Task scheduling for dual-arm industrial robots through Constraint Programming

(MiniZinc modeling and solver comparison)

Tommy Kvant
ada09tkv@student.lu.se

March 1, 2015

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jacek Malec, jacek.malec@cs.lth.se
Supervisor: Maj Stenmark, maj.stenmark@cs.lth.se

Examiner: Klas Nilsson, klas.nilsson@cs.lth.se

Abstract

In a society where more and more production becomes automated it demands robots that are as flexible and versatile as humans. Such flexibility demands automatic scheduling of tasks. In this thesis we approach the problem using Constraint Programming and through a case study we present a model for a dual-armed robot that is able to deal with a more flexible workload. We also introduce filters to cut down the runtime of the solver. To evaluate the model we tested it on 6 solvers; G12/FD, JaCoP, Gecode, or-tools, Opturion CPX and Choco3. The results show that the model can produce a solution as good as the one manually implemented for the case study. We introduce filters on the domains of some of the variables and they made an improvement on the runtime for many of the solvers. We also found that the runtime of the solvers varied a lot and could range from several hours to just a few milliseconds using the same data. Unfortunately, in many of the tests the solvers did not complete their searches within the time limit of 4 hours. In some cases when using MiniZinc version 2.0.1, the solvers were not able to read the FlatZinc files. The fastest solver in our tests was Gecode using MiniZinc version 2.0.1.

Keywords: Constraint Programming, MiniZinc, JaCoP, G12, Gecode, or-tools, Opturion CPX, Choco3, scheduling, dual-arm robots

Acknowledgements

I would like to thank my supervisors, Jacek Malec and Maj Stenmark, for their support and constructive feedback.
I would also like to thank Krzysztof Kuchcinski at the Institute of Computer Science, Lund University, for his valuable input on the model.
Lastly I would like to thank Johan Wessén at ABB for giving us access to the data used in [Ejenstam, 2014] and providing help interpreting the data.

Contents

Chapter 1
Introduction

More and more of the production in today's society is getting automated. Product series have short lifespan and the focus of the production must change quickly. It is expensive to have at hand robots for every possible occasion, thus such production is often outsourced to low-wage countries. Often with worse working conditions than in the west. This puts pressure on the robot manufacturers to develop robots that are versatile like humans and thus eliminating the need to have multiple robots to do multiple tasks which will lower the costs and close the gap of what a human and robots are able to do.

Current robot setups usually have one robot performing one task all the time, as opposed to flexible robots which will be changing between many different tasks and assemblies. One of these flexible robots is ABB's robot YuMi®. YuMi®is a dual armed robot made to work alongside humans and able to perform some of the most complex tasks, such as mount a nut or thread a needle[ABB, 2014]. It accomplishes this by using a wide variety of sensors, e.g., force sensor, visual sensors, etc. Usually a robot replaces humans to perform dangerous or heavy tasks, while YuMi®is mainly designed for small parts cooperation with humans, i.e. usually human roles in todays manufacturing environment.

In order to support rapid change-over between tasks, certain problems such as scheduling of the assemblies need to be automated. In this thesis we will be using Constraint Programming (CP) to automate the scheduling process in order to cut down on the scheduling time. Constraint Programming provides a general interface to solve problems without needing to build a complete framework from scratch. Constraint Programming also suits this problem well since scheduling is a classic constraint problem.

1.1 Project goal

The goal of this thesis is to present a generic CP model suitable for a robot such as YuMi®, able to handle the type of jobs YuMi®is able to perform. The scope of the thesis will cover assemblies where the robot can change tools, but only being able to pick up one object at a time. Also, the change between two tools will take the same amount of time regardless whether it is from tool 1 to tool 2, or the other way around. The model will cover the use of trays, fixtures and outputs.

The model will be constructed using the MiniZinc language and tested with 6 CP solvers. We will compare the results from the solvers, both to see how well our model can perform and how well the solvers perform relative to one another.

1.2 Related work

[Drobouchevitch et al., 2006] conclude that the increasing number of machines in a robotic cell causes an explosive growth in combinatorial possibilites. They also provide evidence that a dual-gripper cell is more productive than a single-gripper cell.

[Thörnblad et al., 2013] concludes that when a cell is part of an assembly flow, the targeting of due dates instead of makespan, the total time for the assembly, is to prefer. The reason is that the focusing on makespan runs the risk of exacerbating an already unreliable flow. However, the assembly we want to construct is not a part of a flow, and thus we do not concern ourselves with maintaining a stable flow through the cell, but only to optimize the assembly in the cell.

[Yuan and Xu, 2013] states that Constraint Programming is only effective on small problems of flexible job shop scheduling. In order to effectively solve problems of larger size they suggest to use methods such as large neighbourhood search (LNS) or iterative flattening search. They also show that LNS together with Hybrid Harmonic Search produces good results.

Unfortunately MiniZinc does not support the implementation of custom searches such as LNS. Therefore we try to solve the problem of ineffectiveness by using filter such as those presented by Vilím in [Vilím and Barták, 2002b] [Vilím, 2002] and
[Vilím and Barták, 2002a].

[Ejenstam, 2014] conducted a similar study also on the YuMi®robot. In the study Google or-tools was used to write the model and also implemented *Systematic Tree Search*, *Random Restart* and *Local Search* and compared the results of using different combinations of them. The case study and goal of the study is different from this thesis and therefore the resulting models differ, as is discussed in chapter 6. In the study it was conclude that *Systematic Tree Search* combined with *Random Restart* produced schedules with better results than the reference solution.

Unfortunately, not many comparisons between MiniZinc compatible solvers where found. There is an annual competition held by NICTA where solvers can compete, this is the most comprehensive documentation of the performance of the solvers we have found. Unfortunately, they only present which solver wins a category and no statistics are presented. Hence, no deeper comparison can be made from the result. In the latest competition held,

2014, or-tools won three out of the four gold medals, Opturion CPX won all four silver medals and Choco won three out of the four bronze medals[NICTA, 2014c].

When presenting MiniZinc for the first time, initial tests were also presented comparing, amongst others, G12/FD, Gecode using FlatZinc code and native Gecode. The tests show that MiniZinc was competitive with the native Gecode model and on average the Gecode front-end for FlatZinc was about 200ms faster than the G12/FD
[Nethercote et al., 2007].

Another comparison found was [Becket et al., 2008] where they tested 10 solver on 12 problems. Unfortunately, the only solver tested there that we also used in this thesis is the G12/FD solver. So comparison with our results is hard. Although they did not draw any conclusions, G12/FD seem to fare relatively well compared to the other solvers tested.

1.3 Report structure

First we will present the approach we have taken in the thesis and present the relevant background information in chapter 2. In chapter 3 we will present the case study assembly used in the thesis. Then we will present an in depth view of the model created in chapter 4. In chapter 5 we will present the setup used to evaluate the model and the result of the evaluation. In chapter 6 we will discuss the results and we will come to a conclusion in chapter 7.

Lastly we have two appendices. Appendix A contains constraints which is not crucial for solving of the problem, but is still a part of the model. In appendix B we present all the tools used, which are free to use, and where to acquire them.

Chapter 2
Approach

2.1 Constraint Programming

Constraint programing is a *declarative* paradigm. This means that in contrast to *imperative* paradigm languages, such as C or Java, the focus of solving problems using constraint programming is on specifying the problem and not the algorithm to solve it. However, *declarative* languages, such as Java and C, can be used as a framework Constraint Programming, as in JaCoP, or-tools, and others. One specifies the *domain variables*, or simply variables, and *constraints*. Variables have domains of values, meaning they can take any value in their domain. Variables can often be, depending on language and solver, either integers, floating-points, boolean or symbolic, symbolic being a text or label. For example a symbolic variable representing a week would have the domain $\{Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday\}$, while an integer one could have $\{0, 1, 2, 3, 4, 5, 6\}$.

2.1.1 Constraints

Constraints are set up as relationships between the variables, and thereby limiting the domains of the variables. Integer domains are often used for variables, so for the rest of this section we will assume variables have integer domains. For this domain the following function symbols can be used: $+$, $\times$, $-$ and $\div$. The constraint relation symbols are $=$, $<$, $\leq$, $>$, $\geq$, $\neq$. Together with the function symbols and the constraint relation symbols, one can create simple constraint, called *primitive constraint*. An example of a primitive constraint is $X < Y$, i.e. the values in X's domain has to be lower than in Y's. Primitive constraints can be used to create more complex constraints using the conjunctive connective $\wedge$. An example of this is $X < Y \wedge Y < 10$, i.e. Y has to be less than 10 and X has to be less than Y. Since all constraints have to hold when the model is evaluated, all constraints are implicitly joined by a conjunction. The disjunctive connective $\vee$ is also available and can

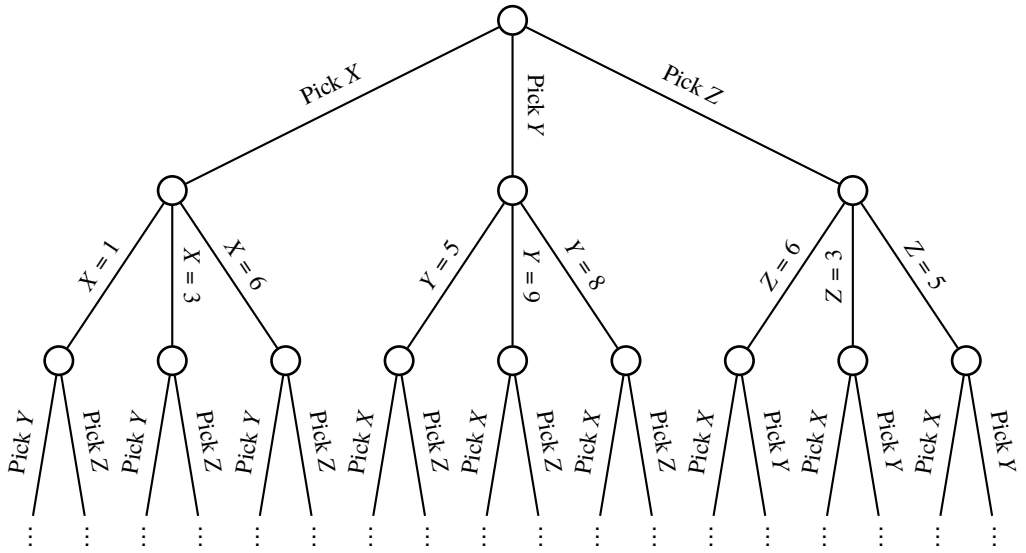

Figure 2.1: The beginning of the search space for the variables X, Y, Z, where $X = \{1, 3, 6\}$ $Y = \{5, 9, 8\}$ $Z = \{6, 3, 5\}$

be used in the same way as $\wedge$.

For example, lets assume we have a problem with two variables, X and Y, $X = 4$ and $Y = \{1..10\}$. Here X has the value 4 and can thereby only take the value 4. Y on the other hand can take the values 1 to 10. This means a solution to this problem can be $X = 4$ and $Y = 1$ or likewise $X = 4$ and $Y = 5$, they are equally correct.
On this problem we can impose a constraint, for example $Y > X$. Now we have set the constraint that Y needs to be larger than X. And since X has a fixed known value we can directly see that $Y > 4$, since $x = 4$. Now with this constraint, we can reduce the domain of Y and now $Y = \{5..10\}$ instead. And now a viable solution can be $X = 4$ and $Y = 7$, but not $X = 4$ and $Y = 3$.

2.1.2 Global constraints

Global constraints are constraints that sets up a relation between an non-fixed number of variables and the global constraints can be reduced to a set of simpler binary constraints [van Hoeve and Katriel, 2006]. They are also context independent[Beldiceanu et al., 2015], which makes them quite convenient to use when modeling and we are using a couple of global constraints that are listed below.

The description of the global constraints in this section come from MiniZinc:s global constraints listing [NICTA, 2014d] and MiniZinc:s tutorial, [Marriott and Stuckey, 2014].

All Different Constraint (`allDifferent`)

The `allDifferent` constraint is pretty straightforward. It takes a set or an array `x` as argument and enforces all variables in `x` to take distinctly different values, i.e. all values will be different.

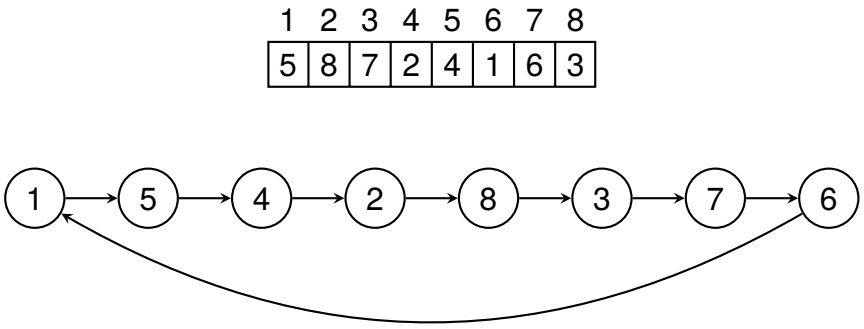

Figure 2.2: Example of the `circuit` constraint enforced on an array of length 8. The visualisation of the array on top. The visualisation of the nodes beneath with arrows from the node to its successor

Circuit Constraint (`circuit`)

The `circuit` constraint takes an array of integers representing nodes, `x`. Each index is representing a node and the index is the number of the node. The value at the index represents the successor of the node at the index. `circuit` enforces the nodes to form a *Hamiltonian circuit*. This means all nodes will be part of the circuit that is formed and no node will have itself as successor. See Figure 2.2 for an example.

Cumulative Constraint (`cumulative`)

The `cumulative` constraint is used to schedule entities that takes a given amount of resources in a system with a known amount of resources available. The constraint takes 5 arguments; an array of start times, an array of durations, an array of resources needed and an integer for how many resources available. So if we have 5 resources available and we have two tasks which requires 3 and 2 resources respectively, they can execute simultaneously. But not if the number of available resources were 3. Each index in the arrays corresponds to an entity.

Global Cardinality constraint (`global_cardinality`)

The `global_cardinality` constraint takes three arguments; an array of variables `x`, an array of integer `cover`, and an array of variables `counts`. The constraint assures that the occurrences of `cover(i)` in `x` is equal to `counts(i)`.

2.1.3 Solver

A Constraint Programming program consists of many of these constraints and variables. When the problem is specified in a model, a *solver* runs the model. The goal of the solver is to satisfy all the constraints, i.e. set the domains of the variables so that they all follow the relationships of the constraints. This is called the *constraint satisfaction problem*, and can be defined as a triple $\langle Z, D, C \rangle$. $Z = \{x_1 \dots x_n\}$ is a finite set of all the variables in the solution, $D(x_i)$, $x_i \in Z$ is a set representing the domain of values the variable x_i can assume, C is the set of constraints imposed on the variables in Z. In order to satisfy all constraints imposed, the solver performs a *search* on the space of possibilities, i.e. the *search space*. The search space has the form of a tree, where each branch is a selection of a variable where the variables domain is reduced into a smaller subset that conforms with the constraints. The solver traverses the tree in search for a solution. When all variables are set to conform with the constraints a solution is found. If the solver reaches a node where a variable domain becomes empty, it has to *backtrack* to a previous node from which it can choose a new variable to set, i.e. traversing a new branch of that node. To make sure that all constraints holds true, called *consistency*, when a change occurs in a variable during search or by propagation itself, the solver performs *propagation*. When a change occurs to a variable, lets call it X, the solver looks at the variables related to this variable through constraints, lets call them Y and Z, and may prune the domains of Y and Z in order to uphold the consistency of the constraints. The solver then propagates onward to the variables related to Y and Z and performs the same procedure. More about solvers can be found in [Tsang, 1993], [Marriott and Stuckey, 1998] and [The G12 Team, 2014].

2.1.4 Reified Constraints

Reified constraints are constraints that couple a primitive constraint with a boolean variable and provide a relationship between the both. An example could be the constraint c and a boolean variable B, the relationship between the both could be $c \Leftrightarrow B$. This says that if c holds, $B = true$ and if $\neg c$ holds, $B = false$. This form of expression can be very useful in expressing complex relations and constraints [Marriott and Stuckey, 1998].

Although it is a convenient way of expressing complex relations, it has its disadvantages. Reified constraints can be inefficient since every reified constraint needs to be propagated all the time. Reified constraints can also propagate poorly, for example if a variable occurs multiple times in an expression [Jefferson et al., 2010]. Due to this, we have tried to avoid direct reified constraints in the MiniZinc code in hope of reducing the total amount of reified constraints in the resulting code.

2.1.5 Branching Heuristics

Branching heuristics is what decides what value in a domain to branch on. It can be declared by the one programming the model and can play a significant role in the effectiveness of the model. An example of a common branching heuristic is `indomain_min` which branches on the smallest value in the domain and if backtracked choses the next smallest value the next time, i.e. working its way up from the smallest value. The opposite of `indomain_min` is `indomain_max`, which starts in the other end of the domain.

Another common branching heuristic is `indomain_median` which branches on the median value of the domain and if backtracked branches on the values on either side of the median and works its way outwards. There are more branching heuristics available and which branching heuristics are available depend on the solver.

2.2 Job-shop scheduling problem

The job shop problem can be described as n jobs of varying size containing a number of operations to be executed in a certain order that needs to be scheduled on m identical machines. Commonly the goal is to minimize the total time for the schedule, called the *makespan*. The traveling salesman problem is a version of the job shop problem where $m = 1$. [Garey et al., 1976] shows that the job shop problem is NP-complete for $m \geq 2$ and $n \geq 3$, hence more complex versions of the job shop problem will be at least this hard.

As described above, the schedule is composed of jobs containing operations. This is the usual way of describing it in the literature, but we will look at it in a slightly different way. Instead of looking at many jobs, we will focus on one job and the operations within that job. In this thesis we will refer to these operations as tasks.

An extension of the job shop problem is the flexible job shop problem. In it, tasks are not locked to be scheduled on a particular machine, but can be scheduled for any of the machines [Thörnblad et al., 2013]. This increases the complexity of the problem.

Yet another extension of the job shop problem is the job shop problem with sequence-dependent setups. This means the time for a task is not just the time it takes to execute the task itself, but also the time it takes to set up the machine, depending on the previous task, in order to execute the task at hand. This is also something that increases the complexity compared to the basic job shop problem.

Our case will be a combination of the flexible job shop problem and the job shop problem with sequence-dependent setup times since, as will be described later, we can change the tools of the machines. The ability to change tools means that all machines can execute all tasks, and the change of tool takes time which means we get a sequence-dependence.

2.3 MiniZinc

There are many solvers for CP problems, but they all use different languages and as a modeler it might be of interest to test how well your model performs on different solvers. To eliminate the need to rewrite models to fit the language of the different solver in order to perform a comparison, MiniZinc was introduced. MiniZinc is a modeling language similar to *Optimized Programming Language* (OPL), but is scaled down and lacks some of OPL's features. MiniZinc's strength lies in that it is coupled with another language called FLatZinc. The difference between MiniZinc and FlatZinc is that MiniZinc is a medium-level language where it is easy for modelers to express themselves and FlatZinc is a low-level language that is easy for interpreters to parse. There is a translator from MiniZinc to

FlatZinc provided, the translation is called *flattening*. MiniZinc provides a set of already defined constraints that solvers can use, however, the translator takes in consideration the solver that is going to be used and can apply custom versions of the constraints specified for that particular solver. [Nethercote et al., 2007]

Although MiniZinc aims at being a standard language in CP, it does not have support for defining custom search algorithms, as many other languages do. This means we cannot utilize algorithms for random restart, local search, etc. To be clear, the solver is the one performing the search, but in some Constraint Programming languages we can define how that search is to be performed [Nethercote et al., 2007]

2.4 Solvers

This thesis will test the model using 6 different solvers; *G12*, *JaCoP*, *Gecode*, *OR-tools*, *Opturion CPX* and *Choco3*. There where three requirements considered when we chose the solvers:

- The solver has to have a FlatZinc parser
- The item has to be free to acquire, either via open source, free license or free academic license.

The model was initially tested during the implementation phase using G12/FD, but after a while G12/FD was unable to produce results and a switch was made to JaCoP. In other words, the model was developed and tested using JaCoP.

2.4.1 G12/FD

G12/FD is a finite domain solver provided by the G12 team, the creators of MiniZinc. It is implemented in Mercury and is the default solver for the G12 FlatZinc interpreter [Becket et al., 2008] [NICTA, 2014c].

2.4.2 JaCoP

JaCoP stands for Java Constraint Programming solver, and is an open source Java library for Constraint Programming that is available under the GNU Affero GPL license. It has been developed since 2001, mainly by Krzysztof Kuchcinski and Radoslaw Szymanek. The library provides many global constraints in order to make modeling more efficient. It is used by researchers all around the world and has proven its efficiency by winning silver medal in the MiniZinc Challenge [Szymanek, 2010b] [Szymanek, 2010a].

2.4.3 Gecode

Gecode is a free constraint solver under the MIT License implemented in C++. It officially provide a MiniZinc interface, but many external projects provides additional interfaces. One of its strengths is that it can perform parallel searches using multiple cores and this gives the solver great efficiency. This has lead to Gecode winning all the gold

medals of the MiniZinc challenge in all 5 consecutive years between 2008 and 2012. [Schulte et al., 2014]

2.4.4 or-tools

or-tools is an open source constraint solver under the Apache License 2.0 implemented in C++. or-tools is developed by Google and is part of their Operational Research. In addition to C++ and MiniZinc, or-tools also has interfaces for Python, Java and C# [van Omme et al., 2014].

2.4.5 Opturion CPX

Opturion CPX is a constraint solver developed by Opturion Pty Ltd, a commercial outcome of the G12 project. The same ones that created G12/FD, MiniZinc and FlatZinc. Opturion CPX is a commercial product and therefore not free. Although, they provide academic licenses which was used for the thesis. Since it originated from G12, the language for implementing models is MiniZinc.

Unlike the other solvers used, Opturion CPX is not a pure *finite domain* solver, but rather a combination of solving techniques from CP and propositional logic (SAT). This makes CPX extremely efficient in solving large models. It is said that because Opturion CPX only generates propositional variables needed for the search, the search is not necessarily slowed down due to large domains. Proof of this can be shown by the number of awards claimed in the MiniZinc challange [Opturion Pty Ltd, 2013] [Opturion Pty Ltd, 2014a] [Opturion Pty Ltd, 2014b].

2.4.6 Choco3

Choco3 is a finite domain [Fages et al., 2014] constraint solver implemented in Java and it is free under the BSD license. The development of Choco has been going on since the early 2000s and Choco3 is the latest version. Although sharing the name, Choco3 is not the same system as its predecessor Choco2, but a complete new implementation of the previous system [Charles Prud'homme, 2014].

Chapter 3
Case Study

In order to develop and test the model, we have chosen to focus on one representative case study. In this case study the robot is assembling an enclosed emergency stop button used in industrial environments, see Figure 3.1. The assembly is presented in [Stolt et al., 2013] and has an associated video of the assembly[1]. Please note that this is not the video used to extract the times for the tasks, so the times may differ from the ones used. Also, only the assembly of one stop button is performed in this thesis, not several as in this video.

This assembly is composed of 5 components; a top, a button, a nut, a switch and a bottom. A combination of components that is not the complete assembly we will call a sub-assembly. The button needs to be inserted in the top and then the nut needs to be screwed onto the underside of the button in order to secure it to the top. The switch needs to be mounted in the bottom and, lastly, the top part, with button and nut, needs to be mounted on the bottom with the switch. In Figure 3.2 we see the top-button-nut assembly to the right and the bottom-switch assembly to the left. Note that the screws in the figure are not part of the case study assembly.

The assembly includes objects such as trays and fixtures. A tray is a holder where components reside until they are needed in the assembly. A fixture is a holder in which components can be put so that another components can be mounted on it.

21 steps have been identified as needed for the assembly and they are taken from a video of an existing assembly created by hand. For an illustration of the order of the assembly steps, see Figure 3.2 The steps are as follows:

Take top Takes the top component from its tray

Put top in fixture Puts the taken top component in a fixture

Take button Takes the button component from its tray

[1] `http://www.youtube.com/watch?v=7JgdbFW5mEg`

Figure 3.1: Picture of the button used in the case study. The screws are not part of the case study assembly

Mount button on top Mounts the taken button component onto the top component in a fixture

Angle top-button Angles the sub-assembly that is the top and button component, from here on called top-button, so it can be supported by the other machine.

Lift top-button, hold top button Lifts the top-button by holding the button

Lift top-button, support Lifts the top-button by supporting the top from underneath

Turn top-button Turns the top-button by holding the button

Take nut Takes the nut from its tray

Mount nut on top-button, hold Mounts the nut on top-button while holding the button

Mount nut on top-button, mount Mounts the nut on the top-button holding and screwing the nut. The sub-assembly created by the top-button and the nut is here on after called top-button-nut

Fixate top-button-nut Fixates the top-button-nut using the side of a fixture in order to get it straight.

Put top-button-nut in top tray The top-button-nut is put in the top tray in order to put it away for a while to be picked up later.

Take top-button-nut from top tray Takes the top-button-nut from the top tray where it was previously put.

Take bottom Takes the bottom component from its tray.

Put bottom in fixture Puts the bottom component in a fixture.

Take switch Takes the switch component from its tray.

Mount switch in bottom Mounts the switch on the bottom in the fixture the bottom was put in. The sub-assembly created here will be called bottom-switch.

Take bottom-switch Takes the created bottom-switch from the fixture.

Put bottom-switch on table Puts the bottom-switch on the table.

Mount top-button-nut on bottom-switch Mounts the top-button-nut on the button-switch on the table.

Most of the steps are self explanatory, but there are a few steps which are quite special that might need some more explanation. In steps **Mount nut on top-button, hold** and **Mount nut on top-button, mount** the button component has been put in the hole of the top and needs to secured using the nut component. To do we utilize somthing that is special for YuMi®, using both arms in order to mount the nut. This is done mid air with one arm holding the top-button by gripping the button part, having it upside-down compared to how it was in the fixture, and with the other arm screwing the nut in place. The preparation for this starts at the end of step **Mount button on top**, where the sub-assembly top-button was just created, but the button is only loosely sitting in the hole of the top. The sub-assembly is angled in task **Angle top-button** about 45° in order to create a gap under the sub-assembly so the other arm can reach under the sub-assembly and help lift it from the fixture in tasks **Lift top-button, support** and **Lift top-button, hold top button**. Finally the top-button is rotated by only holding it with one arm in the button part of the sub-assembly and are now ready for the nut to be mounted. It should be noted that since the operations of lifting the top-button and mounting the nut takes two arms and we have here split the operations into two tasks each, one for each arm, the tasks **Mount nut on top-button, hold** and **Mount nut on top-button, mount** needs to be performed at the same time. The same goes for **Lift top-button, support** and **Lift top-button, hold top button**.

As mentioned before, the steps listed are taken from a video of an assembly. This makes the times used in the case study approximated. First the times where approximated using seconds as the unit of time. But it showed to be hard to approximate some of the tasks as they sometimes where under 1 second. Because of this and to get better print outs of the assembly using `SchedPrinter` (see appendix B) we multiplied the times we could estimate well from the video by 5 and approximated the other tasks as well as we could. This means that the real times for the tasks in the case study, and in the files mentioned in appendix B, are $1/5$ of the time presented. Because of this, comparing the time from the solvers with the time of the manual assembly is harder. We had to approximate the time of the manual assembly using the times we have approximated. By analysing the video and using the approximated times we got a time of 516 time units for the manual assembly.

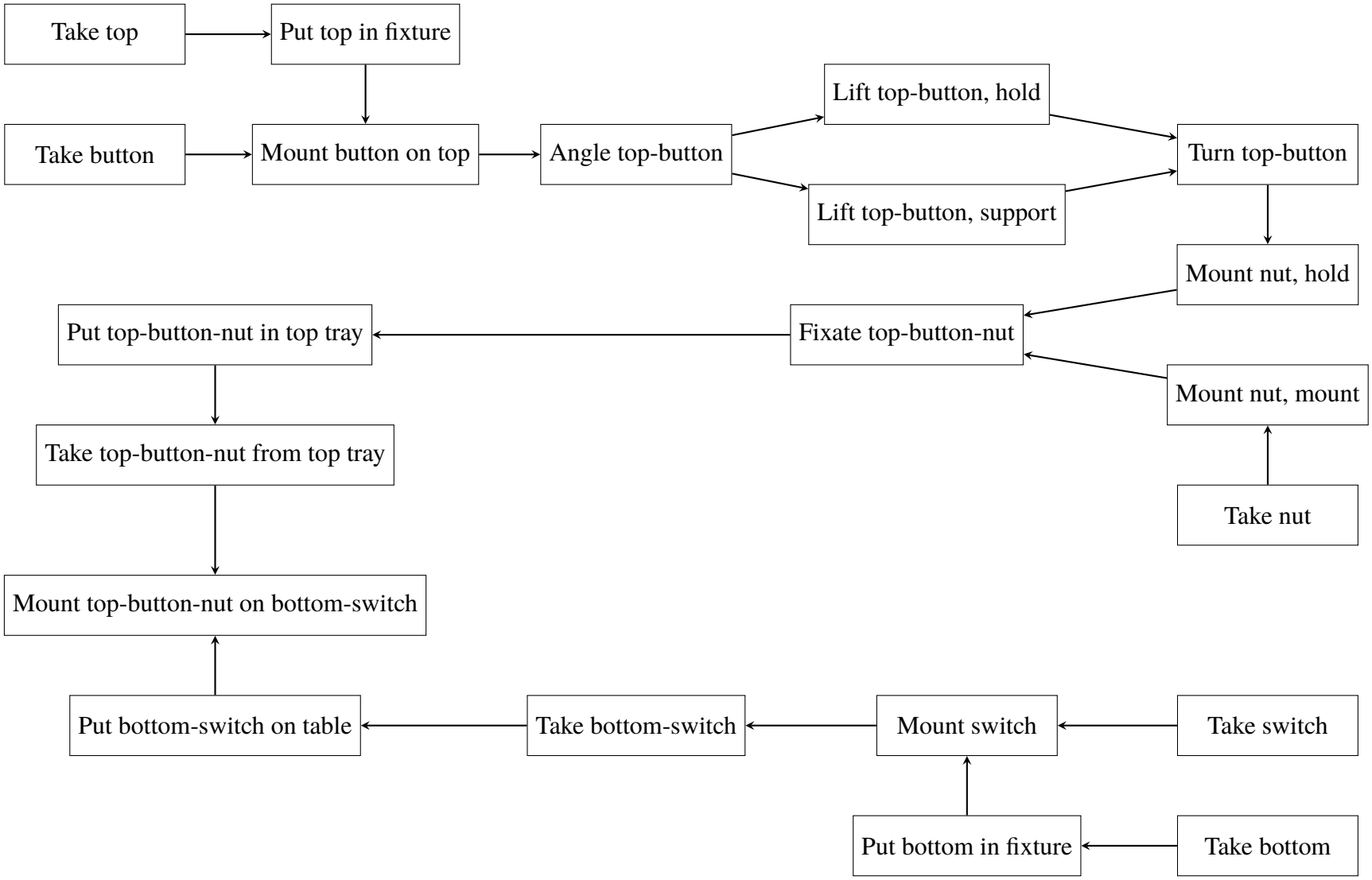

Figure 3.2: The case study assembly

Chapter 4
Model

This model is inspired by the work in [Ejenstam, 2014]. That model is centered around work performed in fixtures. So tasks can easily be labeled *tray* if it uses a tray, *fixture* if it uses a fixture, etc. These are common robot in cell assembly procedures; take a component from a tray, put it in a fixture, get another component, mount the component on the the component in the fixture. But YuMi can perform much more complex tasks than that. We want to be able to schedule mounting tasks that do not incorporate a fixture. We have used a similar way of generalizing tasks by labeling them with *tray*, *fixture*, etc.

Before going into details, we will give a brief overview of how the scheduling works. The model presented is centered around tasks. A task is an action that manipulates a component in some way and is performed at a certain spacial coordinate in the room. However, the model does not care about the exact coordinates, but rather the duration it takes to travel between the coordinates. This time is used to establish how long the move from one task to the next will take. These moves are present for all tasks. If two tasks are performed at the same location, the move time will be 0. The times needs to be calculated beforehand and put in a matrix which is used to generate the input file for the model. The procedure is described in appendix B.

In the model each robot arm/manipulator is called a machine. Hence, a two-armed robot is modeled in the same way as two one-armed robots. To compensate for the placement of the machines there are variables that can be set as shown below. The arms can be equipped with certain tools, different tasks can require different tools and the arms can during execution change tools. The change of a tool is incorporated in the move from one task to another. This is part of what the model will try to decide, when in the assembly should we put the changes between the tools. If a change occurs between two tasks, it will be shown by the move time being extended with the duration of a tool change. To both know how long the move between two tasks will be and if there needs to be a tool change, we need to know which task comes before another task, i.e. the predecessor.

The goal of the assembly is to assemble components into sub-assemblies and further into a final assembly. All the intermediate assemblies before the final assembly are called sub-assemblies. For reasons explained further down, we will in this thesis call components fed from the outside into the assembly, such as buttons, for *primitve* components instead of just components.

The tools used to generate the data used in this thesis are free to use and are described in appendix B. The complete model file used can be found at `https://github.com/Arclights/Thesis-Tools` under *Data*.

4.1 Variables

The solver takes a description of the robot cell in the form of a MiniZinc data file. The file describes the number of available arms, tools, trays, fixtures etc. These are *model variables* and they will be explained further down among the *static variables*.

4.1.1 Model Variables

- $nbrTasks$
- $nbrMachines$
- $nbrTools$
- $nbrTrays$
- $nbrFixtures$
- $nbrComponents$
- $nbrOutputs$
- $nbrConcurrentGroups$
- $nbrOrderedGroups$
- $tray(t)$
- $output(t)$
- $fixture(t)$
- $componentsUsed(t)$
- $mounting$
- $taking$
- $moving$
- $putting$
- $concurrentTasks(k)$
- $orderedGroup(k)$
- $toolNeeded(t)$
- $duration(t)$
- $taskSubComponents(t)$
- $taskCompleteSubComponents(t)$
- $timeMatrix3D(t1,\ t2)$
- $tasksOutOfRange(t)$

task	task	task	task	sTask1	sTask2	gTask1	gTask2

Figure 4.1: An example of the tasks and start and goal tasks seen as an array for an assembly with 4 tasks and 2 machines

4.1.2 Static variables

Static variables are variables that have a fixed value, or is a set or list containing fixed values.

First we define the number of tasks to be scheduled. Each task is identified.

$$nbrTasks \in \{1, \ldots, 2^{32} - 1\} \tag{4.1}$$

$$tasks = \{1, \ldots, nbrTasks\} \tag{4.2}$$

As mentioned, this model is based on the technique of using predecessors to determine which task comes directly before another. This creates the need to have source and a sink node for each machine. We call them start tasks and goal tasks. As they are not provided as parameters, the model creates them and give them identifiers with numbers greater than the tasks to be scheduled. Each machine has to have a start task and a goal task. This means that there are as many start and goal tasks as there are machines. They are arranged so that all the start tasks come first and then all the goal tasks. One can easily find the start task for a machine by $nbrTasks + m$, where m is the machine in question. It is also easy to find the matching goal task by $nbrTask + m + nbrMachines$. If one thinks of the tasks, start and goal tasks as an array where the index is the number of the task, then it would look like in Figure 4.1.

$$startTasks = \{nbrTasks + 1, \ldots, nbrTasks + nbrMachines\} \tag{4.3}$$

$$goalTasks = \{nbrTasks + nbrMachines + 1, \ldots, nbrTasks + nbrMachines \times 2\} \tag{4.4}$$

We group together all tasks in one set in order for a more readable notation further down.

$$allTasks = tasks \cup startTasks \cup goalTasks \tag{4.5}$$

Here we define the machines available for the assembly. A machine in this model is an arm.

$$nbrMachines \in \{1, \ldots, 2^{32} - 1\} \tag{4.6}$$

$$machines = \{1, \ldots, nbrMachines\} \tag{4.7}$$

These are the tools that can be fitted on an arm. The model assumes that there is a set of $nbrTools$ for each machine. I.e. if $nbrTools = 2$ and $nbrMachines = 2$, there is a set of tool 1 and tool 2 for machine 1, and another set of tools 1 and 2 for machine 2. There cannot be a combination of tools such as, for example, only tool 1 for machine 1 and a set of tools 1 and 2 for machine 2.

$toolNeeded(t)$ defines the tool that task t needs.

$$nbrTools \in \{1, \ldots, 2^{32} - 1\} \tag{4.8}$$

$$tools = \{1, \ldots, nbrTools\} \tag{4.9}$$

$$toolNeeded(t) \in tools,\ t \in tasks \tag{4.10}$$

nbrComponents defines the number of components used. All components need to be uniquely identified in the assembly, so even if we use 4 identical screws in an assembly, we need to define all 4 screws. As mentioned before, we distinguish between components and *primitve* components. The reason for this is that in the model we do not distinguish between a *primitve* component and a sub-assembly, they are the same. And in the model we call them components. The reason for this is because we found it easier to only have one sort of object to deal with when it comes to what will be assembled, instead of two. This means that the final assembly is also a component, i.e. the product produced by the assembly is a component. In other words, in this thesis *primitve* components and sub-assemblies are sub sets of components.

$componentsUsed(t)$ defines the set of components task t uses. A task usually only uses one component at a time, but uses two in the case of mounting tasks, the mounted component and the component mounted on.

To know when a sub-assembly is created we set it as *compoentCreated* for the task where it is created. This cannot happen anywhere else than in a mount task, although there is no check in the model for it. If no component is created in a task, $componentCreated = 0$.

$$nbrComponents \in \{1, \ldots, 2^{32} - 1\} \tag{4.11}$$

$$components = \{1, \ldots, nbrComponents\} \tag{4.12}$$

$$componentsUsed(t) \subset components,\ t \in tasks \tag{4.13}$$

$$componentCreated(t) \in components \cup \{0\},\ t \in tasks \tag{4.14}$$

Since components also can be sub-assemblies, it means a component can have subcomponents. These have been grouped in different groups to assist the constraints.

$taskSubComponents(t)$ is the set of components that make up the subcomponents for the components used in task t. One can think of the subcomponents as layers with the component on top, call it origin component, and the layer below are the components that make up that component, and so on. $taskSubComponents(t)$ contains the components one layer down, if the component itself is not a *primitve* component. In that case, $taskSubComponents(t)$ contains that component instead. See Figure 4.2 for an example.

$$taskSubComponents(t) \subset components,\ t \in tasks \tag{4.15}$$

To use the layer metaphor again, $taskCompleteSubComponents(t)$ contains all the layers below the origin component, for all the components in task t, not including the origin component itself. If the origin component is a *primitve* component, the set is empty. See Figure 4.3 for an example.

$$taskCompleteSubComponents(t) \subset components,\ t \in tasks \tag{4.16}$$

$subComponents(c)$ contains only the the *primitve* subcomponents for component c, one layer down. If c is a *primitve* component or is only made of sub-assemblies, the set is empty.

$$subComponents(c) \subset components,\ c \in components \tag{4.17}$$

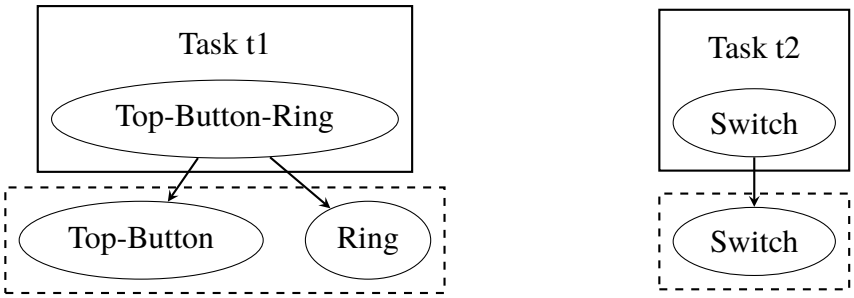

Figure 4.2: *taskSubComponents* for two tasks, $t1$ and $t2$. $t1$ contains a sub-assembly, Top-Button-Nut, and $t2$ contains a primitive component, Switch. The *taskSubComponents* for each task is shown in the dashed box beneath them.

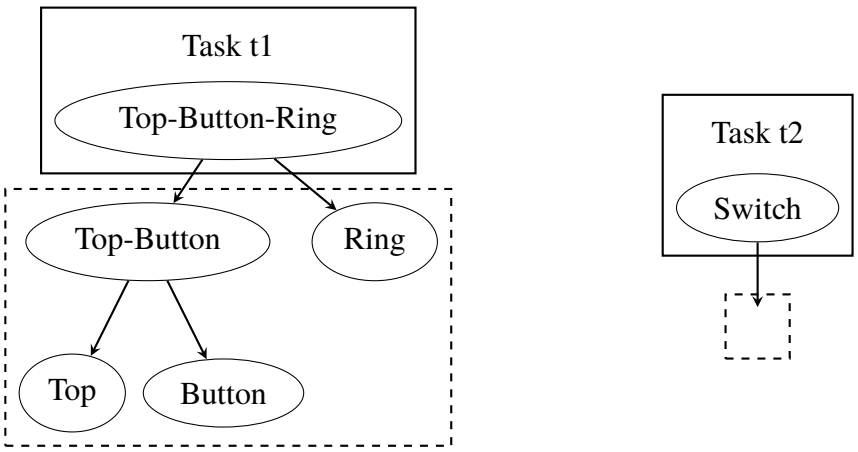

Figure 4.3: *taskCompleteSubComponents* for two tasks, $t1$ and $t2$. One contains a sub-assembly, Top-Button-Nut, and the other contains a primitive component, Switch. The *taskCompleteSubComponents* for each task is shown in the dashed box beneath them.

Trays are used to hold components until we need them in the assembly. It can be that the tray holds the components from the beginning, as with *primitve* components fed to the assembly, or it can be a sub-assembly put there during the assembly to be picked up again later. Each *primitve* component has its own tray, so we can have a button tray, a cover tray, etc.

$tray(t)$ is the tray task t uses. If no tray is used by the task, $tray(t) = 0$.

$$nbrTrays \in \{1, \ldots, 2^{32} - 1\} \tag{4.18}$$

$$trays = \{1, \ldots, nbrTrays\} \tag{4.19}$$

$$tray(t) \in trays \cup \{0\},\ t \in tasks \tag{4.20}$$

$fixtures$ defines the fixtures available in the assembly. A fixture is primarily used to hold a component in order for another component to be mounted on that component. Although, as was shown in the case study in chapter 3, the fixture can be used for purposes other than just holding components.

$fixture(t)$ is the fixture task t uses. If no fixture is used by the task, $fixture(t) = 0$

$$nbrFixtures \in \{1, \ldots, 2^{32} - 1\} \tag{4.21}$$

$$fixtures = \{1, \ldots, nbrFixtures\} \tag{4.22}$$

$$fixture(t) \in fixtures \cup \{0\},\ t \in tasks \tag{4.23}$$

$outputs$ defines the outputs available. An output is the final stage for a component in an assembly. After it is put here, it will not be removed. Although, there can still be other components mounted on the component put on the output. In that respect an output can be viewed as a fixture, only that the components put there can not be removed.

$output(t)$ is the output used by task t. If no output is used by the task, $output(t) = 0$.

$$nbrOutputs \in \{1, \ldots, 2^{32} - 1\} \tag{4.24}$$

$$outputs = \{1, \ldots, nbrOutputs\} \tag{4.25}$$

$$output(t) \in outputs \cup \{0\},\ t \in tasks \tag{4.26}$$

$concurrentTasks(k)$ is the k:th concurrent group among the concurrent groups defined. A concurrent group is a group of tasks that has to be performed at the same time. Hence, a concurrent group can not be larger than the amount of machines available, although, there is no check for it in the model.

$$nbrConcurrentGroups \in \{1, \ldots, 2^{32} - 1\} \tag{4.27}$$

$$concurrentGroups = \{1, \ldots, nbrConcurrentGroups\} \tag{4.28}$$

$$concurrentTasks(k) \subset tasks,\ k \in concurrentGroups \tag{4.29}$$

$orderedGroup(k)$ is the k:th ordered group specified, there are $nbrOrderedGroups$ ordered groups. An ordered group is an array of tasks that have to come in a very specific order. An example of this could be if an assembly has many move tasks that need to be performed one after another in order to make some intricate movement. As seen in section 4.2, we can reason about the relation between tasks if they use a certain component and

are a certain kind of action. But we cannot reason about two move tasks, there is no way to tell which should come before the other based on the component they use.

$orderedGroup(k)$ is an array and the tasks in it will be scheduled in the order they come in the array. All the tasks in the group will be performed on the same machine.

If one wants to access a certain task in a group, one can use $ordered(k, i)$ to access the i:th element of the k:th group.

$orderedSet$ is the set of all tasks included in some ordered group.

$$nbrOrderedGroups \in \{1, \ldots, 2^{32} - 1\} \tag{4.30}$$

$$orderedGroups = \{1, \ldots, nbrOrderedGroups\} \tag{4.31}$$

$$orderedGroup(k) \subset tasks,\ k \in orderedGroups \tag{4.32}$$

$$ordered(k, i) \in tasks,\ i \in \{1, \ldots, |orderedGroup(k)|\},\ k \in orderedGroups \tag{4.33}$$

$$orderedSet = \bigcup_{\forall k \in orderedGroups} orderedGroup(k),\ orderedSet \subset tasks \tag{4.34}$$

$tray(t)$, $output(t)$ and $fixture(t)$ cannot be set at the same time for a task, since that would mean that the task is performed at two locations at the same time, although this is not checked by the model. The only restriction for what kind of tasks can be performed using these containers is that outputs cannot be used by take tasks and trays cannot be used by a mount tasks. If an assembly contains these combinations, the output or tray should be changed to a fixture.

Each task performed can be classified as either a mount task, a take task, a move task or a put task, but only as one of them.

Taking A task that picks up a component is a taking task. The location of the component is specified by either a tray or a fixture, but not an output since there is no reason to pick up something that has been placed on an output.

Mounting A task that mounts a component on another component is a mounting task. This assumes that the component to mount is picked up and in the hand. The location of the component to mount on is defined by either a fixture or an output.

Putting A task that puts a component somewhere is a putting task. Where a component is put is defined by either a fixture, a tray or an output.

Moving A task that moves a component from one place to another is a moving task. The model already puts in moves between tasks and if, for example, the first task is a take task and the second task is a put task, the move in between them is essentially a move that moves a component from one place to the another. Although, sometimes it can be handy to define a task that explicitly moves a component. An example of that can be if one wants to spin a component around. Then one can specify a take task in order to pick up the component, a move task to turn it, and a put task to put the component back. In this case there will be three moves of the component; one to move from the take task to the move task, the move task itself, and a move from the move task to the put task.

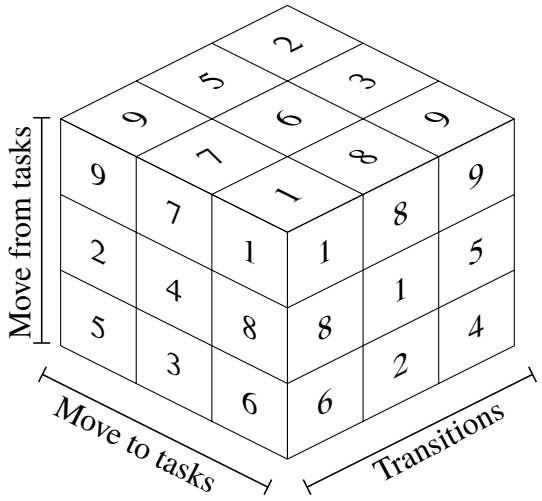

Figure 4.4: The timeMatrix3D

$$mounting \subset tasks \tag{4.35}$$

$$taking \subset tasks \tag{4.36}$$

$$moving \subset tasks \tag{4.37}$$

$$putting \subset tasks \tag{4.38}$$

$putting(c)$, $mounting(c)$, $taking(c)$ and $moving(c)$ are subsets of respective set above based on the component involved.

$$putting(c) \subset putting,\ c \in components \tag{4.39}$$

$$mounting(c) \subset mounting,\ c \in components \tag{4.40}$$

$$taking(c) \subset taking,\ c \in components \tag{4.41}$$

$$moving(c) \subset moving,\ c \in components \tag{4.42}$$

$duration(t)$ is simply the duration of task t.

$$duration(t) \in \{0, \ldots, 2^{32} - 1\},\ t \in tasks \tag{4.43}$$

For the model to decide how long a move between two tasks should take and whether there should be a change of tool in between, a matrix is used, $timeMatrix3D$, see Figure 4.4. This is a 3-dimensional matrix and contains the times for moving between all the tasks depending on what tool change occurs. On its y-axis it has the tasks to move from, on the x-axis the tasks to move to, and on the z-axis the different transitions between tools that can occur. There are $nbrMachines$ more rows on the y-axis than there are columns on the x-axis. This is because we also account for the starting position of the machines, so each start task has move times associated with them for moving to the other tasks. As the matrix is constructed the way it is, there is no move times between start tasks.

$timeMatrixDepth$ is the length of the z-axis, i.e. the depth of the matrix. It should be said that the reason for using the method described below is to reduce the size of the matrix and avoid redundancy.

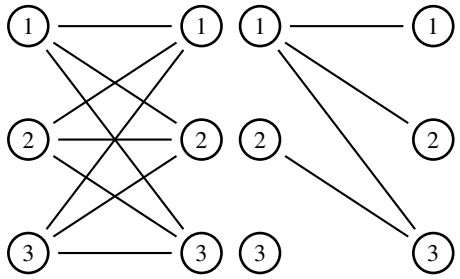

Figure 4.5: All the transitions between the tool states to the left. The reduced number of transitions between the tool states to the right

What we mean with "different transitions" is easiest shown through an example. Let us say we have 3 tools available for each machine. We consider each tool state as a node in a graph, see Figure 4.5, with the old tool state to the left and the new tool state to the right. Between them we can draw the different ways we can change state. The we start to consider which ones we actually need. We can change from tool 1 to tool 1, which is not changing tool at all. The same can be done for tool 2, but not changing tool here costs just as much time as with tool 1. So the change from tool 1 to itself covers not changing tool for this tool, as well as for all the other tools, thereby we only need to keep track of one of these changes. We can also change from tool 1 to tool 2. And we can change back from 2 to 1, although here in the model we assume the change from one tool to another takes the same time the other way around as well. Therefore, we consider the change from tool 1 to tool 2 the same as from tool 2 to tool 1, and only keep track of one of them. If we keep considering the rest of the transitions this way, we will end up with a reduced number of transitions, in our case 4, see Figure 4.5

It is clear that, $timeMatrixDepth$ obeys the equation (4.44).

$$timeMatrixDepth = \frac{n^2 - n + 2}{2},\ n = nbrTools \tag{4.44}$$

$$\begin{aligned} timeMatrix3D(t(from), t(to), k) &\in \{0, \ldots, 2^{32} - 1\}, \\ t(from) &\in tasks \cup startTasks, \\ t(to) &\in tasks,\ k \in \{0, \ldots, timeMatrixDepth\} \end{aligned} \tag{4.45}$$

Depending on the physical layout of the assembly, sometimes not all the tasks can be done by all machines. It could be that the machines would collide or simply that the spatial location is out of reach for the machine. In those cases we can specify that tasks are out of hand for a specific machine. This is the only time when we distinguish between the two machines and connect the machine in the model model with the machine in the real world. In all other aspects the machines in the model are identical and have the potential to perform the same work.

$$taskOutOfRange(m) \subset tasks,\ m \in machines \tag{4.46}$$

4.1.3 Decision variables

Decision variables are variables that can take many values. It is these values that the solver sets out to determine in order to solve the problem.
The model has to decide which task uses which machine.

$$usingMachine(t) \in machines,\ t \in tasks \tag{4.47}$$

Each task has a predecessor that tells the model what other task comes right before the task in question on the same machine.

$$pred(t) \in allTasks,\ t \in allTasks \tag{4.48}$$

In order to create an upper limit for variables dealing with time, we create a rough upper limit of the complete assembly. It simply takes the longest duration for a task and the longest duration for a move between tasks and assert it for all the tasks.

$$\begin{aligned} maxE =& (max(\{duration(t) : t \in tasks\}) + \\ & max(\{timeMatrix3D(t_1, t_2, k) : \\ & \quad \forall t_1 \in tasks \cup startTasks, \\ & \quad \forall t_2 \in tasks, \\ & \quad \forall k \in \{0, \ldots, timeMatrixDepth\}\}) \times nbrTasks \end{aligned} \tag{4.49}$$

Each task has to have a start time. We set it to be anywhere between time 0 and the maximum possible end calculated before.

To simplify notation we also introduce one more variable called $end(t)$. It is the time when task t ends and is simply the sum of the start and the duration of the task.

$$start(t) \in \{0, \ldots, maxE\},\ t \in allTasks \tag{4.50}$$

$$end(t) = start(t) + duration(t),\ t \in allTasks \tag{4.51}$$

As mentioned before, each task has a move time connected to it since it takes a certain amount of time to move from one task to another. Since this time depends on both what task comes before it and what tools are needed for both of the tasks, the duration for the move is a decision variable as opposed to the duration for the task itself.

$$moveDuration(t) \in \{0, \ldots, maxE\},\ t \in allTasks \tag{4.52}$$

$$moveStart(t) \in \{0, \ldots, maxE\},\ t \in allTasks \tag{4.53}$$

$$moveEnd(t) = moveStart(t) + moveDuration(t),\ t \in allTasks \tag{4.54}$$

Since the goal of the assembly is to complete the assembly in as little time as possible, we set up a variable for it, *makespan*. It is this variable the solver will try to minimize.

$$makespan \in \{0, \ldots, maxE\} \tag{4.55}$$

The last variable is for determine what tool should be used for a task. With $toolNeeded$ we specify what tool is needed for the specific task. But we do not need to specify a tool if the task does not need any specific tool. That is why we need to determine what tool should be used for those tasks. Leaving the option open by not specifying any particular tool opens up for optimisations since it could mean we can avoid costly tool changes.

$$toolUsed(t) \in tools,\ t \in allTasks \tag{4.56}$$

4.2 Constraints

In this section some of the most important constraints for the model will be described. For a full list of used constraints see *Appendix A*, while for the MiniZinc code see *Appendix B*.

makespan should represent the total time of the whole assembly. That means it should be equal to the largest end time among all the tasks. We can enforce that by limiting the end time for each task to be less or equal to the *makespan*.

$$(\forall t \in tasks)\ end(t) \leq makespan \tag{4.57}$$

Start and goal tasks are special tasks since they act as source and sink nodes. This means they never get scheduled in time as ordinary tasks, we set them to all start at time 0 and they do not have a duration variable, since they do not take up any time. We also assign them to machines so each start and goal task pair have their own machine from the start.

$$(\forall t \in startTasks \cup goalTasks)\ start(t) = 0 \tag{4.58}$$

$$\begin{gathered}(\forall m \in machines)\ usingMachine(nbrTasks + m) = m \\ \wedge\ usingMachine(nbrTasks + nbrMachines + m) = m\end{gathered} \tag{4.59}$$

We enforce the $tasksOutOfRange(m)$ variables by simply saying that the tasks in the variable can not be assigned the machine m.

$$(\forall m \in machines)\ (\forall t \in tasksOutOfRange(m))\ usingMachine(t) \neq m \tag{4.60}$$

As said before, the $toolNeeded$ contains what tool is needed for a task. We need to translate it into what tool is used. It is done by simply taking the value from $toolNeeded$ and assigning it to $toolUsed$ for the tasks where a tool is specified, i.e. $toolNeeded$ is not 0.

$$(\forall t \in tasks,\ toolNeeded(t) \neq 0)\ toolUsed(t) = toolNeeded(t) \tag{4.61}$$

4.2.1 Precedences

These constraints deals with the order in time in which the tasks have to come.
A very fundamental part of the relation between a task and the move to it is that we cannot start a task before we have moved to it.

$$(\forall t \in tasks)\ Start(t) \geq moveEnd(t) \tag{4.62}$$

If we want to mount two components together, we first have to put the first component in a fixture before we can mount the other component on it. Hence, the put task has to end before we can start with the mount task.

$$\begin{gathered}(\forall comp \in components) \\ (\forall mountTask \in mounting(comp)) \\ (\forall putTask \in putting(comp)) \\ end(putTask) \leq moveStart(mountTask)\end{gathered} \tag{4.63}$$

In the case mentioned above we also have take tasks for both components and they must both be performed before we can start mounting anything.

$$\begin{gathered}(\forall comp \in components) \\ (\forall mountTask \in mounting(comp)), \\ (\forall takeTask \in taking(comp)), \\ end(takeTask) \leq moveStart(mountTask)\end{gathered} \tag{4.64}$$

Say we want to put a component away for a while and pick it up again later. Then we need to do that in a tray. This is the only time we put anything in a tray, usually we just take components from them. So we can apply the (4.65) constraint which says that if there is a take and a put on the same tray, then the take has to happen after the put.

$$\begin{gathered}(\forall comp \in components) \\ (\forall putTask \in putting(comp),\ tray(putTask) > 0) \\ (\forall takeTask \in taking(comp),\ tray(putTask) = tray(takeTask)) \\ end(putTask) \leq moveStart(takeTask)\end{gathered} \tag{4.65}$$

When there is a put task and a take task on a fixture where a sub-component of the component being taken is the component being put, the put task has to happen before the take task.

$$\begin{gathered}(\forall f \in fixtures) \\ (\forall putTask \in putting,\ fixture(putTask) = f) \\ (\forall takeTask \in taking,\ fixture(takeTask) = f \wedge \\ componentsUsed(putTask) \subset taskSubComponents(takeTask)) \\ end(putTask) \leq moveStart(takeTask),\end{gathered} \tag{4.66}$$

Since we can do many sub-assemblies on the same fixture, we need to ensure that if a component is put in the fixture, there cannot be a component from another sub-assembly put or mounted there before the sub-assembly is done.

We can observe that the task of doing a sub-assembly begins with a put of a component in a fixture and a take of a component from the same fixture. The taken component will have the put component as a sub-component. With this knowledge we start by extracting all put tasks for a fixture. Then we extract all the corresponding take tasks, i.e. the take tasks for that fixture where the component used in the put task is among the sub-components for the component in the take task. Although, there is the case where we construct a component by first doing some mounting, then we take it up to maybe turn it or fixate it, and then put it back in the fixture for further mounting. In this case we will get two takes matching with the first put. So we need to identify which take task is the first one. We do this by choosing the take task with the least amount of subcomponents.

Now we have a 1:1 matching of take tasks and put tasks. To ensure the time between when a put task occurs and when the take task occurs, we apply a *cumulative* constraint over that time and the limit of the fixture is always 1.

When [and] are used together with : as below, it means they are array generators. What is left of the : is what is put in the array and what is right of it is the condition. A

case here which might be confusing is the last argument to the cumulative constraint. It simply states that it is an array of ones with the same length as *puts*.

$$
\begin{aligned}
&(\forall f \in fixtures) \\
&puts = [put : put \in putting,\ fixture(put) = f], \\
&takes = [min(\{take : take \in taking,\ fixture(take) = f, \\
&\qquad componentsUsed(put) \subset taskCompleteSubComponent(take)\}) : \\
&\qquad put \in puts], \\
&cumulative([moveStart(task) : task \in puts], \\
&\qquad [abs(end(takes(i)) - moveStart(puts(i))) : i \in \{1, \ldots, |puts|\}], \\
&\qquad [1 : i \in \{1, \ldots, |puts|\}], \\
&\qquad 1)
\end{aligned}
\tag{4.67}
$$

The fundamental property of the tasks in a concurrent group is that they need to execute at the same time on different machines. We ensure this with (4.68).

$$
\begin{aligned}
&(\forall group \in \{1, \ldots, nbrConcurrentGroups\}) \\
&\quad (\forall t_1 \in concurrentTasks(group)) \\
&\quad (\forall t_2 \in concurrentTasks(group) \setminus \{t_1\}) \\
&start(t_1) = start(t_2) \wedge \\
&usingMachine(t_1) \neq usingMachine(t_2),
\end{aligned}
\tag{4.68}
$$

A very logical observation we can do is that components cannot be used before they are created. This is enforced in (4.69).

$$
\begin{aligned}
&(\forall t_1 \in tasks,\ componentCreated(t_1) > 0) \\
&(\forall t_2 \in tasks,\ componentCreated(t_1) \in componentUsed(t_2)) \\
&moveStart(t_2) \geq end(t_1)
\end{aligned}
\tag{4.69}
$$

A similar observation as for (4.69) is that we have to perform all tasks with a component before it is part of a sub-assembly. Therefore we can say that all tasks need to have an end time smaller than the start time of the tasks having the tasks component as sub-component.

$$
\begin{aligned}
&(\forall precTask \in tasks) \\
&\quad (\forall t \in tasks,\ precTask \neq t, \\
&\qquad componentUsed(precTask) \cup taskCompleteSubComponents(t) \\
&\qquad \subset taskCompleteSubComponents(t), \\
&\qquad componentsUsed(precTask) \cup taskCompleteSubComponents(t) \neq \emptyset) \\
&end(precTask) \leq moveStart(t),
\end{aligned}
\tag{4.70}
$$

Trays, fixtures and outputs can only be used one at a time. We can rephrase this into saying that tasks using trays cannot overlap, tasks using fixtures cannot overlap, etc. We ensure

this by applying the *cumulative* constraint through (4.71), (4.72) and (4.73).

$$\begin{aligned}
&(\forall f \in fixtures) \\
&fixtureTasks = [t : t \in tasks,\ fixture(t) = f], \\
&cumulative([start(t) : t \in fixtureTasks], \\
&\qquad [duration(t) : t \in fixtureTasks], \\
&\qquad [1 : t \in fixtureTasks], \\
&\qquad 1)
\end{aligned} \tag{4.71}$$

$$\begin{aligned}
&(\forall tr \in trays) \\
&trayTasks = [t : t \in tasks,\ tray(t) = tr], \\
&cumulative([start(t) : t \in trayTasks], \\
&\qquad [duration(t) : t \in trayTasks], \\
&\qquad [1 : t \in trayTasks], \\
&\qquad 1)
\end{aligned} \tag{4.72}$$

$$\begin{aligned}
&(\forall o \in outputs) \\
&outputTasks = [t : t \in tasks,\ output(t) = o], \\
&cumulative([start(t) : t \in outputTasks], \\
&\qquad [duration(t) : t \in outputTasks], \\
&\qquad [1 : t \in outputTasks], \\
&\qquad 1)
\end{aligned} \tag{4.73}$$

4.2.2 Predecessors

All tasks need to have a predecessor that tells the model what task comes directly before it on the same machine. This means that a task can only have one predecessor. It can be seen as the way a machine needs to travel through its tasks in order to complete the assembly, where we have a start task at the start and a goal task at the end. If we were to connect the start and the goal task we wold have a circuit, hence we could view each machine as a circuit. And we could model each machine as a circuit, but then we would need to synchronise all the sub-circuits and ensure that tasks only appeared in one sub-circuit. This would make for quite a few constraints and would make the model more complex. Instead we model all the machines as one circuit and we tie together the goal task of one sub-circuit with the start task of the next for each sub-circuit, to form a large circuit. Then we tie together the goal task of the last sub-circuit with the start task of the first, see (4.75). Lastly we can apply the `circuit` constraint over all *pred* variables.

The attentive reader might have observed that the nodes in the `circuit` constraint have successors and not predecessors. Even if it is the wrong way around, it does not matter if the constraint sees the predecessor variable as a successor or a predecessor, it will form a circuit anyway.

$$\begin{aligned}
&(\forall startTask \in startTasks \setminus \{nbrTasks + 1\}) \\
&pred(startTask) = startTask + nbrMachines - 1
\end{aligned} \tag{4.74}$$

$$pred(nbrTasks + 1) = nbrTasks + nbrMachines \times 2 \tag{4.75}$$

$$circuit(\{pred(t) : \forall t \in tasks\}) \tag{4.76}$$

A fundamental part of a predecessor is that it is the task directly before the task in question, therefore the predecessor has to end before the task starts, or more specific, even before the move to the task.

$$(\forall t \in tasks)\ moveStart(t) \geq end(pred(t)) \tag{4.77}$$

Another fundamental part is that a predecessor is a task performed on the same machine as the task in question. This is enforced by (4.78).

$$\begin{gathered}(\forall t \in tasks \cup goalTasks) \\ usingMachine(t) = usingMachine(pred(t))\end{gathered} \tag{4.78}$$

In a sense, the ordered groups are forced predecessors and hence we enforce that by simply by making a task predecessor to the next task in the array.

$$\begin{gathered}(\forall k \in orderedGroups) \\ (\forall i \in \{1, \ldots, |orderedGroup(k)|-1\}) \\ pred(ordered(k, i + 1)) = ordered(k, i)\end{gathered} \tag{4.79}$$

The following two constraints can seem very specific, but are essential to the scheduling in our model.

In order to properly connect the taking of a component and the mounting of one, we need to ensure that if there is no put task, the take task has to be the predecessor of the mount task.

But we must also ensure the following: The put cannot be on the same fixture or output as the mount. This is because a component that will be mounted in a fixture or output will always first be picked up, then put in either a fixture or output, then mounted with with another component. The component mounted on will also be part of the mounting task. Therefore, if the component is the one being mounted, there will be two tasks; one where the component is taken, and one where the component is mounted. And that is no problem, the constraint applies. But if the component is the one being mounted on, there will be three tasks; one where the component is taken, one where the component is put in a fixture or output, and one where it is mounted on. In this case the take task cannot be the predecessor of the mount task, since the component first must be put in the fixture or output, and then the other take task, where the component being mounted on this component is taken, should be the predecessor of the mount task. Hence we ensure there are no put tasks working on the same fixture or output as the mount task.

The final case we must consider is when there are move tasks involved. There can be a case of a take task of a component, then a couple of move tasks, and lastly a mount task. In this case, the take task cannot be the predecessor of the mount task, and this constraint does not apply. If we applied it, it would contradict the (4.79) constraint. So we need to

ensure that the take task is not in an ordered group either.

$$\begin{aligned}
&(\forall c \in components) \\
&(\forall mountTask \in mounting(c)) \\
&\quad puts = \{p : p \in putting(c), \\
&\qquad (fixture(p) > 0 \wedge fixture(p) = fixture(mountTask)) \\
&\qquad \vee\ (output(p) > 0 \wedge output(p) = output(mountTask))\}, \\
&(\forall takeTask \in taking(c),\ takeTask \notin orderedSet,\ puts = \emptyset) \\
&pred(mountTask) = takeTask
\end{aligned} \tag{4.80}$$

As with (4.80), in order to ensure the relation between when a component is picked up and when it is put that the take task is the predecessor of the put task, i.e. we must first pick up the components before we put it down, and there cannot be anyu other task in between.

Also as with (4.80), there are a few cases to consider. If we want to put a component away for a while in order to pick it up later, there will be a put task and a take task on that component. But in this case the take task cannot come before the put task, since we need to put it down before we can pick it up. So we put in the clause that this constraint does not apply if the put task is on a tray.

We also need to consider the occurrence of move tasks. If there is a move task involved between the take and put, the take task cannot be the predecessor of the put task.

$$\begin{aligned}
&(\forall c \in components,\ moving(c) = \emptyset) \\
&(\forall putTask \in putting(c),\ tray(putTask) = 0) \\
&(\forall takeTask \in taking(c)) \\
&pred(putTask) = takeTask
\end{aligned} \tag{4.81}$$

(4.82) is the constraint that decides if there should be a tool change or not between two tasks. It first calculates what tool state transition will occur between the two tasks, k, by taking the difference between what tool is used in the task and its predecessor. If they use the same tool, no transition needs to occur, i.e. no tool change needed and the difference would be 0. We add 1 to k since the indexes start at 1 in *MiniZinc* and a result of 0 should take constraint to the first index dept-wise in the $timeMatrix3D$.

$$\begin{aligned}
&(\forall t \in tasks) \\
&\quad k = abs(toolUsed(t) - toolUsed(pred(t))) + 1, \\
&moveDuration(t) = timeMatrix3D(pred(t),\ t,\ k)
\end{aligned} \tag{4.82}$$

4.3 Filter

In [Vilím and Barták, 2002b] [Vilím, 2002] [Vilím and Barták, 2002a] Vilím shows that filtering the domains of variables when, as in our case, using sequence dependent setup times can have a great effect on the runtime. Here we present a set of filters in order to minimize the domains of the variables.

4.3.1 Temporal filter

The largest domains in the model are the domains for the variables dealing with time, i.e. the temporal variables. Reducing those has the potential to cut much of the processing time.

We start with defining two variables, $maxMoveDurs(t)$ and $minMoveDurs(t)$. These contain the maximum duration and the minimum duration, respectively, for each task taken from the time matrix.

$$\begin{aligned}&(\forall t \in tasks)\\ &maxMoveDurs(t) = max(\{timeMatrix3D(t, j, k) :\\ &\qquad\qquad \forall j \in tasks,\\ &\qquad\qquad \forall k \in \{1, \ldots, timeMatrixDepth\},\\ &\qquad\qquad j \neq t\})\end{aligned} \tag{4.83}$$

$$\begin{aligned}&(\forall t \in tasks)\\ &minMoveDurs(t) = min(\{timeMatrix3D(t, j, k) :\\ &\qquad\qquad \forall j \in tasks,\\ &\qquad\qquad \forall k \in \{1, \ldots, timeMatrixDepth\},\\ &\qquad\qquad j \neq t\})\end{aligned} \tag{4.84}$$

By using the newly created variables we can -*define yet another two. These define a new maximum for the total time of the assembly, $maxEnd$. It is similar to $maxE$ in equation (4.49), although much more thorough in the filtering. These variables also define a new minimum for the total time of the assembly, it was earlier ju set to 0.

To calculate the maximum end we look at the worst case scenario for the assembly. The worst case would be if all the tasks had to be done one after the other, one at a time, on the same machine and they would take the longest time, according to the time matrix, to move between them. See Figure 4.6. This can simply be defined by summing the durations and maximum move durations for all the tasks.

To calculate the minimum end we look at the best case scenario. The best case scenario is if all the tasks can be evenly scheduled over all machines, taking the shortest, according to the time matrix, time to move between them. See Figure 4.7. We can define this by summing up the durations and minimum move durations for all the task end divide the sum with the number of machines available. If the tasks can be perfectly evenly scheduled across all the machines, the total assembly time will be equal to $minEnd$, if they cannot

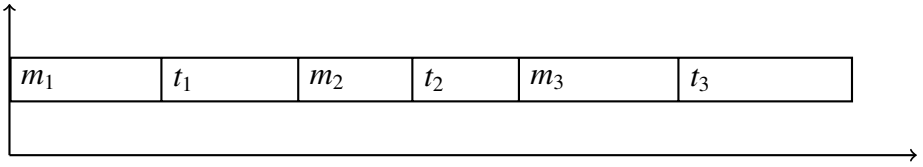

Figure 4.6: The worst case assembly

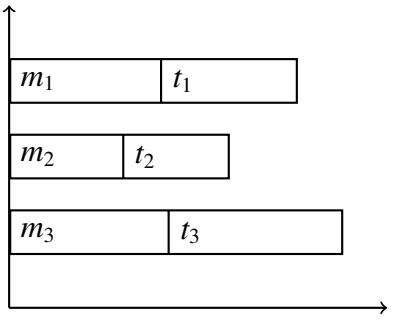

Figure 4.7: The best case assembly

minEnd will always be smaller than the total assembly time.

$$maxEnd = \sum_{\forall t \in tasks} duration(t) + maxMoveDurs(t) \tag{4.85}$$

$$minEnd = \frac{\sum_{\forall t \in tasks} duration(t) + minMoveDurs(t)}{nbrMachines} \tag{4.86}$$

We can now start using *maxEnd* and *minEnd* to filter variables

We set and upper bound for the start of a task by setting it to happen at latest the duration of the task time units before the *maxEnd*, since the task has to have time to execute before the end.

To set a lower bound for the start of a task, we simply reason that the move to the task can start at its earliest at time 0. Therefore, we limit the task to start earliest direct after the minimum move duration to it.

The difference between (4.87) and (4.88) is that the upper limit can be set for all sorts of tasks, even the start and goal tasks, but the lower limit cannot be set for start and goal tasks. This is simply because the start and goal tasks do not have any move times to them since they are source and sink nodes.

$$\begin{aligned} &(\forall t \in allTasks) \\ &start(t) \leq maxEnd - duration(t) \end{aligned} \tag{4.87}$$

$$\begin{aligned} &(\forall t \in Tasks) \\ &start(t) \geq minMoveDurs(t) \end{aligned} \tag{4.88}$$

In order to limit the move start to a task we use the same reasoning as with start. But now we have to account for that there comes a task after the move and a duration of the move itself. So we have to subtract the duration of the task and the duration of the move. Since we do not know the exact length of the move, we have to use the value we know the duration can not be lower than, which is $minMoveDuration$.

$$(\forall t \in tasks)\ moveStart(t) \leq maxEnd - (duration(t) + minMoveDurs(t)) \tag{4.89}$$

We have already calculated the limits for the whole assembly, $maxEnd$ and $minEnd$. Now we just enforce them on the makespan.

$$\begin{aligned} makeSpan \leq maxEnd \wedge \\ makespan \geq minEnd \end{aligned} \tag{4.90}$$

The first thing that has to happen to a component in the assembly is that is has to be picked up. So since the assembly starts out with empty machines the first thing that has to happen, with the exception of a tool change, is a take task. Therefore, we can limit that start of the tasks not being take tasks to happen at earliest after the task with the smallest sum of duration and minimum move duration.

$$\begin{aligned} &(\forall t \in tasks \setminus taking) \\ &moveStart(t) \geq min(\{duration(tt) + minMoveDurs(tt) : \forall tt \in taking\}) \end{aligned} \tag{4.91}$$

The start of a task can be even further limited by analysing the components used by the task and how that relates to what components are created by other tasks.

Lets take task t as an example. We start by getting all tasks that create the components that are used in task t, $prevTasks$. These tasks have to come before task t since the component that they create cannot be used before they are created. If the number of machines is greater than or equal to the number of task preceding task t, then the best scheduling that can be done is to do all tasks in parallel. That means that task t can start at earliest after the one of the preceding tasks taking the longest to complete.

$$\begin{aligned} &(\forall t \in tasks) \\ &prevTasks = \{task : task \in tasks, \\ &\qquad componentCreated(task) \in componentsUsed(t)\}, \\ &nbrMachines \geq |prevTasks|, \\ &\qquad 0 < |prevTasks|, \\ &start(t) \geq max(\{duration(pt) + minMoveDurs(pt) : \forall pt \in prevTasks\}) \end{aligned} \tag{4.92}$$

But if the number of machines is fewer than the number of preceding tasks, the best we can do is divide them as equally as possible over the machines. This is the same reasoning as when we calculated $minEnd$ in equation (4.86).

$$\begin{aligned} &(\forall t \in tasks) \\ &prevTasks(t) = \{task : task \in tasks, \\ &\qquad componentCreated(task) \in componentsUsed(t)\}, \\ &nbrMachines < |prevTasks(t)|, \\ &start(t) \geq \frac{\left(\sum_{\forall pt \in prevTasks(t)} duration(pt) + minMoveDurs(pt)\right)}{nbrMachines} \end{aligned} \tag{4.93}$$

To set the upper limit for the start of tasks we use a little bit different strategy.

We know that if a component c has been mounted on another component, c cannot be used again on its own. Therefore, a task that uses component c has to come before the tasks that uses a component in which c is a part of.

We use the same strategy as in (4.92) and look at the best case scenario where the tasks are performed concurrently on all machines. The difference here from (4.92) is that here we have to look at the maximum end of the assembly and subtract the successor task which takes the longest to perform and the duration of the task in question.

$$\begin{aligned}
&(\forall t \in tasks) \\
&succTasks(t) = \{task : \ task \in tasks, \\
&componentsUsed(t) \subset taskCompleteSubComponent(task)\}, \\
&nbrMachines \geq |succTasks(t)|, \\
&\qquad 0 < |succTasks(t)|, \\
&start(t) \leq maxEnd - max(\{duration(st) + minMoveDurs(st) : \\
&\qquad \forall st \in succTasks(t)\}) - duration(t)
\end{aligned} \tag{4.94}$$

As with (4.93) we look at the worst case scenario.

$$\begin{aligned}
&(\forall t \in tasks) \\
&succTasks = \{task : \ task \in tasks, \\
&componentsUsed(t) \subset taskCompleteSubComponent(task)\}, \\
&nbrMachines \leq |succTasks|, \\
&start(t) \ \leq maxEnd - \frac{(\sum_{\forall st \in succTasks} duration(st) + minMoveDurs(st))}{nbrMachines} \\
&\qquad - duration(t)
\end{aligned} \tag{4.95}$$

4.3.2 Predecessor filter

Since the predecessors are searched by the solver before searching the start variables, reducing their domains has the potential to help reduce the total runtime considerably.

In our model the tools can only pick up one component at a time. This also means that if a task puts down a component or mounts one, there cannot be a mount or put task directly afterwards.

$$(\forall t1, \forall t2 \in taking)\ pred(t1) \neq t2 \tag{4.96}$$

$$(\forall t1, \forall t2 \in putting \cup mounting)\ pred(t1) \neq t2 \tag{4.97}$$

Using a similar reasoning as in (4.94) and (4.95), we can find the tasks that cannot be the predecessor of task t. We look at what tasks uses the components that has the components used in t as sub-components. This means that those components cannot come before task t, and therefore cannot be predecessors of t.

$$\begin{aligned}
&(\forall t \in tasks) \\
&nonPredecessors(t) = \{t_2 : \ t_2 \in tasks, \\
&componentsUsed(t) \subset taskCompleteSubComponents(t_2) \vee \\
&componentsUsed(t) \subset subComponents(componentCreated(t_2))\} \\
&(\forall nonPred \in nonPredecessors(t)) \\
&pred(t) \neq nonPred
\end{aligned} \tag{4.98}$$

As mentioned before, a component has to be picked up first before it can be manipulate in any way and the assembly has to start with a take task. Therefore, we can say that an assembly cannot start with neither a put task nor a mount task.

The same could be said for move tasks, but since they need to be in an ordered group, a constraint like these would make no difference.

$$\begin{aligned}
&(\forall startTask \in startTasks) \\
&(\forall putTask \in putting) \\
&pred(putTask) \neq startTask
\end{aligned} \tag{4.99}$$

$$\begin{aligned}
&(\forall startTask \in startTasks) \\
&(\forall mountTask \in mounting) \\
&pred(mountTask) \neq startTask
\end{aligned} \tag{4.100}$$

The same way we can observe that an assemble needs to start with a take task, we can observe that an assembly cannot end with a take task. There is no component in the assembly that does not end up in the finished assembly, therefore the assembly cannot end with a machine holding a component, since it needs to be on the output in some way.

$$\begin{aligned}
&(\forall goalTask \in goalTasks) \\
&(\forall takeTask \in taking) \\
&pred(goalTask) \neq takeTask
\end{aligned} \tag{4.101}$$

Continuing the reasoning around what tasks can come first and not we can expand with which tasks have to come last. We cannot limit the last task on each arm to be on an output, because it does not necessarily need to be that. Although, among the last tasks in the assembly there needs to be a task on an output. We can easily check that by placing a constraint over the *pred* variables for the goal tasks. This is what (4.102) says.

We can do the same reasoning with what needs to come first in the assembly. As we stated before, a take task has to be the first task in the assembly. And as with the last tasks in the assembly, the first task of a machine does not have to be a take task, but there needs to be at least one take task among the first tasks. We ensure this in (4.103) by placing a

constraint over the *pred* variables for the take tasks.

$$\begin{aligned} counts &= [i : task \in outputTasks,\ i \in \{0,1\}], \\ outputTasks &= [task : task \in tasks,\ output(task) > 0], \\ goalPreds &= [pred(task) : task \in goalTasks], \\ &global\_cardinality(goalPreds,\ outputTasks,\ counts) \\ &\wedge \sum counts > 0 \end{aligned} \tag{4.102}$$

$$\begin{aligned} counts &= [i : \forall task \in startTasks,\ i \in \{0,1\}], \\ takePreds &= [pred(task) : \forall task \in taking], \\ startTasksArray &= [task : task \in startTasks], \\ &global\_cardinality(takePreds,\ startTasks,\ counts) \\ &\wedge \sum counts > 0 \end{aligned} \tag{4.103}$$

4.4 Heuristics

The search was declared to go over the variables through a sequential search [Marriott and Stuckey, 2014] in the following order:

1. The *usingMachine* variables
2. The predecessor variables for take tasks that are not on an output
3. The predecessor variables for put tasks that are not on an output
4. The predecessor variables for mount tasks that are not on an output
5. The predecessor variables for tasks that are on an output
6. The predecessor variables in general
7. The start times of the tasks

The branching heuristics chosen in the searches 1-6 above was `indomain_median`. This means the median value of the domain will be chosen to branch on. This is because there is no numerical relationship between the value the variables can take and the order in which they should come. If constructing the MiniZinc data file using `AssemblyConv`, see appendix B, the values of the variables depends on the order in which entities are declared. For search 7 above, the heuristic is `indomain_min`, which means the smallest value of the domain will be chosen to branch on. This heuristic is chosen since the values of the start variables are the ones we want to minimize, so we search from the bottom up.

Chapter 5

Evaluation

Here we will present the evaluation of the model. Each solver was given a time limit of 4 hours to complete as it seemed long enough to guarantee us a result from all the solver, but short enough to be manageable to test. But as can be seen in section 5.2, this was not always the case. In the case where the solvers could produce a result in the given time frame the solvers where run 10 times and the average time was used.

Originally, the 1.6 version of MiniZinc was used. But during the making of the model version 2.0 was released. According to NICTA, the 2.0 version is a complete rewrite of the MiniZinc-to-FlatZinc comipiler but that the resulting FlatZinc file is compatile with 1.6 solvers, and therefore no changes should be required for the solvers [NICTA, 2014a]. We therefore thought it could be of interest to compare how the 2.0 version performed compare to the 1.6 version. During the test phase of the thesis version 2.0.1 was released that should fix some bugs in 2.0 [NICTA, 2014b]. Therefore, version 2.0.1 is used.

Apart from measuring the time, we also analysed the resulting FlatZinc file to see if we could see any correlation between the running time and the data run. The data extracted was the number of integer and boolean variables, the number of arrays, the number of constraints and the percentage of constraints that were reified. We measure the reifieds since even if we try to avoid direct reifieds in the MiniZinc code, the FlatZinc could still contain reified constraints depending on how it translates the MiniZinc code.

In order to see if the filtering we introduced made any difference, we also measure the different combinations of the filters and the absence of filters. Because of all the combinations of parameters we want to test, there will be 8 test cases for each solver.

What we want to achieve is to get the solver to reach the optimal solution and conclude that it is the optimal solution, that is what we mean when we say *result*. And it should be noted that the result is only analysed in comparison to the handmade assembly, so it will not be tested on an actual physical robot.

5.1 The Setup

The computer used to run the solvers was equipped with an Intel i7 2.8GHz quad core CPU, 8GB DDR3 1333MHz memory and running Mageia 4.

The versions of each of the solvers are presented in the table below.

Solver	Version
G12/FD	1.6.0
JaCoP	4.2
Gecode	4.3.2
or-tools	Rev. 3782
Opturion CPX	1.0.2
Chcoco 3	Solver: 3.2.2 Parser: 3.2.0

Note that for Choco3, the solver and parser has different version numbers. This is because they are not distributed together and therefore have slightly different version numbers. It is possible to specify the number of cores used by the solver for Gecode, or-tools and Choco3. Hence, these three were given access to all 4 cores when running.

The versions of the solvers used are the latest as of December 2014.

5.2 The results

The time for the assembly reached by the solvers as the optimal was 512 time units. This is the time we will use as the optimal when we henceforth talk about which solvers reached the optimal but did not finish the search.

The results presented in the tables 5.1 to 5.6 are the analysis of the FlatZinc files. The tests are grouped into 4 categories of runs; with predecessor filter and temporal filter, with only predecessor filter, with only temporal filter and with no filter at all. In each of these groups there are two groups, one for each version we tests, i.e. MiniZinc 1.6 and 2.0.1.

In the tables presented the filter names has been shortened to *Pred* and *Temp*. *Pred* means the predecessor filter and *Temp* means the temporal filter. We will also use two additional notations in the rime rows; "-" and "!". "-" means that the run did not complete within the given time frame. "!" means that the solver raised an error when reading the FlatZinc file.

The runtime will be presented in two formats, milliseconds and *hours:minutes:seconds*, in the latter the remaining milliseconds are omitted.

G12/FD

Filter	Pred & Temp		Pred		Temp		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	-	-	-	-	-	-	-	-
Time	-	-	-	-	-	-	-	-
# of integer variables	174	293	174	314	154	270	154	291
# of boolean variables	162	97	162	97	142	106	142	106
Total # of variables	336	390	336	411	269	376	369	397
# of arrays	31	53	31	53	29	51	29	51
# of constraints	2437	588	983	588	2153	562	699	562
% reified	7.50%	15.81%	18.61%	15.81%	7.57%	16.90%	23.31%	16.90%

Table 5.1: Results for G12/FD

JaCoP

Filter	Pred & Temp		Pred		Temp		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	658	-	1011156	-	-	-	-	-
Time	0:00:00	-	0:16:51	-	-	-	-	-
# of integer variables	129	225	129	246	129	230	129	251
# of boolean variables	42	27	42	27	42	27	42	27
Total # of variables	171	252	171	273	171	257	171	278
# of arrays	30	53	30	53	28	50	28	50
# of constraints	2195	429	741	429	1951	420	497	420
% reified	2.87%	6.29%	8.50%	6.29%	3.22%	6.42%	12.67%	6.42%

Table 5.2: Results for JaCoP

Opturion CPX - no warm start

Filter	Pred & Dom		Pred		Dom		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	-	!	-	!	-	!	-	!
Time	-	!	-	!	-	!	-	!
# of integer variables	28736	9531	28736	28876	28716	9508	28716	28853
# of boolean variables	57284	18654	57284	57322	57264	18702	57264	57370
Total # of variables	86020	28185	86020	86198	85980	28210	85980	86223
# of arrays	31	4698	31	14370	29	4714	29	14386
# of constraints	102402	33025	100947	100679	102117	33028	100663	100682
% reified	55.98%	56.25%	56.78%	56.85%	56.11%	56.24%	56.92%	56.85%

Table 5.3: Results for Opturion CPX

Gecode

Filter	Pred & Temp		Pred		Temp		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	-	60	-	71761	-	99	-	71186
Time	-	0:00:00	-	0:01:11	-	0:00:00	-	0:01:11
# of integer variables	129	225	129	246	129	230	129	251
# of boolean variables	42	27	42	27	42	27	42	27
Total # of variables	171	252	171	273	171	257	171	278
# of arrays	30	57	30	57	28	50	28	50
# of constraints	2193	425	739	425	1951	420	497	420
% reified	2.87%	6.35%	8.52%	6.35%	3.22%	6.42%	12.67%	6.42%

Table 5.4: Results for Gecode

or-tools

Filter	Pred & Temp		Pred		Temp		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	271	!	380	!	302	!	457	!
Time	0:00:00	!	0:00:00	!	0:00:00	!	0:00:00	!
# of integer variables	129	225	129	246	129	230	129	251
# of boolean variables	42	27	42	27	42	27	42	27
Total # of variables	171	252	171	273	171	257	171	278
# of arrays	30	57	30	57	28	50	28	50
# of constraints	2193	425	739	425	1951	420	497	420
% reified	2.87%	6.35%	8.52%	6.35%	3.22%	6.42%	12.67%	6.42%

Table 5.5: Results for or-tools

Choco3

Filter	Pred & Temp		Pred		Temp		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	-	-	-	-	-	-	-	-
Time	-	-	-	-	-	-	-	-
# of integer variables	129	246	129	246	129	230	129	251
# of boolean variables	42	27	42	27	42	27	42	27
Total # of variables	171	273	171	273	171	257	171	278
# of arrays	30	57	30	57	28	50	28	50
# of constraints	2193	425	739	425	1951	420	497	420
% reified	2.87%	6.35%	8.52%	6.35%	3.22%	6.42%	12.67%	6.42%

Table 5.6: Results for Choco3

If one just does a quick comparison of the numbers in the tables without doing any deeper analysis, one can see that the results from Gecode, or-tools and Choco3, shown in tables 5.4, 5.5 and 5.6, respectively, are identical. This hints that the FlatZinc files for these solvers should be very similar. And indeed, if one compares the FlatZinc files for these three solvers for one of the cases, they are as good as identical. The difference between them is the naming of some constraints and sometimes the placement in the file.

As can be seen in the times or the runs, not many of the runs completed in time. Only 9 runs completed by 3 solvers. Interesting to note is the very varying runtimes. It ranges from 60 milliseconds to 16 minutes. Almost no solver could solve the problem using

MiniZinc 2.0.1, two of them threw errors reading the FlatZinc file, despite its claimed backwards compatibility, and most others took too long time solving it. But, interestingly Gecode manages to solve the problem using 2.0.1, even faster than using 1.6, and even being the fastest to solve the problem of all the runs in just 60 milliseconds.

By our definition of a result, these are the runs that found the optimal solution and ended the search by concluding that the optimal solution was found. This means that there are other runs that still produced results, sometimes even the optimal one, but did not end the search in time and therefore not qualify as a result.

The G12/FD runs using predecessor and temporal filter with version 2.0.1, using predecessor filter with version 1.6 and using predecessor filter with version 2.0.1 all find solutions, although none of them are the optimal.

The JaCoP runs using predecessor and temporal filter with version 2.0.1 and using predecessor filter with version 2.0.1 both finds all the solutions, including the optimal solution, immediately. This is also the case for JaCoP using predecessor filter with version 1.6.

The Gecode run using predecessor filter with version 2.0.1 finds three solution immediately of which one is the optimal. Interestingly, the run using no filter also with version 2.0.1 finds only one solution and that one is the optimal in roughly the same time.

The Choco3 runs using the predecessor and temporal filters and the ones using predecessor filter, independent of version, all finds two solutions immediately.

We should mention that some of the tests ran for much longer than the time limit set at 4 hours. And although not all produced results even when left for more than 4 hours, some managed to produce results. One that could be mentioned is Opturion CPX that in some cases took around 6 hours to complete.

We should also mention that Opturion CPX has the option to use the search strategy "warm start". This was tested but, as with regular complete search, did not produce a result within the time frame and thus we have not specifically included the results here.

We will present some more tables that show the change in the values. The first 4 columns are grouped by the version of MiniZinc and in each of those groups we show the change in the values if each of the filters are added. The last column shows the change in values if going from using the 1.6 version to using the 2.0.1 version. The values presented are averages. For example, the change for when adding the temporal filter for version 1.6 is the average of the changes from not using any filters to using the temporal filter and from using the predecessor filter to using the temporal and predecessor filters. The average for the last column is just the average of the changes for all cases when switching from 1.6 to 2.0.1.

As mentioned before, the results for Gecode, or-tools and Choco3 are identical, and thus we grouped them together into one table below.

G12/FD - Change

	1.6		2.0.1		
	+Pred	+Temp	+Pred	+Temp	1.6 - 2.0.1
# of integer variables	12.99%	0.00%	8.21%	-6.95%	78.28%
# of boolean variables	14.08%	0.00%	-8.49%	0.00%	-32.74%
Total # of variables	24.91%	0.00%	3.62%	-5.20%	31.44%
# of arrays	6.90%	0.00%	3.92%	0.00%	73.41%
# of constraints	26.91%	177.96%	4.63%	0.00%	-52.39%
% reified	-10.54%	-63.61%	-6.45%	0.00%	48.88%

Table 5.7: Change in values for G12/FD

JaCoP - Change

	1.6		2.0.1		
	+Pred	+Temp	+Pred	+Temp	1.6 - 2.0.1
# of integer variables	0.00%	0.00%	-2.08%	-8.45%	84.50%
# of boolean variables	0.00%	0.00%	0.00%	0.00%	-35.71%
Total # of variables	0.00%	0.00%	-1.87%	-7.62%	54.97%
# of arrays	7.14%	0.00%	6.00%	0.00%	77.62%
# of constraints	30.80%	244.39%	2.14%	0.00%	-54.13%
% reified	-21.89%	-70.41%	-2.02%	0.00%	35.80%

Table 5.8: Change in values for JaCoP

Opturion CPX - Change

	1.6		2.0.1		
	+Pred	+Temp	+Pred	+Temp	1.6 - 2.0.1
# of integer variables	0.07%	0.00%	0.16%	-67.02%	-33.19%
# of boolean variables	0.03%	0.00%	-0.17%	-67.43%	-33.63%
Total # of variables	0.05%	0.00%	-0.06%	-67.29%	-33.48%
# of arrays	6.90%	0.00%	-0.23%	-67.27%	31742.94%
# of constraints	0.28%	1.44%	-0.01%	-67.20%	-33.91%
% reified	-0.24%	-1.42%	0.01%	-1.06%	0.18%

Table 5.9: Change in values for Opturion CPX

Gecode or-tools Choco - Change

	1.6		2.0.1		
	+Pred	+Temp	+Pred	+Temp	1.6 - 2.0.1
# of integer variables	0.00%	0.00%	-2.08%	-8.45%	84.50%
# of boolean variables	0.00%	0.00%	0.00%	0.00%	-35.71%
Total # of variables	0.00%	0.00%	-1.87%	-7.62%	54.97%
# of arrays	7.14%	0.00%	14.00%	0.00%	84.29%
# of constraints	30.55%	244.65%	1.19%	0.00%	-54.27%
% reified	-21.81%	-70.45%	-1.09%	0.00%	36.46%

Table 5.10: Change in values for Gecode, or-tools and Choco3

When we look at the change in values for the filters we see that the predecessor filter for G12/FD using version 1.6 increases the number of variables, both integer and boolean, as opposed to the other solvers which does not change the number of variables at all. Except for Opturion CPX, which adds a very small amount of variables. When using G12/FD with 2.0.1 the predecessor filter increases the number of integer variables as well, as opposed to the other solvers decreasing the number of integer variables. Again except Optuion CPX who increases the number a very small amount.

Both the predecessor filter and temporal filter increases the number of constraints significantly, except for Opturion CPX where the change is not that great. And the largest increase is for the temporal constraint, which lies around 200% for all the solvers. This can be compared to using 2.0.1, where the change is 0%. The same can be observed for the predecessor filter, but here we still have a small change in the number of constraints, but not as large as with 1.6. The temporal filter also decreases the number of variables in 2.0.1, as opposed to 1.6 where it did not change that number at all. When using the temporal filter with Opturion CPX with 2.0.1, the filter helps to reduce all the values, and quite dramatically so, all except reified lies around 67%.

Compared to 1.6, 2.0.1 seems to increase the number of integer variables by a lot, around 50%. And once again except for Opturion CPX where it decreases by around 30%. The same goes for boolean variables, but in the other direction. For all solvers, including Opturion CPX, the number of boolean variables decreases by around 33%. The number of arrays also increases for all the solvers, especially for Opturion CPX, where the increase is a staggering almost 32000%. 2.0.1 also decreases the number of constraints by about 48% on average for all the solvers. But despite removing constraints, the amount of reified constraints increases by about 40%, except for Opturion CPX where the change is only 0.2%.

Unfortunately no solver was able to run both the 1.6 version and the 2.0.1 version. Therefore we cannot make a good comparison and make a definitive statement about which version is the most effective. Although, the 2.0.1 version shows promising results for Gecode.

In Figures 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6 we plot the values taken from tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6 versus time. The time is on the y-axis and is logarithmic.

As can be seen in Figures 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6, we cannot conclude any relation between the time for the solution and the number of either integer, booleans, arrays, constraints or the percentage of reified constraints. This can also be seen if we study the results for Gecode, or-tools and Choco3. Since they get the same values for the FlatZinc files but completely different times, we can conclude that our data does not show any relation between time and those values.

We should also mention that we tried to run all the solvers using a very small assembly, basically one with a pick up and a put down of a component. All solvers solved it in essentially no time, except Opturion CPX, which could not solve it at all, even after 50 minutes.

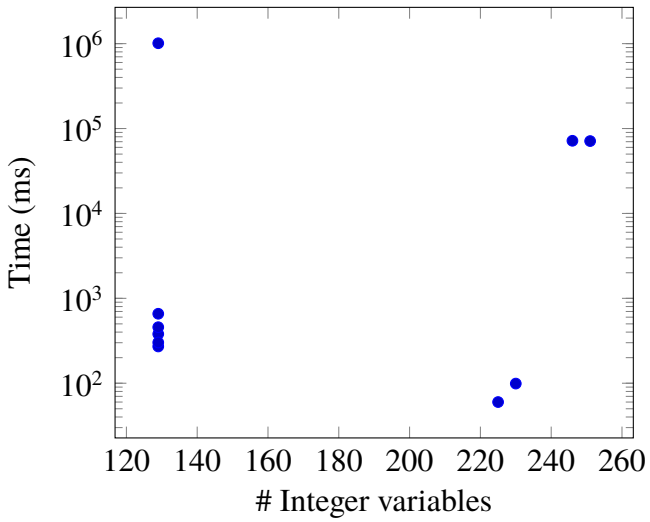

Figure 5.1: # integer variables vs time

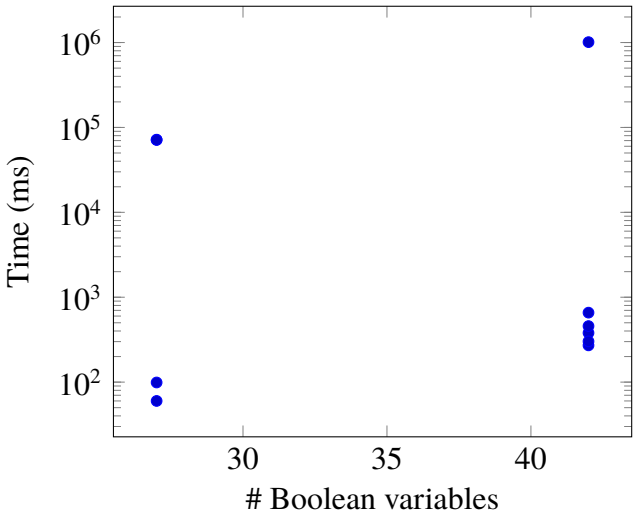

Figure 5.2: # boolean variables vs time

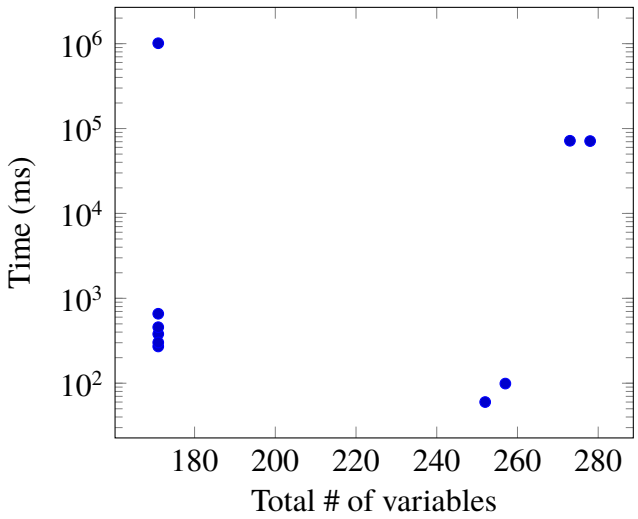

Figure 5.3: Total # of variables vs time

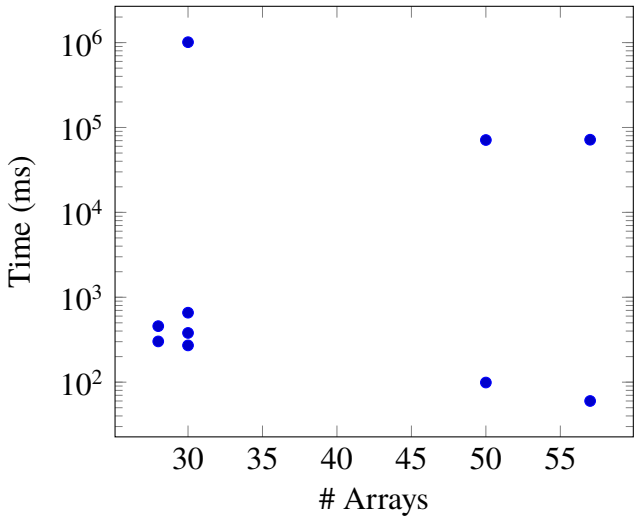

Figure 5.4: # arrays vs time

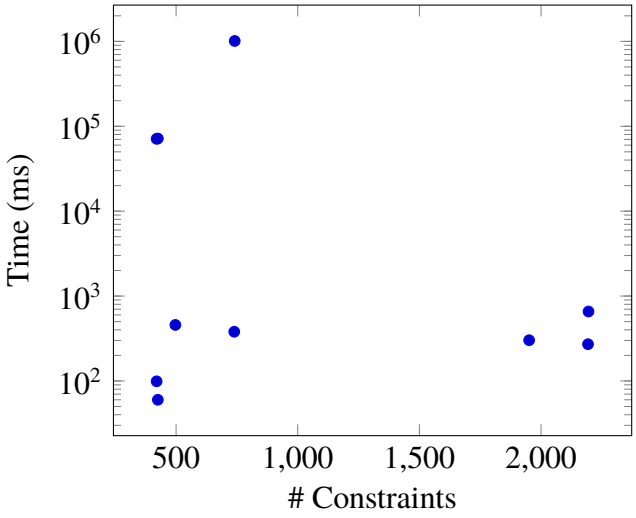

Figure 5.5: # constraints vs time

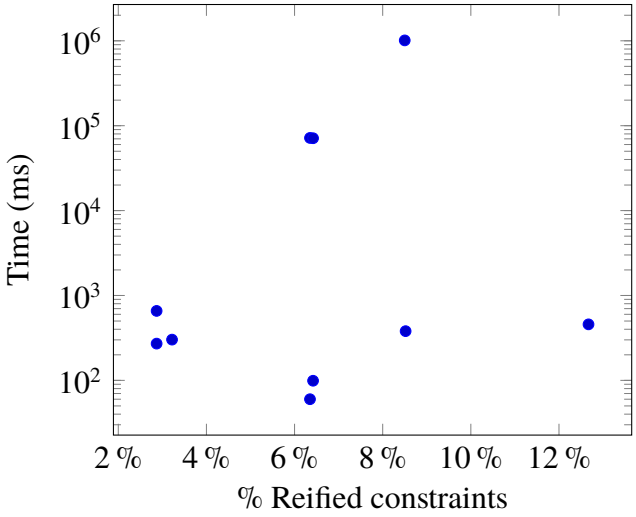

Figure 5.6: % reified vs time

Chapter 6
Discussion

6.1 Model

As we mentioned earlier, the design of this model has taken inspiration from the model in [Ejenstam, 2014]. The development started out with the goal to solve a similar problem but in a different way. Ejenstam solves it as a Vehicle Routing Problem (VRP), but we wanted to try and solve it as a Job Shop Problem. We read through his thesis and took some inspiration from the way of tagging or categorising tasks, such as using tray, using fixture, etc., while trying to not be too much influenced by his solution. But as the development moved on we found our solution tending more and more towards Ejenstam's. For example, we wanted our model to be as pure of a job shop problem as possible, but then we needed to know what task came before a certain task and that warranted for predecessors and creating circuits. Although, there are some differences in our case studies that affect our models in different ways.

In Ejenstam's model each sub-assembly has its own dedicated fixture. Or rather, each sub-assembly that does not require any lifting from the fixture. Where the limit goes is not clear. Take for example the case of putting *component1* in a fixture, mounting *component2* on *component1*, pick upp sub-assembly *component1-component2*, do some operation, maybe turning it over, putting it back in a fixture, and mount *component3* on sub-assembly *component1-component2*. Do we need two fixtures or does one suffice? In our model we can cope with only one fixture even if we have more than one sub-assembly, as in our case study. But this also means we need to ensure that no part is put in each fixture before the fixture has been emptied. We do this with constraint (4.67) where we identify which put and take is associated with each sub-assembly on the fixtures. One does not need to do that in Ejenstams model since there are as many fixtures as sub-assemblies and therefore automatically only one put and one take task for a single fixture. It makes Ejenstams model simpler in this aspect, since the problem of assigning which sub-assembly gets which fixture and when it gets it is solved before the solver tries to solve the problem.

In our model it is part of the problem for the solver to solve.

In both Ejenstam's model and our model we try to perform collision avoidance. By that we mean that we avoid collisions in, for example, fixture, it does not mean that we avoid collisions altogether. We do not assure that arms do not collide in mid air. Although, it is somewhat accounted for when we say that arms cannot reach certain tasks. Since the robot has two arms positioned in such a way that we could divide the work area in two parts where one arm is responsible for one part and the other arm for the other part. In this way we could avoid them colliding when moving about. Although, this might not be the best strategy when aiming for the smallest makespan. The feature of being able to assign tasks that certain arms cannot reach was developed last in our model and right before implementing that feature we got the makespan of 504 time units. Compared to the result we got afterwards, 512, we would get a better makespan if the feature was not used. This is hinting that it could be beneficial to not constrain the tasks the arms cannot reach too much.

Another thing that needs to be considered to avoid the machines from colliding is where they go when a task is completed. Because there can be a wait between when a task ends and the next task starts the machines can be stationary at some point in space. This is nothing our model takes into consideration, instead our model assumes that the machine performing the task does not remain at the point where the task was performed after the task ends, as can be seen in constraints such as (4.71). Instead it is something the one who creates the data to be scheduled needs to think about. The one creating the data might create tasks in such a way that they actually represent the task and a move away from the task to a spot where the machine will be able to wait for the next task. This will not eliminate the problem of the other machine colliding with the machine when it waits in the new spot. But it will eliminate the problem of machines colliding at, for example, a fixture when one arm has put a component in the fixture and the other machine wants to mount another component.

Instead of being able to change tool as in our model, Ejenstam's model gives the possibility to have many tools on the same arm at the same time. Our ambition was to be able to solve the data for Ejenstam's case study using our model. And thanks to Johan Wessen at ABB we were allowed access to the data used. Unfortunately, this part of the functionality of Ejenstam's problem is what put a spanner in the works for that ambition. Because if we wanted to solve Ejenstam's data we would need to be able to have hands with many tools. As it is now, we can have one tool mounted on a machine at a time with the possibility to switch to another tool. If we have a case were we have *tool1* mounted on the machine but we need *tool2* there is only one thing we can do, and that is to change the tool. If we incorporated the possibility to have many tools on one hand and have the case of having three hands were *hand1* having one *tool1*, *hand2* and two *tool2* and *hand3* having one *tool2* and one *tool3*, having *hand1* mounted on the machine and needing *tool2*. This gives the possibility to either switch to *hand2* or *hand3* and thereby increasing the complexity of the problem because a decision on what hand to mount on the machine at this time will affect the need to change hands in the future in a greater way compared to changing single tools as we do in our model.

By only allowing one tool mounted at a time in our model we can filter the precedences of certain types of tasks. For example in our model, because of only having one tool at a time, we can only pick up one component at a time and we have to put it down or mount

it before picking up another tool. It is in contrast to Ejenstam's model where we can pick up several components after one another, the amount depending on the hand mounted on the machine. Because of this, we can in our model filter taking tasks so they cannot be predecessors to one another and thereby reducing the search space. We do this filtering with constraints (4.96) and (4.97).

In a way our model is more complex than Ejenstam's because we have the options to customise mountings and movements, such as mounting in mid air using only the two machines and no fixture. Or use fixtures to store components or sub-assemblies. But Ejenstam's search space is much larger, mainly because of two reasons. One, they want to not only find an optimal solution using one fixed setup of trays, but rather how can we place the trays to get the most optimal makespan. Which is like our case study, but with another layer on top that needs to be solved. And two, because they can pick up several components after one another before mounting or putting them down. These two factors makes Ejenstam's search space much larger than ours and thus it does warrant for using customised search strategies as the ones tested in his thesis. Since our case study is relatively small it might not necessarily need some customised searches. But as our results show, this seem to vary depending on what solver is used. And also, if the assembly would be larger, i.e. more tasks to perform, such as if we would like to schedule several cycles of the assembly, there might come a need for using customised searches such as the ones in Ejenstam's thesis.

As opposed to our case study, Ejenstam made a time study where he measured the times both for the time of the moves between tasks and for the time of the tasks themselves. The move times where measured using RobotStudio [ABB, 2015]. Whether RobotStudio was used for measuring the task times is unclear. While we on the other hand estimated our times using a video. Ejenstam used the exact measured times from the time study and by looking at the data from the study one can see that the times vary somewhat on the same task depending on what machine performed the task. This might seem a bit odd, but it could be due to an unknown factor that might have to be considered when a task is performed. It could also be a coincidence when the measurements were made. If this is the case, measurements that is prone to errors that will make the times between arms on the same task different from one another have the potential to introduce unfair advantages to one of the machines. In an ideal environment a task should take the same amount of time to perform independently of the machine performing it. This is how we look at it, we use the same times for both arms. Although, as said before, we use estimated times, so our times are even more error prone than Ejenstam's. But our times does not have the potential to give unfair advantages to one machine or another. But it is a big source of error when it comes to comparing the time from the solver and the real assembly in the video. Since we first estimated the times and then modified them a bit, we could not simply compare the results from the solvers with the time in the video. Rather, we had to analyse the video again and try to append the estimated times on the tasks in the video and when they occurred and get a total time of the assembly that way. In future works it would be beneficial if a time study was performed for the use case.

In our model we represent tasks that needs multiple machines by creating as many tasks as machines needed and adding the demand that the tasks needs to execute simultaneously. Another possible way to model it would be to have one task with many machines. But

that would mean that tasks could have many predecessors since there are as many tasks previous to this tasks as there are machines performing this task that need to be linked with this task. And it would also require all the precedence constraints to be remodeled and the `circuit` constraint would not work for this model anymore since tasks would now be able to appear in multiple places on the circuit. So the approach of using multiple tasks to model a task using multiple machines might need additional information from the input, but in all is easier to model with constraints.

In our input to the model we provide the time in a matrix form with all the possible needed times already calculated. This seems to give the solver a good performance since it does not need to calculate it on the fly over and over again. We do a similar thing with the time matrix file we use to generate the input file to the solvers, see B.1.2. Since there are many tasks performed in the same physical location, as in our case study where all fixture related tasks are performed in the same fixture, a change in the location of the fixture would make it a hassle to change all the affected times. We could solve it in the XML file, see B.1.1, by associating all the locations used with keys and then each task would be related to one of those keys. Then the translation tool, see B.2, could look up the position for the tasks and calculate the time the time for the movement between them.

There is a filter applied to the $moveDuration$ for each task that limits its domains, the (A.1) constraint. This filter might not be necessary since the values for $moveDuration$ are assigned using the $3DTimeMatrix$ and therefore not searched like a regular domain. So limiting the domain to just the values it can be assigned from the matrix probably does not make any difference.

When running a smaller test with an assembly that did not have neither concurrent tasks nor ordered tasks it was discovered that assemblies such as that one could not be run using version 2.0.1. The error occurs when translating the MiniZinc code into FlatZinc, so it seems that 2.0.1 handles empty arrays in a different way and does accept them.

6.2 Results

The time from the handmade assembly was 516 time units and the one done by the solvers was 512 time units. The ordering of the tasks is identical in the two solutions. By studying the handmade assembly and the one from the solvers, one can see that the difference lies in that in the handmade schedule the tasks sometimes wait for the another task to finish before starting, or even starting to move to the task. In the assembly created by the solvers, tasks sometimes had to do the same and wait for another task to finish. But that depended on what relationship there was between the tasks, and sometimes the task could proceed earlier than in the handmade assembly and thereby shorten the makespan.

The intention when we started testing was to generate the FlatZinc files and the let the solvers run on them in order to remove the time it takes to translate the MiniZinc file from the measurements. However, one of the solvers took longer time to solve the problem if it just ran the FlatZinc file directly instead of running the default command that translates and runs the solver directly. It seems odd, but was something we just had to deal with. Because of this, all the tests were ran with the default commands that first translate the MiniZinc file and then run the solver. This affects the times, although differently depending on if the solve time relates to the parse and translate time of the MiniZinc file. Although, when one

runs the solvers to solve a problem, one will probably use the command to first translate the MiniZinc file and the run the solver, so doing the measurement for that case comes closer to the real use case than to not incorporate the compilation.

As mentioned before. Opturion CPX can in some instances be a faster solver than a regular FD solver. However, as can be seen in the table 5.3 this does not seem to be the case for this set of data and model.

As can be seen in table 5.3 Opturion CPX produces files with much more variables and constraints than the other solvers. This is probably because it is a solver that combines SAT and FD to solve the problem and thus it would probably need to translate many relations into pure boolean relationships, making the files larger.

The two fastest solvers, Gecode and or-tools, were amongst the solvers were the number of cores could be specified. And as said, they were given access to all 4 cores when running, giving them a slight upper hand compared to the other solvers. But, as we can see from the results, one of the solvers which could not solve the model at all was also among the solvers that got access to all the cores, namely Choco3. Therefore, there seems to be a small connection between the number of cores and the time it takes to solve the problem, but it is not necessarily true for all solvers.

Even if many of the solvers did not produce a result in the form of concluding that they had reached the optimal solution, many did reach solutions and sometimes the optimal solution as well. As mentioned before, the Job Shop Problem is NP-complete, which means that even if a solver reached the optimal solution and ended the search within time for the assembly in this case study, it does not mean it will for another assembly. So even the solvers that did not end the search within time could still be used to generate solutions to this problem if we set a time for when we want a solution and take the best solution the solver has generated in that time.

By looking at the results of JaCoP, Gecode and or-tools we can see that the filters do make a difference in the runtime. Comparing using both groups of filters versus using no filter it shows that the filters can have quite a large impact on the runtime. However, concluding which of the filter groups works the best is hard. If we look at Gecode and or-tools, they seem to show that the temporal filter makes a little more improvement than the predecessor filter. Looking at only the result from Gecode, the temporal filter makes a much greater improvement than the predecessor filter. The predecessor filter is even a little bit slower here. However, if we look at the result for JaCoP, it is the other way around. Here the predecessor filter makes a much greater improvement, it cuts down the time from 4+ hours to just 16 minutes.

As mentioned, when using the 1.6 version the filters increase the number of constraints quite a bit, but the number of reified constraints are decreased at the same time by quite a lot as well. This seems to indicate that the filters do not introduce more reified constraints, or at least not a significant amount, which is good.

The goal of a filter is to decrease the domains of the variables, so the optimal solution would be for the solver to just remove the unnecessary values from the domain without adding new constraints. And we can see when using version 1.6 that both the temporal filter and the predecessor filter introduce new constraints. We briefly analysed how JaCoP handled these constraints and we saw that it used those constraints to prune the domains and removed the constraints. This means that in 1.6 MiniZinc hands over the pruning to the solvers, but there is no guarantee that the solvers will prune the domains. Looking at

the results for when applying the filters using the 2.0.1 version we can see that the number of constraints is not affected, or in the predecessor case affected very little. This seems to indicate that the translator does the pruning itself instead of handing it over to the solvers.

Chapter 7
Conclusions

Through this thesis we can conclude that the presented model representing the problem as a Job Shop Problem works well and can produce a result that is as good as the handmade solution.

We can also conclude that when it comes to choosing solvers there can be a very large difference in the performance. It can range from not being able to solve a problem in 4 hours to solve it in under 100 milliseconds. The best performance was achieved by Gecode in 60 milliseconds using the 2.0.1 version of MiniZinc. This is promising result for the new version. Interestingly the 1.6 version of the same solver could not solve the problem in 4 hours. But although the 2.0.1 version performed remarkably well for Gecode, two of the solvers were not able to read the FlatZinc file produced by that version, more specifically Opturion CPX and or-tools.

We tested applying filters that reduced the domains of temporal variables and predecessor variables. Our tests showed that these filters can have an positive impact on the runtime result, and thereby confirming the results of Vilím in [Vilím and Barták, 2002b] [Vilím, 2002] and [Vilím and Barták, 2002a].

The analysis of the resulting FlatZinc files showed that there was no relation between the resulting FlatZinc file and the runtime of the solver. Moreover, we found no relation between the percentage of reified constraints and the runtime of the solver.

7.1 Further work

Since the result from the solver was not evaluated on physical robot it could be a suitable continuation to try out the result on an actual robot. It could show problems not seen in the result itself, such as collisions etc.

In this thesis we present a set of filter constraints to reduce the domains of the variables. The tests performed compared the result of using the filters versus not using the filters. It

is not necessarily the case that all filters are needed, and all filters do probably not perform equally well. Therefore it might be of interest to test the each individual filter for it self and compare the filters against each other to see which performs the best and take that experience to come up with further filters.

Since MiniZinc does not have support for customized searches, customized searches are not doable in regular MiniZinc. But in [Björdal, 2014] they present a constraint-based local search solver that performs local search in order to improve the runtime. Although Ejenstam concludes that local search performs poorly on the data in that thesis, it might be worth looking into. In [Yuan and Xu, 2013] they conclude that Large neighborhood Search in combination with Hybrid Harmony Search performed very well for instances of Flexible Job Shop Problem. To incorporate such a search in the solver of [Björdal, 2014] might be beneficial to models such as this.

As mentioned, our ambition for this thesis was originally to be able to run Ejenstam's data on our model, but the extended search space prevented us from that. The testing when developing the model, and thereby the tests for the extension to be able to run Ejenstam's data, was performed only on JaCoP. But as we can see from the results, there are solvers that could perform better. Hence, there might be a possibility to solve Ejenstam's data using those solvers. If an incorporation of local search, as mentioned above, can be done, it would probably further the possibility to adapt the model to the data.

As mentioned before, our model assumes that there are as many sets of all the tools defined as there are machines, so that each machine has its own set of tools. This simulates an ideal world where we have all the resources we want, when in fact we might be limited to a few sets of tools. To be able to do more realistic schedulings that considers the amount of tools available, there would need to be a considerable number of constraints added to the model and it might be worth looking into in future work.

Bibliography

[ABB, 2014] ABB (2014). YuMi. http://new.abb.com/products/robotics/yumi. Accessed 2014-10-29.

[ABB, 2015] ABB (2015). RobotStudio. http://new.abb.com/products/robotics/robotstudio. Accessed 2015-02-06.

[Becket et al., 2008] Becket, R., Brand, S., Brown, M., Duck, G., Feydy, T., Fischer, J., Huang, J., Marriott, K., Nethercote, N., Puchinger, J., Rafeh, R., Stuckey, P., and Wallace, M. (2008). The many roads leading to rome: Solving zinc models by various solvers. Technical report, ModRef'08: 7th International Workshop on Constraint Modelling and Reformulation, Sydney Australia.

[Beldiceanu et al., 2015] Beldiceanu, N., Carlsson, M., and Rampon, J.-X. (2015). Global constraint catalog, 2nd edition (revision a).

[Björdal, 2014] Björdal, G. (2014). The first constraint-based local search backend for minizinc.

[Charles Prud'homme, 2014] Charles Prud'homme, Jean-Guillaume Fages, X. L. (2014). *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.

[Drobouchevitch et al., 2006] Drobouchevitch, I. G., Sethi, S. P., and Sriskandarajah, C. (2006). Scheduling dual gripper robotic cell: One-unit cycles. *European Journal of Operational Research*, 171(2):598–631.

[Ejenstam, 2014] Ejenstam, J. (2014). Implementing a time optimal task sequence for robot assembly using constraint programming. Master thesis, Uppsala University.

[Fages et al., 2014] Fages, J.-G., Chabert, G., and Prud'homme, C. (2014). Combining finite and continuous solvers. *ArXiv e-prints*.

[Garey et al., 1976] Garey, M. R., Johnson, D. S., and Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129.

[Jefferson et al., 2010] Jefferson, C., Moore, N. C. A., Nightingale, P., and Petrie, K. E. (2010). Implementing logical connectives in constraint programming. *Artif. Intell.*, 174(16-17):1407–1429.

[Marriott and Stuckey, 1998] Marriott, K. and Stuckey, P. J. (1998). *Programming with constraints : an introduction*. Cambridge, Mass. : MIT Press, cop.

[Marriott and Stuckey, 2014] Marriott, K. and Stuckey, P. J. (2014). A MiniZinc Tutorial. `http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf`. Accessed: 2015-02-12.

[Nethercote et al., 2007] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. (2007). Minizinc: Towards a standard cp modelling language. In *In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer.

[NICTA, 2014a] NICTA (2014a). MiniZinc 2. `http://www.minizinc.org/2.0/index.html`. Accessed: 2015-01-28.

[NICTA, 2014b] NICTA (2014b). MiniZinc 2 Change Log. `http://www.minizinc.org/2.0/changes.html`. Accessed: 2015-01-28.

[NICTA, 2014c] NICTA (2014c). MiniZinc Challenge 2014 Results. `http://www.minizinc.org/challenge2014/results2014.html`. Accessed: 2015-01-25.

[NICTA, 2014d] NICTA (2014d). MiniZinc Global Constraints. `http://www.minizinc.org/2.0/doc-lib/doc-globals.html`. Accessed: 2015-01-19.

[Opturion Pty Ltd, 2013] Opturion Pty Ltd (2013). *Opturion CPX User's Guide: Version 1.0.2*. Opturion Pty Ltd.

[Opturion Pty Ltd, 2014a] Opturion Pty Ltd (2014a). About us. `http://www.opturion.com/about_us.html`. Accessed: 2015-01-26.

[Opturion Pty Ltd, 2014b] Opturion Pty Ltd (2014b). CPX Discrete Optimiser. `http://www.opturion.com/cpx.html`. Accessed: 2015-01-26.

[Schulte et al., 2014] Schulte, C., Tack, G., and Lagerkvist, M. Z. (2014). *Modeling and Programming with Gecode*. Corresponds to Gecode 4.3.2.

[Stolt et al., 2013] Stolt, A., Linderoth, M., Robertsson, A., and Johansson, R. (2013). Robotic assembly of emergency stop buttons. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, Japan.

[Szymanek, 2010a] Szymanek, R. (2010a). About JaCoP. `http://jacop.osolpro.com/index.php?option=com_content&view=article&id=46:about-jacop-article&catid=34:general&Itemid=28`. Accessed: 2015-01-26.

[Szymanek, 2010b] Szymanek, R. (2010b). JaCoP Overview. http://jacop.osolpro.com/index.php?option=com_content&view=article&id=19&Itemid=27. Accessed: 2015-01-26.

[The G12 Team, 2014] The G12 Team (2014). *Specification of Zinc and MiniZinc v.1.0*. NICTA, Victoria Research Lab, Melbourne, Australia.

[Thörnblad et al., 2013] Thörnblad, K., Strömberg, A.-B., Patriksson, M., and Almgren, T. (2013). An efficient algorithm for solving the flexible job shop scheduling problem. In *25th NOFOMA conference proceedings, June 3-5 2013, Göteborg, Sweden*, page 15.

[Tsang, 1993] Tsang, E. (1993). *Foundations of constraint satisfaction*. Academic Press.

[van Hoeve and Katriel, 2006] van Hoeve, W.-J. and Katriel, I. (2006). Chapter 6 - global constraints. In Francesca Rossi, P. v. B. and Walsh, T., editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169 – 208. Elsevier.

[van Omme et al., 2014] van Omme, N., Perron, L., and Furnon, V. (2014). or-tools user's manual. Technical report, Google.

[Vilím, 2002] Vilím, P. (2002). Batch processing with sequence dependent setup times: New results. In *Proceedings of the 4th Workshop of Constraint Programming for Decision and Control, CPDC'02*, Gliwice, Poland.

[Vilím and Barták, 2002a] Vilím, P. and Barták, R. (2002a). A filtering algorithm sequence composition for batch processing with sequence dependent setup times. Technical Report KTIML 2002/1, Charles University, Faculty of Mathematics and Physics, KTIML MFF UK, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic.

[Vilím and Barták, 2002b] Vilím, P. and Barták, R. (2002b). Filtering algorithms for batch processing with sequence dependent setup times. In Ghallab, M., Hertzberg, J., and Traverso, P., editors, *Proceedings of the 6th International Conference on AI Planning and Scheduling, AIPS'02*, pages 312–321. The AAAI Press.

[Yuan and Xu, 2013] Yuan, Y. and Xu, H. (2013). An integrated search heuristic for large-scale flexible job shop scheduling problems. *Computers & Operations Research*, 40(12):2864–2877.

Appendices

Appendix A
Extended Model

This appendix contains constraints that are included in the model, but are not as essential for the assembly as the ones in chapter 4.

A.1 Temporal filter

$$\begin{aligned}&(\forall t \in tasks)\\&(\forall i \in \{0,\ldots,maxMoveDurs(t)\} \setminus \{timeMatrix3D(task, j, k) :\\&\qquad \forall j \in tasks,\ \forall k \in \{1,\ldots,timeMatrixDepth\},\ t \neq j\})\\&moveDuration(t) \neq i\end{aligned} \tag{A.1}$$

We know that the value for move duration will be one of the values in the time matrix, hence we can restrict the duration to only those values. We do that by coming up with the values that the duration cannot assume, and limits the duration to not have those values in its domain.

A.2 Predecessor filter

$$alldifferent(\{pred(t) : \forall t \in tasks\}) \tag{A.2}$$

The `circuit` constraint already sees to it that the predecessors of the tasks forms a circuit. This means that all the predecessors will take on different values. However, we apply this `alldifferent` constraint in order to help the `circuit` make the predecessor variables take on different values.

$$\begin{aligned}(\forall comp \in components)\\ (\forall mountTask \in mounting(comp))\\ (\forall takeTask \in taking(comp))\\ pred(takeTask) \neq mountTask\end{aligned} \tag{A.3}$$

For all the tasks that operate on the same component we can restrict so the mount task of the component cannot be the predecessor of the take task.

$$\begin{aligned}(\forall comp \in components)\\ (\forall putTask \in puttingcomp,\ tray(putTask) > 0)\\ (\forall takeTask \in taking(comp),\ tray(putTask) = tray(takeTask))\\ pred(putTask) \neq takeTask\end{aligned} \tag{A.4}$$

We can also restrict the predecessor of a put task for a component to not be the take task for that component, if the two tasks are performed on the same tray. This can help in a situation when we want the assembly to put down a part for a moment and pick it up later, because if the part does not come in the tray from the beginning, i.e. it is not the component tray, we first need to put it in the tray before we are able to take it.

$$\begin{aligned}(\forall f \in fixtures)\\ (\forall putTask \in puttingcomp,\ fixture(putTask) = f)\\ (\forall takeTask \in taking(comp),\ fixture(takeTask) = f,\\ componentsUsed(putTask) \subset taskSubComponents(takeTask))\\ pred(putTask) \neq takeTask\end{aligned} \tag{A.5}$$

As with constraint 4.66, but we limit the predecessors instead.

$$\begin{aligned}(\forall group \in \{1, \ldots, nbrConcurrentGroups\})\\ (\forall t_1 \in concurrentTasks(group))\\ (\forall t_2 \in concurrentTasks(group)/\{t_1\})\\ pred(t_1) \neq t_2 \wedge pred(t_2) \neq t_1\end{aligned} \tag{A.6}$$

Since concurrent tasks need to happen simultaneously on different machines, they cannot be the predecessor to each other.

$$\begin{aligned}(\forall t_1 \in tasks,\ componentCreated(t_1) > 0)\\ (\forall t_2 \in tasks,\ componentCreated(t_1) \in componentUsed(t_2))\\ pred(t_1) \neq t_2\end{aligned} \tag{A.7}$$

Sub-assembly components can only be used after they are created. Therefore, we can say that a task that uses a component created at task t cannot be the predecessor of task t.

$$
\begin{aligned}
&(\forall precTask \in tasks) \\
&(\forall t \in tasks,\ precTask \neq t, \\
&\ componentUsed(precTask) \cup taskCompleteSubComponents(t) \\
&\qquad\qquad \subset taskCompleteSubComponents(t), \\
&\ componentsUsed(precTask) \cup taskCompleteSubComponents(t) \neq \emptyset) \\
&\ pred(precTask) \neq t
\end{aligned}
\tag{A.8}
$$

As in 4.70, tasks has to be performed before the tasks having the component in the task as sub-component. This means the task cannot have any of these tasks as predecessor.

$$
\begin{aligned}
&(\forall concGroup \in concurrentTasks,\ |concGroup| = nbrMachines) \\
&\quad concComps = \bigcup_{\forall i \in concGroup} componentsUsed(i), \\
&concSubComps = \bigcup_{\forall i \in concGroup} taskCompleteSubComponents(i), \\
&\quad postTasks = \{postTask :\ postTask \in tasks, \\
&\qquad\qquad concComps \cap taskCompleteSubComponents(postTask) \neq \emptyset\} \\
&\quad preTasks = \{preTask :\ preTask \in tasks, \\
&\qquad\qquad componentsUsed(preTask) \cap concSubComps \neq \emptyset\}, \\
&\quad (\forall postTask \in postTasks) \\
&\quad (\forall predTask \in preTasks) \\
&pred(postTask) \neq preTask
\end{aligned}
\tag{A.9}
$$

If there is a group of concurrently executing tasks that take up all machines available they will act as a wall between the tasks before and after the group. It is guaranteed that the tasks after the group cannot have the tasks before the group as predecessors. We can extract the tasks before and after the concurrent tasks by analysing the components used, since the components used in the concurrent tasks will have the components used before as sub-components.

Appendix B
File & Tool Manuals

The data and tools used in this thesis are available for free to download and use at `https://github.com/Arclights/Thesis-Tools`. In this section we will describe the tools and data. All the tools are written in Java and are run using the `java -jar` command.

B.1 File Formats

There are two files used to produce the data for the solvers; it is the XML file that describes the assembly, and the time matrix file that describes the time it takes an arm to move between two tasks. For complete versions of the files used in the thesis see the link above.

B.1.1 Assembly XML

In order to more easily create the data needed by the solvers, we created an XML format that that is more easy to deal with, that later is translated into MiniZinc code. For an outline of the XML format see listing B.1. Note that although it contain the basic parts, it is not a legit assembly XML file as it is not complete.

All ids must be unique within the area they are used. So for example there can only be one tray called "tray1", but there could be a fixture called "tray1" as well, but that would be bad practice since it would be confusing. Ids used must also be declared before they are used again. For example a component must be declared through a `Component` tag before being referenced to in a task. The number of declarations can theoretically be infinite, but since everything will be represented by integers in the model we are practically limited to the limit of integers.

The `Output` tag, `Tray` tag and `Fixture` tag defines an output, a tray and a fixture respectively. The `Component` tag defines a component. All components used in the assembly needs to be defined, including the sub assemblies, since we treat sub assemblies

as components in our model. We also describe what components make up a subcomponent using the `SubComponent` tag, it can both be regular components or sub assemblies.

To define tools we use the `tool` tag and to define machines we use the `Machine tag`

To define a task we use the `Task` tag. Together with the id we specify the length of the task in some time unit. Inside the tag we declare the properties of the task. If the task is performed in a tray, we specify the tray used. The same goes for fixture. Only one of them can be declared at a time, since we cannot be at a tray and a fixture at the same time. We specify which components are used in the task. There can be multiple components associated with a task, but it is limited by the number of machines available. Although the translation program does not check whether or not this limit is exceeded. If there is a component created at the task we specify if by the `ComponentCreated` tag. There can only be a most one component created per task and can only occur in tasks where the action is mounting, but this is not checked by the translator either. If there is a particular tool needed for the task we specify it with the `ToolNeeded` tag. We specify what kind of action a task is using the `Action` tag.

To define a set of tasks that comes in an ordered group we use the `OrderedGroup` tag. The order of them listed in the XML file is the order in which they will be scheduled. There can be multiple ordered groups specified.

To define a set of task that needs to be performed concurrently we use the `ConcurrentGroup` tag. The order in the XML does not matter.

To define a set of tasks out of range of a machine we use the `TasksOutOfRange` tag. The id is the id of the machine that the tasks are out of range for. This is not declaring the machine, so it needs to be defined previously.

To define the tool change durations we use the `ToolChangeDurations` tag and for each tool change we want to define we use the `Change` tag. And we supply the tool we are changing from, the tool we are changing to and the duration it takes to perform.

B.1.2 Time Matrix

To supply the time it takes to move between the tasks we provide a time matrix using a time matrix. A row represents the task we are going from and the column represents the task we are going to. There are as many columns as there are tasks in the XML file, but there are one more rows than there are columns. This is to account for the starting position of the machines.

The file format is a comma separated values file (CSV). To identify which row and column belongs to the correct task, each row and column starts with the corresponding id from the XML file. The row for the start task must be named *Start*. The times from every task to the place where the tools are changed and vice versa needs to be included as a column and a row with the name *Change tool*. The order in which the ids comes in the row and columns has to be the same. The file needs to be separated with semicolons in order to allow for commas in the ids. This means that the ids used in the assembly cannot have semicolons in them. The values for the times can be decimal values, they will be rounded of after the calculations by the translator. Figure B.1 shows an example of the structure of the time matrix.

```
<Assembly>
   <Output id="Output"/>

   <Tray id="top-tray"/>

   <Fixture id="Front fixture"/>

   <Component id="Top"/>
   <Component id="Button"/>
   <Component id="Top-Button"/>

   <Subcomponents id="Top-Button">
      <Component id="Top"/>
      <Component id="Button"/>
   </Subcomponents>

   <Tool id="tool1"/>
   <Tool id="tool2"/>

   <Machine id="m1"/>

   <Task id="Mount button on top" Duration="25">
      <Tray id="top-tray"/>
      <Fixture id="Front fixture"/>
      <Component id="Top"/>
      <Component id="Button"/>
      <ComponentCreated id="Top-Button"/>
      <ToolNeeded id="tool1"/>
      <Action id="Mounting"/>
   </Task>

   <OrderedGroup>
      <Task id="Angle top-button"/>
      <Task id="Lift top-button, hold top-button"/>
      <Task id="Turn top-button"/>
   </OrderedGroup>

   <ConcurrentGroup>
      <Task id="Lift top-button, hold top-button"/>
      <Task id="Lift top-button, support"/>
   </ConcurrentGroup>

   <TasksOutOfRange id="m1">
      <Task id="Take ring"/>
      <Task id="Take bottom"/>
      <Task id="Take switch"/>
   </TasksOutOfRange>

   <ToolChangeDurations>
      <Change FromToolId="tool1" ToToolId="tool2" Duration="60"/>
      <Change FromToolId="tool2" ToToolId="tool1" Duration="60"/>
   </ToolChangeDurations>
</Assembly>
```

Listing B.1: The basic parts of the assembly XML. This is not a legitimate assembly file.

	Take top	Put top in fixture	...	Mount button on top
Start	3	7	...	7
Take top	0	8	...	8
Put top in fixture	8	0	...	0
⋮	⋮	⋮	⋱	⋮
Mount button on top	8	0	...	0

Figure B.1: Example of time matrix structure

B.1.3 MiniZinc data file

This file is not included in the link above, but it is generated by `AssemblyConv`, see below. In the evaluation we test the model with combinations of the two different filters. To toggle the filters there are two parameters that can be set at the start of the file, `tempFilter` and `predFilter`. These are boolean variables that indicate if the temporal filter or the predecessor filter should be turned on respectively.

B.2 AssemblyConv

`AssemblyConv` is the program used to convert the XML file and the time matrix file into data for the solver. It takes the matrix file and the XML as arguments and produces a MiniZinc file. For a detailed description of the syntax run the program without parameters.

B.3 SchedPrinter

`SchedPrinter` is used for visualising the outputted assembly from the solvers. It creates a Gant diagram in ASCII and outputs it to the screen. To get it to a file one can simply pipe it to one and it can then be shown in a regular text editor. But make sure that text wrapping is turned of as it will interfere with the visualisation. It takes a text file containing the output from the solver. This can easily be obtained by piping the output from a solver into a file. The program also takes an argument whether the text file provided is in the format that the JaCoP solver provides or the format G12 provides. The solvers using the G12 format is G12/FD, Gecode and or-tools, and the solvers using the JaCoP format is JaCoP and Choco3. The output format for Opturion CPX is nknown since we have not been able to run the model on it. When printing from the JaCoP format, the user has to supply the `.dzn` file so the printer can know the name of the tasks. For a detailed description of the syntax run the program without parameters or with `-h` as parameter. The output of the tasks with duration 0 will look weird, but that is because we cannot fit any characters in a box with a width of 0.

B.4 FZNstat

To obtain the statistics used in chapter 5 we use `FZNstat`. The program does not take any parameters but will go over all the `.fzn` files in the directory it is run in and produce a result file for each of the `.fzn` files. The names of the result files are the same as the `.fzn` files but with `_stat` at the end. So for example `jacop.fzn` will get the result file `jacop_stat`. The result files will contain a little bit more information about individual constraints than what was presented in chapter 5.