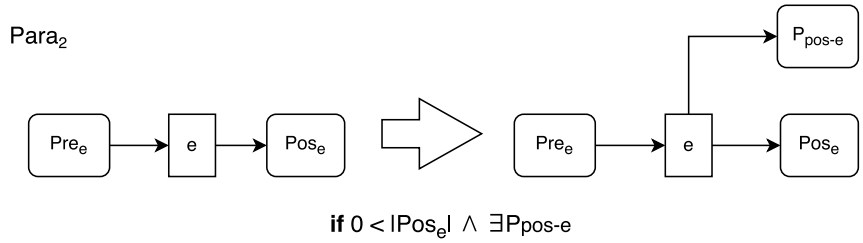

Figure 4.8: A visual representation of the $Para_2$ transformation rule

We here define the total order P_{Pos_e} as:

$$P_{Pos_e} = (p[1], \prec), \quad \texttt{where} \quad p \in ParaMapPaths_{Pos_e}$$

Not shown in fig. 4.8 is that we also update the function AW of our new configuration to AW'. We define AW' as:

$$AW' = AW \cup p, \quad \texttt{where} \quad p \in ParaMapPaths_{Pos_{s,e}}$$

4.5 Swap

In this section we will describe a set of transformation rules know as swaps. There are two distinct cases of swaps. One in which we swap out a module in a line with a free module, which is able to do the same work as the one it is replacing is currently doing on some recipe. Second, one in which we swap out a module within a line, and another module within a line that are capable of the same work, which the other is currently doing. These transformations can be beneficial, if we swap a faster module with a slower module at a choke point, allowing for more throughput.

4.5.1 $Swap_0$: External Swap

For the case of swapping a module out from a configuration with a free module, we have described the transformation rule $Swap_0$, which can be seen in fig. 4.9.

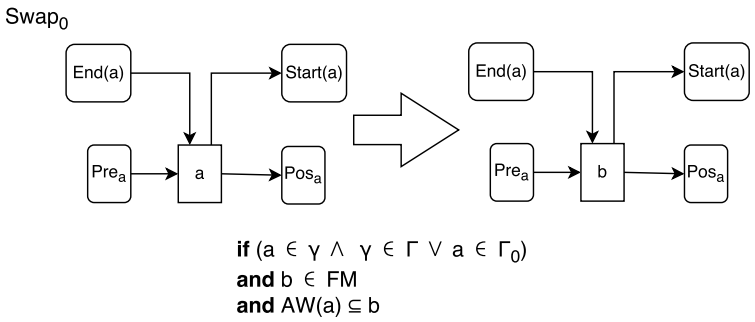

Figure 4.9: A visual representation of the $Swap_0$ transformation rule

Not shown in fig. 4.9 is that we also update the function AW of our new configuration to AW'. We define AW' as:

$$AW' = (AW \cup \{(b, AW(a))\} \setminus \{(a, AW(a))\}$$

4.5.2 $Swap_1$: Internal Swap

For the case of swapping a module from a configuration with another module in the same configuration, we have described the transformation rule $Swap_1$, which can be seen in fig. 4.10.

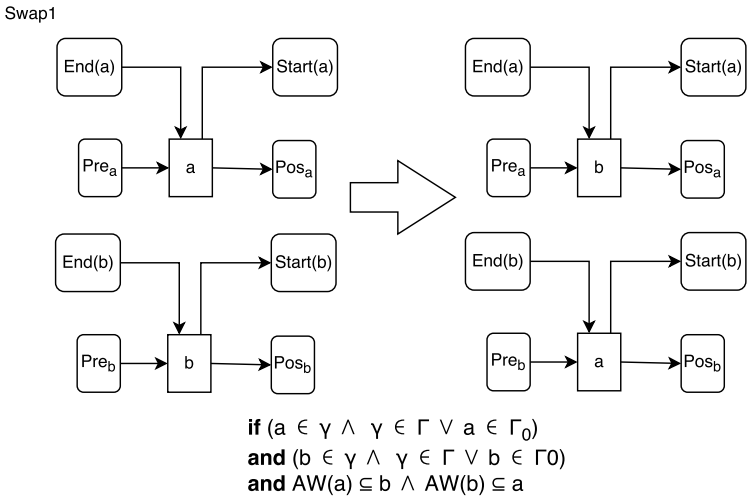

Figure 4.10: A visual representation of the $Swap_1$ transformation rule

Not shown in fig. 4.10 is that we also update the function AW of our new configuration to AW'. We define AW' as:

$$AW' = (AW \cup \{(b, AW(a)), (a, AW(b)\} \setminus \{(a, AW(a)), (b, AW(b))\}$$

4.6 Conflicts

As mentioned back in chapter 3, we did not enforce the physical rules of modules to their fullest in the UPPAAL model. These entail that a module may not connect to another module, if this module is not a neighbour and if the connection would force two modules to take up the same space. The anti-serialization transformations rules can currently create configurations, where lacking module connections mean that not all recipes can be fulfilled. This problem will be solved in section 4.6.1. Furthermore both our anti-serialization and parallelization rules can cause intersections to occur

as a result of adding new lines. We will solve this problem in section 4.6.2 and section 4.6.3.

4.6.1 Restricting Branches Anti-Serialization

For the anti-serialization transformation rules described in section 4.3, we did not ensure that the first and last elements of the new line can be vertically connected to the s and e modules on the main line. Furthermore, until now, we have imagined that no other branch has been made from our line before performing a transformation. Yet, this will not always be the case, and as such can cause some conflicts.

Neighbour Errors

Imagine a situation similar to what the rule AS_0 in fig. 4.3 describes. Here we have specific s and e modules chosen on the line Γ_0, and we have chosen to branch out a specific recipe r. Between the two modules we have $M_{s,e}$. From this we can calculate $A_{r,s,e}$ and $B_{r,s,e}$ as described before. If $0 < |B_{r,s,e}|$, then we may branch out some modules only used to work on r. However in most cases $B_{r,s,e}$ will not have a size such that it can be connected directly to the s and e modules. We therefore modify our AS_0 transformation rule, as to solve this.

If $|A_{r,s,e}| + 2 > |B_{r,s,e}|$, then the result of our AS_0 transformation rule will be the top of fig. 4.11. In this case, we append the new line with transport modules to make the two lines fit each other. Notice that the modules beneath the new line are marked as shadowed, i.e. coloured red. This is used in order to handle transformation conflicts as described below in section 4.6.1. We also introduce blue rounded boxes with indexes to the top right of them. These boxes mean that anything inside of them are serialized i times, where i is the index given with the box.

If $|A_{r,s,e}| + 2 \leq |B_{r,s,e}|$, then the result of our AS_0 transformation rule will be the bottom of fig. 4.11, In this case we append Γ_0 with transporters instead of the new line as to make them fit together.

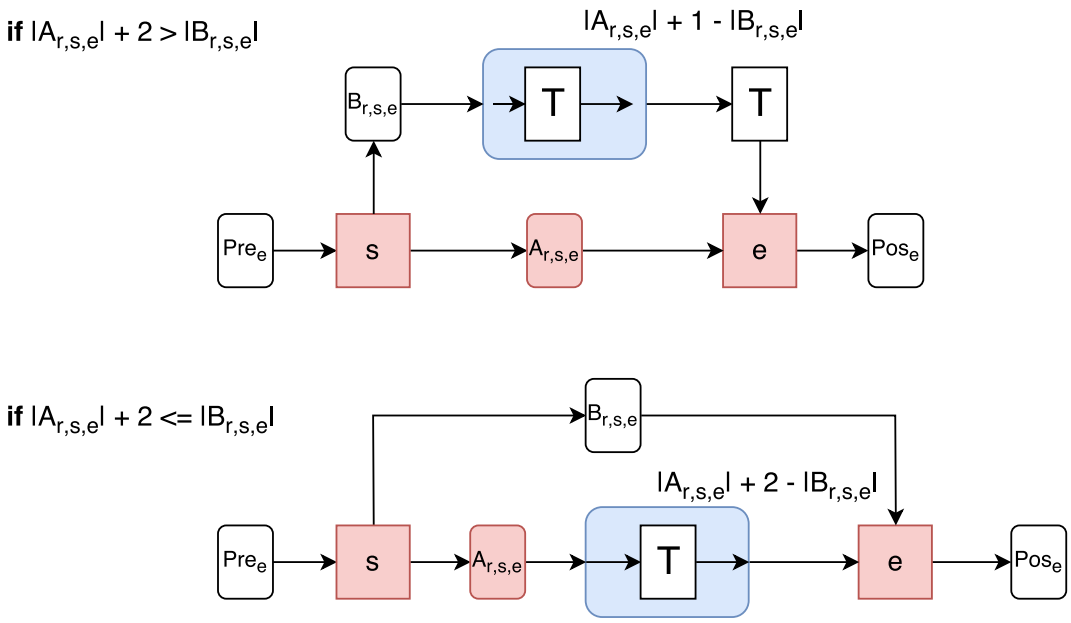

Figure 4.11: The new results of AS_0

Shadowed Modules

On the top part of fig. 4.12, we have already made a branch from $s1$ to $e1$, which results in the modules on Γ_0 being shadowed. Being shadowed means that if we remove the module and just reconnect the old line as usual, then the branch becomes too long to connect back to its old line. Shadowed boxes are visualized with the colour red. The line needs to connect back to its designated point on the old line as the module located here is a common module. This module performs work on the recipe r otherwise worked on by the new line and should therefore not be bypassed. To get around this we, as shown in fig. 4.12, alter our anti-serialization transformation in this case to replace a shadowed module with a transport module, as not to skew the two previous lines away from each other. The example shows this done for a single module, but it may be done regardless of the amount of shadowed modules, which we remove. In fig. 4.12 we also introduce rounded boxes with three dots in them. These boxes just means that here appears any number of modules in a total order.

As an exception, we may not remove a shadowed module, if it also used as a branching in or branching out point for some line. This means that the module is common and is used by the branch. Removing it would keep us from producing items according that branch's specific recipe. Therefore, we never allow for these modules to be removed, even if this means that we can not perform certain anti-serializations.

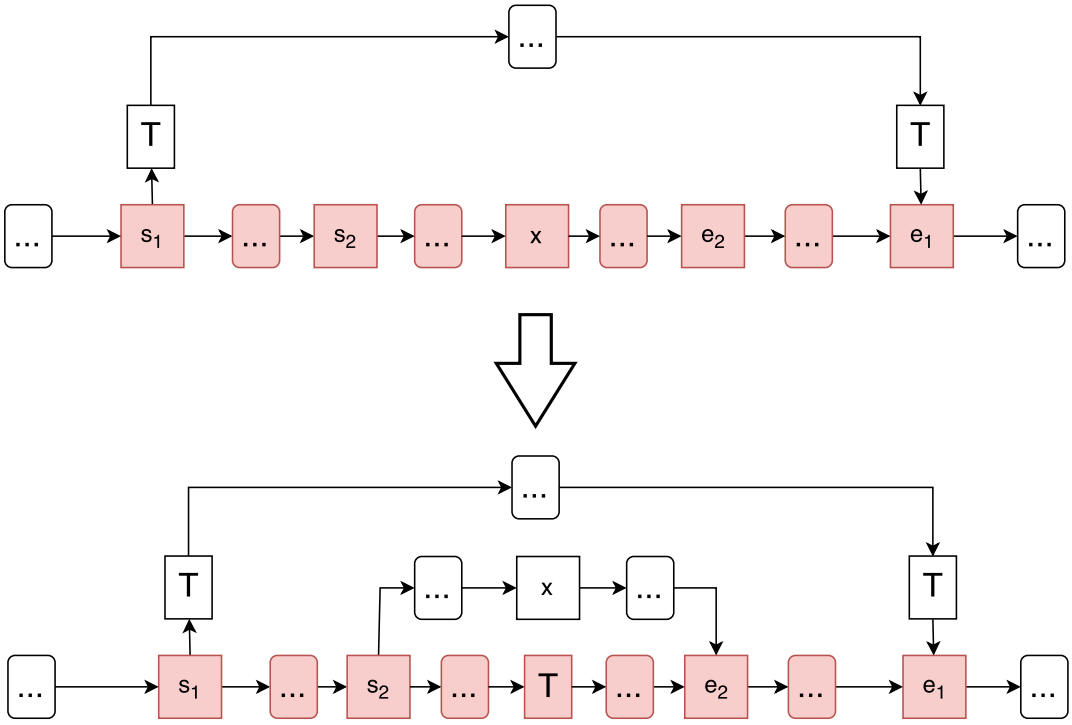

Figure 4.12: Transformation that handles the case where an anti-serialization removes a shadowed module

4.6.2 Push Around

In push around we handle the case, where a new off branching line intersects with an old line, by moving the new line above the old line. In the case where the new line entirely covers the old, we perform the transformation depicted in fig. 4.13. As shown, the new line simply uses transport modules to lift itself up above the old line. These transport modules are technically not a part of the line and only serve the purpose of avoiding the intersection. If two modules are connected vertically according to either the *Start* or *End* functions, we may add transport modules to ensure that the described vertical path can exist.

There is also the case, where the new line is covered by the old line entirely. In this case, we use the transformation in fig. 4.14. Here we do not need to append any transport module, as we move up through the already existing modules, which we would otherwise intersect with.

In the cases where the new entirely intersects the old it will add transport modules where needed, or otherwise guide itself vertically through modules that already exists. These examples only show the case, where a single old line is placed above the main line, from which we branch off. In the case of more line levels, the new line will simply climb vertically until it finds a level where it may be placed without intersection.

Push around is the intersect handling, we use when inserting new lines as a result

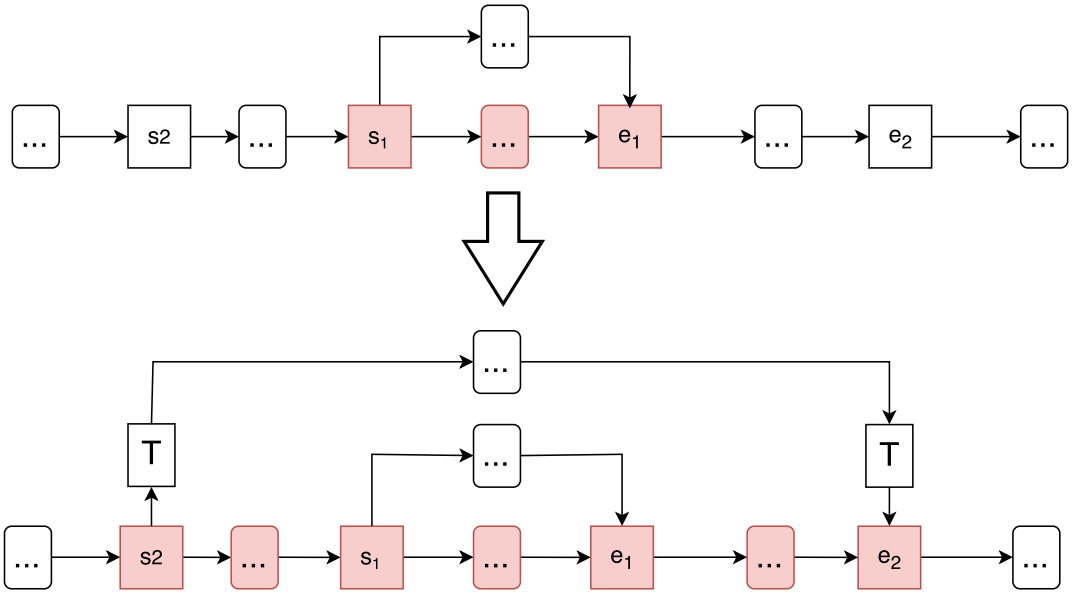

Figure 4.13: How Push Around handles the case, where we insert a new line that covers the entirety of an old line

of anti-serialization. This is chosen as there is no need to keep this new line close to Γ_0, from which it sprouted.

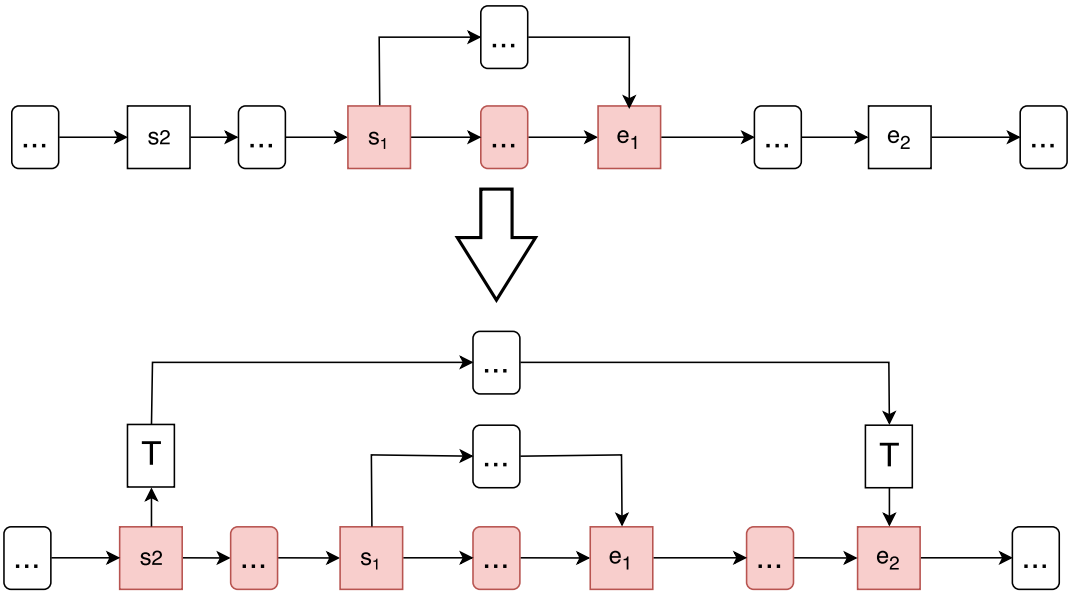

Figure 4.14: How Push Around handles the case, where we insert a new line that is covered by the entirety of an old line

4.6.3 Push Beneath

The other type of intersect handling that we use is called Push Beneath. Here we do the opposite of Push Around and handle intersection conflicts by placing the new line, where we want to place it and then moving vertically any old lines that may intersect. If this move creates another intersection we simply move the lines which we pushed into. This is done until no intersections remain.

In the case where the new line is covered entirely by the old line we avoid intersection as in fig. 4.15. By pushing the new line up one level, we intersect the old, which needs to move up as well. This warrants that the old line gets support from transport modules.

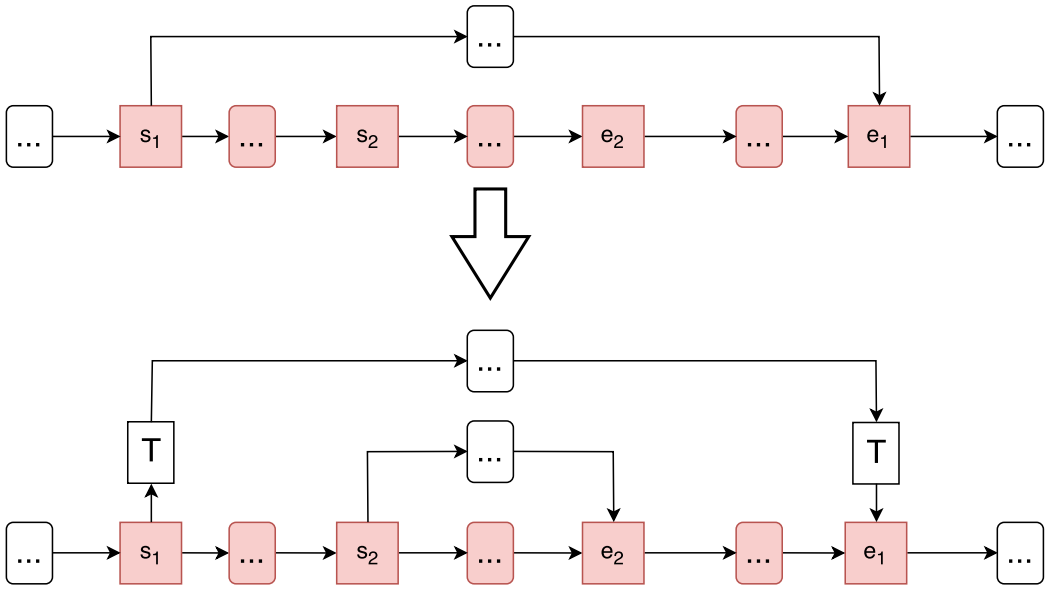

Figure 4.15: How Push Underneath handles the case, where we insert a new line that is covered entirely by an old line

In the case, where the new line covers the old, we handle intersection as in fig. 4.16. As we push up the new line, we need to push up the old. However, we need not use transport modules to reconnect the old line, instead it can be reached by flowing vertically through the new line.

Again, the cases where there is a partial intersection between old and new line is easy to imagine. If the moving up of an old line creates a new intersection, we just move up the line that was already set in place. This is done until no more intersections occur. When this has been done, some old lines may have been disconnected from their main line. However, as these vertical connections are still included in either the *Start* or *End* function, transport modules may be appended, until a path has been recreated.

As can be seen, Push Underneath functions in a manner opposite to Push Around.

We decide to use it, when handling intersections that occur as a result of a parallel transformation. We want our parallel lines to be close to the line, which it sprouted from. Otherwise we may not reap the benefits of adding extra modules. This is not needed as much, when doing anti-serialization, which is why we use Push Around for that instead.

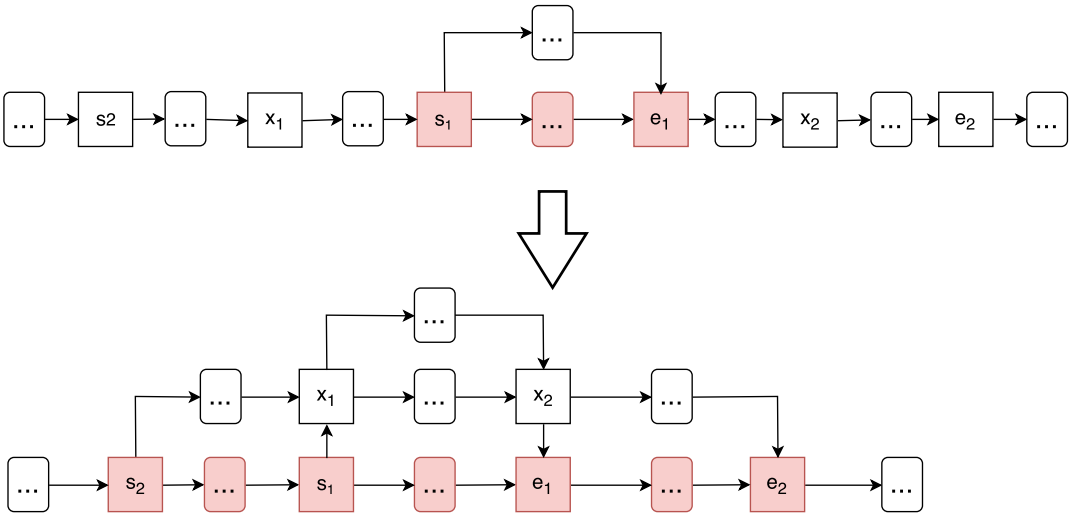

Figure 4.16: How Push Underneath handles the case, where we insert a new line that entirely covers an an old line

Chapter 5

Configuration Optimization

We have now defined both a UPPAAL model, which allows us to rate configurations through simulation, as well as a set of formal transformation rules that allow us to generate new configurations. When given a set of modules and an order, these tools allow us to search through an expanding set of candidate configurations, which we can compare through their ratings. This chapter will explain how we implement this search using python.

5.1 UPPAAL Configuration Generation and Rating

As seen in appendix A, to set up a single configuration in UPPAAL requires a lot of work. UPPAAL saves a system definition along with templates to an XML file. We are able to set up new configurations within UPPAAL by altering this file. Being able to generate this XML file for any configuration that we want to simulate and rate is needed in order to implement our search.

This generation is implemented with the *generate_xml* function, which can be found in the generate_xml.py file along with all its helping functions. We will not go in depth with how it works, however its prototype can be seen in code 5.1. As arguments it takes the path to the XML file *template_file*, a list of *SquareModule* objects *modules*, list of *Recipe* objects *recipes*. In addition, it needs the path, where to write the new XML file *xml_name* and the path, where to write the file containing the query to be run on the configuration *q_name*. This is the query run on the configuration to check if all items in the given order can be completed.

```
generate_xml(template_file, modules, recipes, xml_name, q_name)
```

Code 5.1: Prototype of the *generate_xml function*

Recipe and *SquareModule* are python classes, which we have defined in order to help us set up orders and configurations.

The prototype of *Recipe*'s constructor can be seen in code 5.2. To insantiate, we need the recipe's name *name*, a dictionary describing the recipe's acyclical dependency graph *dependencies*, name of a module to start on *start_module*, direction to

enter the start module from *start_direction* and the amount of items needed to be produced according to this recipe. Thus a list of *Recipe* objects actually describes an order.

```
class Recipe:
    def __init__(self, name, dependencies, start_module, \
                                        start_direction, amount):
```

Code 5.2: Prototype of the *Recipe* class

The prototype of *Module*'s constructor can be seen in code 5.3. To instantiate, we need the module's name *m_id*, a dictionary mapping names of work to the amount of time it takes to perform and a 4 by 4 array describing transport time *t_time*. In addition, we must set the length of the module's queue in *queue_length*, as well as setting the boolean *pass_through* to indicate, whether we can pass through this module, while it is working. To each of the parameters *up*, *down*, *left*, *right* we can assign at most one neighbour module, which this module may pass items to. To make a configuration, we connect different modules through these last four parameters.

```
class SquareModule(Module):
    def __init__(self, m_id, wp_time,  t_time,
                                        queue_length, allow_passthrough,
                                        up=None, down=None,
                                       left=None, right=None):
```

Code 5.3: Prototype of the *SquareModule* class

When we have a list of *Recipe* objects, describing an order, and a list of connected *Module* objects, describing a configuration, we may get the configuration rating by calling the *get_best_time* function found in *uppaalAPI.py*. This can be seen in code 5.4. The function first generates a new XML file and query file from the configuration and its order using *generate_xml*. During this generation, the names of modules, works and items are translated to numerical ids in UPPAAL. The *generate_xml function* returns three dictionaries, which are mappings from the new module, work and item names to the old ones.

It then runs the UPPAAL model checker through the verifyta executable directly on the generated XML and query file. This is done with the *run_verifyta* function. It also takes a few additional arguments to run the verifyta executable. $t2$ means that we want the fastest trace, $o3$ means that we perform an optimal first search, u means that we want a result summary, and y makes sure that we get the trace.

This returns two strings, result and trace. From result we extract the rating using the *trace_time* function. Additionally, we send the trace to the *get_traversal_info* function along with the name mappings from before. This analyses the trace, including all states and transitions, and extracts useful information about production. This includes a map from modules to the items that they work on *worked_on*. The map from module names, to items they transport *transported_through*. We also

get the mapping from modules to their active works set, mentioned in chapter 4, *active_works*. This is used to update the module objects in the configuration according to the work that they have actually done on items. These mappings are returned from the function along with the rating.

```
XML_FILE = 'temp.xml'
Q_FILE = 'temp.q'

def get_best_time(recipes, modules, template_file, verifyta):
    m_map, w_map, r_map =\
        generate_xml(template_file, modules,
                     recipes, xml_name, q_name)

        result, trace = run_verifyta(
                                XML_FILE, Q_FILE,
                                "-t 2", "-o 3",
                                "-u", "-y",
                                verifyta=verifyta)


    time = trace_time(trace)
    trace_iter = iter((trace.decode('utf-8')).splitlines())
    worked_on, transported_through, active_works = \
        get_travsersal_info(trace_iter, m_map, r_map, w_map)

        return time, worked_on, transported_through, active_works
```

Code 5.4: *get_best_time* function

With these objects and functions, we have now introduced the basic constructs used to set up configurations and orders through python, as well as how to extract the ratings from configurations. As to get an idea of how they can be applied, appendix B to generate the rating for the construct and order pair described in section 3.4.1.

5.2 Tabu Search

We have now defined how we can set up configurations and orders in python, as well as how to get the rating of a configuration producing a specific order. We now tackle the problem of finding the optimal configuration, given an order of items and a set of available modules. Finding the optimal configuration is a difficult problem to solve. Therefore we do not focus on creating an optimization algorithm, but rather a heuristic. Unlike optimization algorithms, heuristics do not guarantee a globally optimal solution. Instead it promises a locally optimal solution, which may be good enough in practice.

5.2.1 Choosing a Meta-Heuristic

A meta-heuristic is a problem-independent technique used to develop heuristics for optimization problems. We choose to look into the local search family of meta-heuristics. These describe heuristics, where we try to tackle the problem of optimizing some measure, by moving between candidate solutions to our problem. In our case, the measure we want to optimize is the configuration rating, where we want to find the configuration with the minimal value.

The most basic form of local search is hill climbing. Here we perform a local search by starting from some initial candidate solution. We then generate the neighbouring solutions and compare them on their measure value. The function returning the set of neighbours to a solution x is called $N(x)$. The neighbour with the best measure is chosen as the new frontier. Search continues by generating all neighbours to this solution as to compare measures and find an even better frontier. This continues until we have a frontier, where no neighbours have a better measure. The final frontier is then chosen as the best solution. The problem with this approach is that we are very likely to move into a local optima.

Tabu Search is another type of local search, developed by Fred W. Glover [10]. It focuses on using memory as a manner of moving away from local optima, which otherwise catches us when hill climbing. Memory comes in two different flavors, short term and long term. As we search, any solution that we pick as frontier is added to the short term memory, also known as a tabu list. Let us call the set of solutions in short term S. We define the set of neighbours to a frontier x, from which we may pick a new frontier as $N(x) \setminus S$. This means that we can not pick any neighbour residing in short term memory. This allows us to pick neighbours, which we would otherwise have dismissed as frontiers. In addition, we are allowed to pick an ill fitting neighbour to be frontier, if no other neighbour optimizes our measure. Both of these constructs guide us around local optima.

Long term memory can have a few different definitions. It may be used as an extension of short term memory. Both short and long term have a limited size. This means that they will have to forget old frontiers to make room for new ones. When short term memory forgets a solution, it may end up in long term memory and continue to have us steer clear of that particular solution. In other implementations, long term memory is instead used to reset the search. It may happen that we enter a bad search area with many low ranking local optima. To escape this, we may replace our current frontier with one found in long term memory, as to drive search into a new area.

Memories may also be defined in different ways. A memory may describe an entire solution or perhaps just a subset of attribute values for a solution. In short term memory, the latter will in general exclude more possible frontiers.

When performing a tabu search, there needs to be a balance between diversification and intensivation. Hill Climbing is pure intensivations, where we constantly try to optimize our measure. Random search on the other hand is pure diversification, where there is no active attempt made to optimize our measure. Depending on

implementation, many factors in a tabu search may control diversification and intensivation. As an example, the larger short term memory is, the more diversification. The smaller it is the more intensivation.

5.2.2 Tabu Search Implementation

In this subsection, we will describe how we have implemented a tabu search, which is used to optimize on configuration rating. While it has been implemented in python, we have decide to explain it using psuedo code as to easier communicate, what we have done. The psuedo code can be seen in code 5.5. The actual implementation can be found in *tabu_search.py*.

The search is given a configuration as a list of connected *Module* objects and an order as a list of *Recipe* objects. In addition it takes a special transport module *transport*, which it used to append configurations and make them physically possible as described in section 4.6. Lastly, it takes the amount of iterations to perform *iter* and the maximum size of short term memory *max_short_size*. The search starts by setting up its memory. Short term memory has a fixed size, while long term does not. Dynamic memory is a mapping from configurations to their rating. It keeps record, so that we do not have to recalculate the rating of the same configuration several times.

Next a call to *get_init_configs* generates all initial single line configurations, which can produce the stated order using a minimum amount of modules. This is done by composing the functional dependency graphs of our recipes and then for every possible topological sort of the resulting graph, we place down a line of modules accordingly.

The entire set of initial configurations are evaluated using the *evaluate* function. This function uses code 5.4 to produce the rating as described in section 5.1. It also updates *dynamic_memory* to record the rating for the given configuration. Once evaluated, the configurations are added as memory to long term memory. We choose a memory to be defined as an entire configuration. In the actual implementation, configurations are stored as describtive strings. If we ever need to go back to a configuration, we can use one of these memory strings, to set up the configuration again as a list of connected *Module* objects.

Once all initial configurations have been evaluated, we pick the one with the best rating as our frontier. From this point on the actual search occurs for *iter* iterations in the following loop. At each iteration we call *get_neighbour_func*. This function returns one of the three functions, which may generate frontier neighbours: *neighbours_anti_serialized*, *neighbours_parallelize* or *neighbours_swap*.

Using the selected function *func*, we generate a set of neighbours to our frontier. These are all evaluated and the frontier is chosen as the best configuration in the set of neighbours, which are not included in short term memory. If no frontier is found, either no neighbors were generated or short term memory excludes them all, we call *backtack* to find an old configuration in long term memory to use as frontier. Thus, we use long term as a way to reset our search, when in a bad situation. Having

```
def tabu_search(modules, recipes, transport, iter, max_short_size):
        short_mem[max\_short\_size]
        long_mem = []
        dynamic_mem = {}

        # Get initial liniear lines and evalutate them
        init_configs = get_init_configs(modules, recipes)
        for config in init_configs:
                evaluate(config, dynamic_mem)
                long_mem.append(config)

        frontier = best(long_mem)

        # The actual Tabu Search
        from 0 to iter:
                # Select how to generate neighbours
                func = get_neighbour_func()

                # Get neighbours and evaluate
                neighbours = func(modules, recipe, transport)
                for c in neighbours:
                        evaluate(c)

                # Select the best neighbour not in short_mem
                frontier = best(neighbours - short_mem)

                # If all neighbours are in short_mem
                if frontier == None:
                        # We go to some config in long_mem
                        frontier = backtrack(long_mem)
                        long_mem.remove(frontier)

                # Update memory
                if len(short_mem) > max_short_size:
                        short_mem.pop()
                short_mem.append(c)
                long_mem.append(c)

        return best(dynamic)
```

Code 5.5: pseudocode showing a simplified version of the tabu search implementation

now found a new frontier, it is added to short and long term memory. If short term memory is overflowing, we remove its oldest element to make room.

Once we exit the loop, we return the best configuration found during the search.

5.3 Neighbour Functions

In this section we describe, how we implement the transformation rules described in section 4.2 as neighbour functions. These neighbour functions are used by our tabu search for generating frontier neighbours as shown in code 5.5. The code for these functions can be found in the *neighbour_functions* directory and the code for solving conflicts in *path_placers.py*.

5.3.1 Anti-Serialization

For anti-serialization, we decided to implement the neighbour function in such a way that it retrieves all possible anti-serializations of a randomly chosen recipe on the main line. Meaning that whenever the transformation rules, AS_0, AS_1 and AS_2 could be applied on the main line for a chosen recipe, we did. We then return the resulting configurations as a set of neighbours. We also take care of shadowing and adding transport modules as brought up in section 4.6.1.

We could have implemented the neighbour function for anti-serialization in such a way that it returned all possible uses of the anti-serialize transformation rules on all recipes. We however decided against this, as it could result in too many neighbours that our tabu search would have had to evaluate in one iteration.

5.3.2 Parallelization

For parallelization, we designed the neighbour function such that we get all possible largest parallelizations. This is done because. as we assume that if we can parallelize a set of modules on a line, then the neighbours that are a configuration with a subset of these modules parallelized is likely to be worse. This might not always be the case, but it is an assumption we make, so that again our tabu search will not have to perform too many evaluation in a single iteration.

WAdditionaly, we only implemented the $Para_0$ rule, and not $Para_1$ and $Para_2$. $Para_1$ was not implemented due to how, as we hcandled items start location. As in our implementation a only give one start module to *Recipe* objects. An item can only start on a single specefic module, and $Para_1$ requires that an item can start onhas the option to start in one of multiple location. $Para_2$ was not implemented due to time constraints and for the gullible notion of symmetry.

5.3.3 Swap

For swap, we implemented the neighbour function, such that we get all possible swaps. This can possible lead to a lot of neighbours having to be evaluated, if

a configuration has a lot of free modules that are capable of being swapped. We can combat this, by not making the swap transformations as likely as the other transformations in our tabu search.

5.3.4 Conflicts

For our conflict solving transformation rules described in section 4.6, we implemented special functions that are capable of solving line conflicts, which a neighbour might have after being generated by either *Anti-Serialization* or *Parallelization*. A *push_around* function implements the *PushAround* rule for fixing conflicts that occur due to anti-serialization. A *push_beneathe* function implements the *PushBeneath* rule for conflicts that occur due to parallelization. We always check the neighbours returned by our neighbour functions for conflicts and resolve them before we evaluate them. We are not interested in evalutating and comparing neighbours, which we can not physcially set up.

5.4 Generating the Running Example

In this section we will use our Tabu Search to generate a configuration that shares similarities to our running example, which can be seen in fig. 2.2 and found in section 2.3. We will use the same modules that are included in the running example to optimize an initial linear configuration. We will optimize the configuration such that it is able to produce items of the recipes described in fig. 4.1 as fast as possible. Then we will compare the configuration generated through our tabu search to the running example, which we made by hand.

5.4.1 The Configuration Found by the Tabu Search

As mentioned, our total set of modules for the tabu search will be the modules seen in fig. 2.2. Furthermore, we wish to optimize for an order asking to produce 3 items for each recipe described in fig. 4.1. This is done in the hopes that 3 items of each recipe is enough to satiate the configuration enough, such that it encourages parallelization transformations. The resulting configuration of the tabu search can be seen in fig. 5.1.

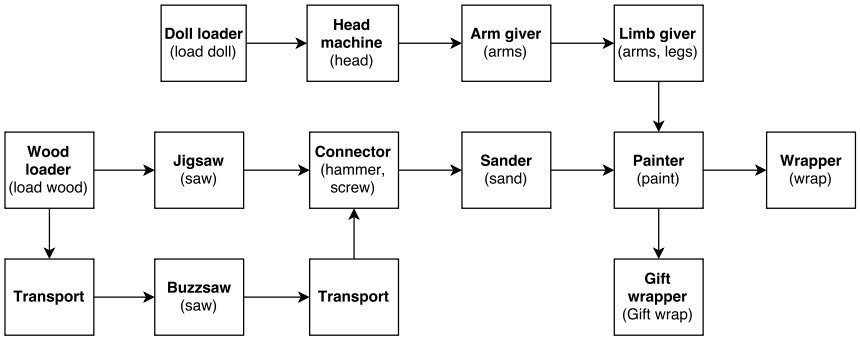

Figure 5.1: The configuration generated by our tabu search.

This configuration generated by our tabu search shows promise, as it has both utilized our anti-serialization and parallelization rules and is somewhat similar to the running example.

We can see that it has used the anti-serialization rules because of the modules that uniquely only produce work for the doll recipe, are disconnected from the rest of the configuration. With the branches joining and splitting from the configuration at the common module `Painter`. In general, our tabu search has anti-serialized much in the same way as we did, when we made fig. 2.2 by hand. The only exception is that we connected the modules `Sander` and `Wrapper` through a transport module, this being done so that rocking horse items may avoid going through `Painter`. We however have not made any type of transformation rule that would be allow the tabu search to do this.

We can see that the search has used parallelization rules, as we have two paths from `Wood Loader` to `Connector` that both can do the work *saw*. This is different from the running example, as we there had the modules `Jigsaw` and `Buzzsaw` placed next to each other instead of parallel. In our UPPAAL model, an item can not pass through a module that can perform work on it. Thus we have not set up a transformation rule, which would place *Jigsaw* and *Buzzsaw* next to each other. If run on our model, all the work *saw* would simply be done on the first module that the item enters, and the second would function merely as a glorified transport module.

A genuine advantage that the running example has over the generated configuration is that it parallelized the modules `Arm Giver` and `Leg Giver` from `Limb Giver`. This was not possible for our tabu search, as our implemented parallelization rules describe that when constructing a parallel line, each module in the new line must perform the active work of one specefic module in the old line. There is no mapping of one module to several in our parallization, as we otherwise see in the running example. This could be a beneficial rule to have implemented for our search.

Overall we are satisfied by the outcome that is our tabu search, as it seems capable, at least for this example, to heuristically find optimized solutions.

Chapter 6

Discussion

Having created our final implementation of our model and tabu search, we will discuss both in this chapter. These will be general thoughts on what we have produced, including what we could have done differently.

6.1 Modeling

To rate configurations, which produce specific item orders, we developed a model in UPPAAL. With this model we can simulate items being produced by a factory. We may also generate the fastest timed trace, and from it extract the time it takes to produce the order. In this section, we will discuss what features we could have added to the model in addition to how it may be optimized further. We bookend the section by bringing up how the model compared to a real life factory.

6.1.1 Left Out Features

In our current model, we may have a case, where the front element in a module's queue is waiting to be worked on by that module. At the same time, all items behind it may just need to pass through the module. On the actual CP Learning Factory setup, which we described in section 3.4.1, they get around this by adding extra queues for items that need to be worked on by the module. In the case of the *Robot Arm* module, it will guide items that it must work on onto an inner conveyor belt, where they will wait for processing. This ultimately allows for other items to more easily pass through the module. Thus we have a real life case, which our model is not designed to handle. The issue with implementing the feature is that it will increase the global state space during search, as fewer items are stuck in queues and are free to transition around the configuration.

We were not able to exactly generate the running example from fig. 2.2 through our tabu search. This is in part because we have not described a transformation rule, which allows us to parallelize modules serially as is the case with *Jigsaw* and *Buzzsaw* in the aforementioned example. We did not do this, as our model does not reap the

benefits of such a configuration. In order to keep down the state space, our model requires that an item may not pass through a module, which may work on it. This means that any wooden sword we create will be sawed by *Jigsaw* and then pass right through *Buzzsaw*. The issue with removing this condition is that our search may unnecessarily branch further out into our configuration in search of another module, which may perform the same work.

Our model was not designed with our transformation rules in mind. We focused on keeping it as general as possible from the beginning. Yet we could get around the problem of parallelization increasing our state space, if we designed the model according to our parallelization rules. If we know that a serial parallelization, as with *Jigsaw* and *Buzzsaw*, can happen, then we know that we should only look ahead by one neighbour, to see if another module may perform the work. Thus we can limit the amount of lookahead branching performed during search.

6.1.2 Optimization

On the issue of optimization in general, we know that our state space begins to grow fast as more branches are added to the configuration. This results in our tabu search becoming slower and slower, as we try to rate configurations of a rising complexity. As the search requires many configurations to be simulated, we want to keep the time spent rating each configuration to a minimum. We realize that there is a balance to find between the number of features, we add to the model, and the state space. We also risk a slower execution time, if the solutions that lower the state space increase the size of our states.

To optimize the model, we could choose to make it less general and implement optimizing features, based on how our transformation rules create new configurations. It may also be beneficial to partition a configuration into several sub-configurations. Rating each smaller part would not take as long as running the whole configuration, as the state space increases exponentially as our configuration becomes more complex. The issue here is that it becomes more difficult to model the real world, as we must concern ourselves with how to handle modules in sub-configurations passsing items onto modules in other sub-configurations.

6.1.3 Comparison to a Real Factory

In section 3.4.1 we looked into how modeled configurations fared compared to a real factory. For simple orders our model fared well, but we see that when orders become more complex, then the total execution time of orders begins drifting apart. In our case, the drift may be because we have not modeled how items may block each other on the inside the modules. We observed that this happened when items had to leave the *Robot Arm* module's work queue. We must also remember that we only compare the total production time of orders, not how the actual factory and modeled configuration fared against each other during production.

The factory, which we compared against, was very simple. If the factory had some of the features mentioned in section 6.1.1 our results would have been different. In addition, it would be interesting to compare against a real factory, which had a branching production similar to our generated. In spite of this, we are rather pleased with our results, as they indicate that our model can be used for some real life configurations.

6.2 Tabu Search

To pick a good configuration for producing an order of items we used the meta-heuristic known as tabu search. In this section we will discuss our generation of neighbours and use of memory in the implementation.

6.2.1 Generating Neighbours

In our implementation of the tabu search, we based our neighbour generation functions on the form transformation rules that we sat up earlier. However these functions are very different in how many neighbours we generate. Swap especially tends to generate a lot of different neighbours. Instead of generating all possible neighbours, it would instead be interesting to stochastically generate neighbourhoods. By just generating a subset of neighbours which are picked stochastically, we could equalize the amount of neighbours that we evaluate at each iteration. Thus more of the total search time may be spend on interesting backtracking using long term memory, which may be more beneficial to the search.

Each of the three neighbourhood functions has a certain probability of being picked. We looked into having these probabilities altered as more search iterations are performed. Thus changing the search rules. Our basic implementation made it very likely to do anti-serializations first, but as time goes on, parallelization and swap become more likely. The idea being that anti-serialization sets up some basic configuration structure, which the others can then alter in less dramatic ways. It would be interesting to look more into, how we may change up our search rules, and when this should happen.

6.2.2 Using Memory

We discover that short term memory is not used that often during search. This is because our transformation rules has a very low chance of later generating the same configurationn. This may only occasionally happen when we generate neighbours using swap. Short term memory is sometimes used, when we backtrack using long term memory. Yet only if we backtrack to a recent element in long term memory, where it's earlier best neighbour is still in short term memory.

Long term could also be used in a more interesting way. Right now it has unlimited size and every frontier picked is stored within it. We also only use it to backtrack in the case that we find no better frontier. A possibility of backtracking using long

term, which increases at each iteration, could allow us to backtrack more and search new areas.

It could be interesting to define a memory not as a configuration, but as a set of attribute values. Thus we would be able to catch more unwanted neighbours using the short term memory. The issue here being how to meaningfully describe a configuration only by a few attributes, which can be used to guide search.

Chapter 7

Conclusion

In this report we started out by looking at Industry 4.0 and modular factory systems. This lead to the following initial problem:

How may it be possible to optimize the throughput of individual production lines in a modular factory system, while keeping down the overall cost of production?

With the problem stated, we inspected our local modular factory set-up. This allowed us gain some insight into how the technology was used. We also looked into relevant problems and discovered that it was difficult for manufacturers to arrive at an optimal configuration, when they need to produce a certain order. We also sat up a running example of a factory configuration, which we kept returning to through the report.

In response to this we sat up the following problem definition:

How may we, given some order of items and set of available modules, be able to generate a factory configuration which has the fastest schedule of any other candidate configuration?

We attempted to solve this problem by first modeling modular factory systems in UPPAAL. Through this, we were able to simulate the production of orders on different configurations. In addition, we were able to, for a given configuration and order, extract the time taken to produce according to the fastest schedule. This value is known as the configuration's rating. Comparing a configuration set up with our model to an actual simple factory, we found the production times to be about equal for the orders tested.

Later we set up some formal rules, allowing us to transform naive configurations, made up of a single line, into configurations that show off parallelism and branching. In addition the rule allow us to only create configurations, which can physically be set up.

The rules were implemented in python and used to generate different configurations, which were searched through using a tabu search. During the search we

tried to optimize on the configuration rating. By using this search we were able to generate a configuration that was very similar to our running example, which we created by our own hands. This shows that all our work comes together to form a tool, which can be used to generate at least some good factory configurations for producing orders.

Chapter 8

Future Work

In this chapter we will briefly go over some of the additions to our project that we would have like to accomplish in the future, if the time and the means are avaliable.

8.1 Missing Implementations of Transformation Rules

Perhaps the most pressing addition we could make, would be to implement the neighbour functions for $Para_1$ and $Para_2$. $Para_2$ should be relatively straight forward to implement, as there is no limitation stopping us from implementing it. $Para_1$ however would require us to redefine how place items onto a configuration. Currently items can only be added to a configuration at single starting module defined by their recipe. Implementing $Para_1$ requires that we find a way that allows items, which are made according to the same recipe, to have different starting modules. This could be solved by defining recipes, not with a starting module, but rather a start work. Then all modules that can perform this start work can have items ,made to that recipe, started on them.

8.2 More Complex Transformation Rules

We would have liked to describe more complex version of most of our transformation rules. Such as rules describing how we could anti-serialize a group of recipes at once. This would be useful, if we have a group of modules on which items made according to similar recipes are exclusivly worked.

Another more complex transformation rule could be allowing the parallel rules to create parallel branches, where we do not need a one to one correspondence between old modules and parallel modules. A good example was previously explained in section 5.4, as the running example could parallelize out two modules that together could do all the work of a single module.

Lastly, we would like to implement a many to many swap transformation. That, similar to the complex parallelization rule explained above, allows us to swap modules

that combined can do the work of whatever we wish to swap it with. Thus we could for example swap out a single module with a group of modules.

8.3 Better Heuristics for Tabu Search

In the current implementation of our tabu search, we use memory rather ad hod, we would like to improve upon how we define and use both short and long term memory. Such that we in general would faster get better search result. One way we could possibly improve our search is that we for every configuration in long term memory also store the configuration chosen as its most optimal neighbour. This could be used to make sure that when we backtrack to a configuration from long term that we do not pick the same frontier.

Bibliography

[1] R. Davies, "Industry 4.0 - digitalisation for productivity and growth," September 2015. Breifing from EPRS.

[2] FESTO Didactic, "FESTO homepage." http://festo.com. Accessed: 2016-09-22.

[3] FESTO Didactic, "Industry 4.0 - Qualification for the factory of the future," 2015. brochure.

[4] Siemens, "Technomatix," 2014. brochure.

[5] A.-R. Sadeghi, C. Wachsmann, and M. Waidner, "Security and privacy challenges in industrial internet of things," in *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, (New York, NY, USA), pp. 54:1–54:6, ACM, 2015.

[6] S. Yin and O. Kaynak, "Big data for modern industry: Challenges and trends [point of view]," *Proceedings of the IEEE*, vol. 103, pp. 143–146, Feb 2015.

[7] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134–152, 1997.

[8] G. Behrmann, E. Brinksma, M. Hendriks, and A. Mader, "Production scheduling by reachability analysis: a case study," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, (Los Alamitos, CA, USA), pp. 140–140, IEEE Computer Society Press, 2005.

[9] G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. Vaandrager, M. Voorhoeve, S. de Smet, and L. Somers, *Formal Modeling and Scheduling of Datapaths of Digital Document Printers*, pp. 170–187. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[10] F. Glover and R. Marti, "Tabu search," 2006.

Appendix A

UPPAAL Configuration

```
// Module 0
const bool work_array0[NUMBER_OF_WORKTYPES] = \
{false,false,false,false,false,true,false,false};
const int ptime_array0[NUMBER_OF_WORKTYPES] = \
 {0,0,0,0,0,60,0,0};
const mid_t next_array0[NUMBER_OF_OUTPUTS] = {-1,1,-1,-1};
const int \
 ttime_array0[NUMBER_OF_OUTPUTS][NUMBER_OF_OUTPUTS] = \
{{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,117,0,0}};
mqueue0 = ModuleQueue(0, 0, 3, work_array0, false);
mworker0 = \
ModuleWorker(0, 1, work_array0, ptime_array0);
mtransporter0 = \
ModuleTransporter(0, 2, ttime_array0, next_array0, false);

// Module 1
const bool work_array1[NUMBER_OF_WORKTYPES] = \
{false,false,false,false,true,false,true,false};
const int ptime_array1[NUMBER_OF_WORKTYPES] = \
{0,0,0,0,53,0,106,0};
const mid_t next_array1[NUMBER_OF_OUTPUTS] = {-1,2,-1,-1};
const int \
ttime_array1[NUMBER_OF_OUTPUTS][NUMBER_OF_OUTPUTS] = \
{{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,107,0,0}};
mqueue1 = ModuleQueue(1, 3, 3, work_array1, false);
mworker1 = ModuleWorker(1, 4, work_array1, ptime_array1);
mtransporter1 = \
ModuleTransporter(1, 5, ttime_array1, next_array1, false);

// Module 2
```

```
const bool work_array2[NUMBER_OF_WORKTYPES] = \
{true,false,false,true,false,false,false,true};
const int ptime_array2[NUMBER_OF_WORKTYPES] = \
{582,0,0,752,0,0,0,850};
const mid_t next_array2[NUMBER_OF_OUTPUTS] = {-1,3,-1,-1};
const int \
 ttime_array2[NUMBER_OF_OUTPUTS][NUMBER_OF_OUTPUTS] =
\ {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,164,0,0}};
mqueue2 = ModuleQueue(2, 6, 3, work_array2, true);
mworker2 = ModuleWorker(2, 7, work_array2, ptime_array2);
mtransporter2 = \
ModuleTransporter(2, 8, ttime_array2, next_array2, true);

// Module 3
const bool work_array3[NUMBER_OF_WORKTYPES] = \
 {false,true,false,false,false,false,false,false};
const int ptime_array3[NUMBER_OF_WORKTYPES] = \
 {0,20,0,0,0,0,0,0};
const mid_t next_array3[NUMBER_OF_OUTPUTS] = {-1,4,-1,-1};
const int \
 ttime_array3[NUMBER_OF_OUTPUTS][NUMBER_OF_OUTPUTS] = \
{{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,112,0,0}};
mqueue3 = ModuleQueue(3, 9, 3, work_array3, false);
mworker3 = ModuleWorker(3, 10, work_array3, ptime_array3);
mtransporter3 = \
ModuleTransporter(3, 11, ttime_array3, next_array3, false);

// Module 4
const bool work_array4[NUMBER_OF_WORKTYPES] = \
{false,false,false,false,false,false,false,false};
const int ptime_array4[NUMBER_OF_WORKTYPES] = \
{0,0,0,0,0,0,0,0};
const mid_t next_array4[NUMBER_OF_OUTPUTS] = {-1,5,-1,-1};
const int \
 ttime_array4[NUMBER_OF_OUTPUTS][NUMBER_OF_OUTPUTS] =\
{{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,112,0,0}};
mqueue4 = ModuleQueue(4, 12, 3, work_array4, false);
mworker4 = ModuleWorker(4, 13, work_array4, ptime_array4);
mtransporter4 = \
ModuleTransporter(4, 14, ttime_array4, next_array4, false);

// Module 5
const bool work_array5[NUMBER_OF_WORKTYPES] = \
```

```
{false,false,true,false,false,false,false,false};
const int ptime_array5[NUMBER_OF_WORKTYPES] = \
{0,0,68,0,0,0,0,0};
const mid_t next_array5[NUMBER_OF_OUTPUTS] = {-1,-1,-1,-1};
const int \
 ttime_array5[NUMBER_OF_OUTPUTS][NUMBER_OF_OUTPUTS] =\
{{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
mqueue5 = ModuleQueue(5, 15, 3, work_array5, false);
mworker5 = ModuleWorker(5, 16, work_array5, ptime_array5);
mtransporter5 = \
ModuleTransporter(5, 17, ttime_array5, next_array5, false);

// Recipe NoFuse
const node rNoFusenode0 = \
 {0, 1, {1, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rNoFusenode1 = \
 {1, 1, {4, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rNoFusenode2 = \
 {5, 0, {3, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rNoFusenode3 = \
 {6, 1, {0, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rNoFusenode4 = \
 {2, 1, {-1, -1, -1, -1, -1, -1, -1, -1}, 0};
const node rNoFusenode5 = \
 { -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
const node rNoFusenode6 = \
 { -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
const node rNoFusenode7 = \
 { -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
node func_depNoFuse[NUMBER_OF_WORKTYPES] = \
{rNoFusenode0,rNoFusenode1,rNoFusenode2,rNoFusenode3,
rNoFusenode4,rNoFusenode5,rNoFusenode6,rNoFusenode7};

const int number_of_nodesNoFuse = 5;
recipe0 =\
 Recipe(0, 0, func_depNoFuse, number_of_nodesNoFuse, 3);

// Recipe LeftFuse
const node rLeftFusenode0 =\
 {3, 1, {2, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rLeftFusenode1 =\
 {4, 1, {0, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rLeftFusenode2 =\
```

```
 {1, 1, {4, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rLeftFusenode3 =\
 {5, 0, {1, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rLeftFusenode4 =\
 {2, 1, {-1, -1, -1, -1, -1, -1, -1, -1}, 0};
const node rLeftFusenode5 =\
{ -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
const node rLeftFusenode6 =\
 { -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
const node rLeftFusenode7 =\
 { -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
node func_depLeftFuse[NUMBER_OF_WORKTYPES] = \
{rLeftFusenode0,rLeftFusenode1,rLeftFusenode2,rLeftFusenode3,
rLeftFusenode4,rLeftFusenode5,rLeftFusenode6,rLeftFusenode7};
const int number_of_nodesLeftFuse = 5;
recipe1 =\
 Recipe(1, 0, func_depLeftFuse, number_of_nodesLeftFuse, 3);

// Recipe BothFuse
const node rBothFusenode0 =\
 {1, 1, {1, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rBothFusenode1 =\
 {2, 1, {-1, -1, -1, -1, -1, -1, -1, -1}, 0};
const node rBothFusenode2 =\
 {5, 0, {3, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rBothFusenode3 =\
 {6, 1, {4, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rBothFusenode4 =\
 {7, 1, {0, -1, -1, -1, -1, -1, -1, -1}, 1};
const node rBothFusenode5 =\
 { -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
const node rBothFusenode6 =\
 { -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
const node rBothFusenode7 =\
 { -1, -1, {-1,-1,-1,-1,-1,-1,-1,-1}, -1};
node func_depBothFuse[NUMBER_OF_WORKTYPES] = \
{rBothFusenode0,rBothFusenode1,rBothFusenode2,rBothFusenode3,
\rBothFusenode4,rBothFusenode5,rBothFusenode6,rBothFusenode7};
const int number_of_nodesBothFuse = 5;
recipe2 =\
 Recipe(2, 0, func_depBothFuse, number_of_nodesBothFuse, 3);

rid_t rqa[NUMBER_OF_RECIPES] = {0,1,2};
```

```
rqueue = RecipeQueue(rqa, 18);rem = Remover(19);
initer = Initializer();
urge = Urgent();
system mqueue0< mworker0< mtransporter0< mqueue1< mworker1< \
 mtransporter1< mqueue2< mworker2< mtransporter2< mqueue3< \
 mworker3< mtransporter3< mqueue4< mworker4< mtransporter4< \
 mqueue5< mworker5< mtransporter5< rqueue< rem< initer< \
 urge< recipe0< recipe1< recipe2;
```

Code A.1: UPPPAL system declaration when running the AllTypes order on the configuration described in section 3.4.1

Appendix B

UPPAAL Configuration

```
from module import SquareModule
from recipe import Recipe
from configuration.tabu_search import tabu_search
from UPPAAL.uppaalAPI import get_best_time
VERIFYTA = '../UPPAAL/verifyta'
XML_TEMPLATE = "../../Modeler/iter3.4.2.xml"

QUEUE_LENGTH = 3

t0 = [[0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 117, 0, 0]]
m0 = SquareModule(
'Case Loader', {'load case': 60},
 t0, QUEUE_LENGTH)

t1 = [[0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 107, 0, 0]]
m1 = SquareModule(
'Drill', {'drill1': 53, "drill2":106 },
 t1, QUEUE_LENGTH)

t2 = [[0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 164, 0, 0]]
m2 = SquareModule(
```

```
'Robot Arm', {'fuse0': 582, "fuse1":752, "fuse2":850 },
 t2, QUEUE_LENGTH)


t3 = [[0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 112, 0, 0]]
m3 = SquareModule('camera', {'picture': 20},
 t3, QUEUE_LENGTH)


t4 = [[0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 112, 0, 0]]
m4 = SquareModule('Transport0', {},
 t4, QUEUE_LENGTH)

t5 = [[0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0]]
m5 = SquareModule('Package', {'pack': 68},
 t5, QUEUE_LENGTH)

m0.right = m1
m1.right = m2
m2.right = m3
m3.right = m4
m4.right = m5

m2.allow_passthrough = True

r0 = Recipe('NoFuse',
{'load case': set(),
 'drill2': {'load case'},
 'fuse0': {'drill2'},
'picture':{'fuse0'},
 'pack':{'picture'}},
 'Case Loader', 3, 1)

r1 = Recipe('LeftFuse',
```

```
 {'load case': set(),
  'drill1': {'load case'}
, 'fuse1': {'drill1'},
  'picture':{'fuse1'},
'pack':{'picture'}},
'Case Loader', 3, 1)

r2 = Recipe('BothFuse',
 {'load case': set(),
  'drill2': {'load case'},
  'fuse2': {'drill2'},
  'picture':{'fuse2'},
  'pack':{'picture'}},
  'Case Loader', 3, 1)

recipes = [r0 , r1, r2]
modules = [m0, m1, m2, m3, m4, m5]

time, ___, _, _ =
get_best_time(recipes, modules, XML_TEMPLATE, VERIFYTA )
print(time)
```

Code B.1: python script for getting rating of from configuration described in section 3.4.1 running order AllTypes