Task scheduling of multiple industrial robots through constraint programming

Tommy Kvant
ada09tkv@student.lu.se

January 19, 2015

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

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

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

Abstract

Keywords: key word

Acknowledgements

Contents

Chapter 1

Introduction

1.1 Background

More and more of the production in today's society is getting automated. Manufacturers want to cut cost and make the production more effective by eliminating the human work and replace it with robots. But there are drawbacks; robots are expensive and robots does not have the versatility of a human. This puts pressure on the robot manufacturers to develop robots that are more versatile. Thus eliminating the need for manufacturers to have multiple robots to do multiple tasks and thereby lowering costs. And also by making the robots more versatile the 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 oppose to flexible robots which will be performing many different tasks and assemblies. This poses the demand for the scheduling of such robots to be flexible as well. A scheduling of a robot can be a time consuming task. Since manufacturers want as effective assemblies as possible, it can take from days to weeks to perfect an assembly schedule. This is not feasible if you want to use the robot for many different tasks and assemblies. In this thesis we would like to try and automate this scheduling process in order to cut down on the scheduling time. To accomplish this we will be using Constraint programming, as it provides a general interface to solve problems without needing to build a complete framework from scratch. Also, scheduling is a classical constraint problem, thus constraint programming suits this problem well.

One of those robots are ABB's robot YuMi®(formerly known as FRIDA). YuMi®is a dual armed robot made to work along side humans and able to perform the some of the most complex tasks, such as mount a nut or thread a needle.[1] It accomplishes this by using a wide variety of sensors, e.g. force sensor, visual sensors, etc. Usually robot replaces humans to perform dangerous or heavy tasks, YuMi®is mainly designed for small parts assembly, i.e. usually humans roles in todays manufacturing environment.

1.2 Problem specification

What does our problem look like

1.3 Related work

[brucker 2009] mentions the solving of 15x15 benchmark(15 jobs with 15 operations) being solved without heuristics (although not necessarily in CP).[3]

[Thörnblad] concludes that when a cell is part of an assembly flow, the use of targeting due dates is to prefer. Because makespan can exacerbate an already unreliable flow. The assembly we perform is not a part of a flow, and such we do not concern ourselves with maintaining a stable flow through the cell, but only to optimize the assembly in the cell.[13]

[Garey] shows that job shop problems for size $m \geq 2$ and $n \geq 3$ are NP-complete [6]

[yuan] says pure CP is only effective on small problems of FJSP. To effectively perform larger sizes, methods such as discrepancy search, large neighborhood search(LNS) or iterative flattening search. It also shows LNS together with Hybrid Harmonic Search produces good results.[19]

[ejenstam] [andra test med solvers]

1.4 Report structure

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 solution and not the algorithm to solve it. This is done by specifying the solution using *domain variables*, or simply variables, and *constraints*. Variables have a domain of values, meaning they can represent each 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
$\{Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday\}$.

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 $=$, $<$, $\leq$, $>$, $\geq$. Together with the function symbols and the constraint relation symbols, one can create simple constraint, called *primitive constraints*. An example of a primitive constraint is $X < Y$, i.e. the values in X's domain has to 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 has to hold when the model is evaluated, all constraints are joined in a conjunction.(**?**) The disjunctive connective $\vee$ is also available and can be used in the same way as $\wedge$. In some cases the logical implications; reverse implication ($\leftarrow$), forward implication ($\rightarrow$) and bi-implication ($\leftrightarrow$) are also available. Here, for example, the

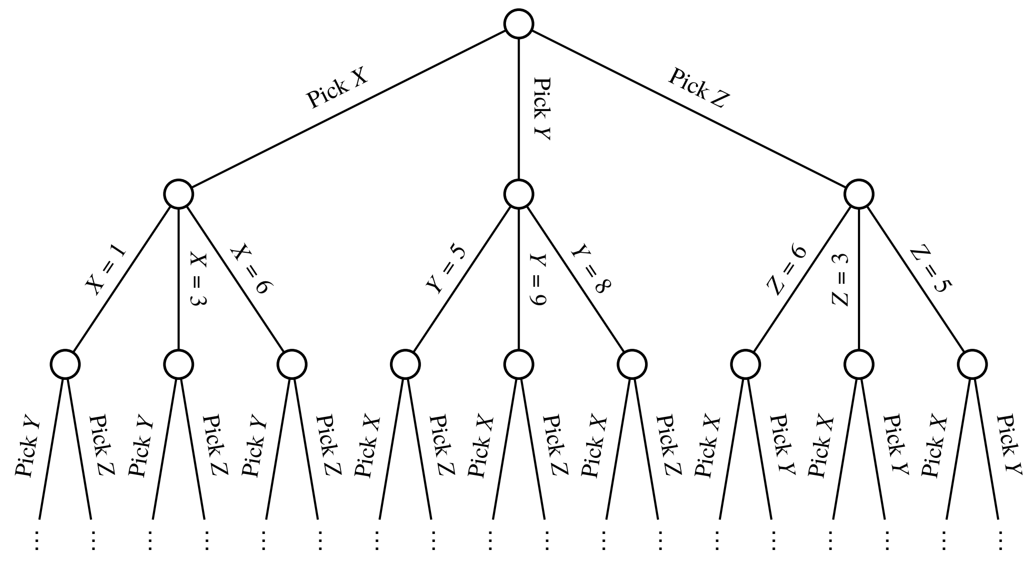

Figure 2.1: The beginning of the search space for the variables X, Y, Z, where $X = \{1, 3, 6\}$ $Y = \{5, 9, 8\}$ $Z = \{6, 3, 5\}$

implication works by taking two logical statement and if the first one evaluates to *true* the other statement should hold as well. For example $X = 0 \rightarrow Y > 5$ says that if $X = 0$ then Y should be larger than 5.

For example, we have a problem with the variables x and y. $x = 4$ and $y = \{1..10\}$. Here the x has the value 4 and can thereby only assume the value 4. y on the other hand can assume 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 get rid of the lower part of y's domain 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

The general discription of the global constraints in this section come from the *Global Constraint Catalogue* [2] and the adaption in MiniZinc is from MiniZinc:s global constraints listing [8].

All Different Constraint (`allDifferent`)

The `allDifferent` constraint is pretty straight forward. It takes a collection `VARIABLES` as argument and enforces all variables in `VARIABLES` to assume distinctly different values, i.e. all values will be different.

In MiniZinc, `VARIABLES` is called `x` and consists of a set of variables.

Circuit Constraint (`circuit`)

The `circuit` constraint takes a collection of nodes `NODES`, where each node has an index and a successor. `circuit` enforces the nodes to form a *Hamiltonian circuit*.

In MiniZinc, `NODES` is called `x` and consists of an array with variables where the index in the array is the index for the variable in that place and the successor value is the value of the variable.

Cumulative Constraint (`cumulative`)

The `cumulative` constraint takes to arguments, a collection of tasks `TASKS` and a limit `LIMIT` for how many tasks can overlap simultaneously. The tasks have 5 attributes; `origin`, `duration`, `end` and `height`. The attributed are pretty self explanatory; `origin` is where in time the task starts, this is the value the constraint tries to determine, `duration` is the duration of the task, the `end` is at which time the task ends. `Height` might not be as straight forward though, it means how many resources the task consumes. `LIMIT` is the limit for how many resources there are available. So if we have a limit of 5 resources and we have two tasks which consumes 3 and 2 resources respectively, they can execute simultaneously. But not if the limit was 3.

In MiniZinc, `TASKS` is split into 3 arguments; `s`, `d` and `r`. Each of them are an array where the index represents a task. The `end` variable is skipped and is implied by the `s` and `d` arguments. `r` is *height* here.

Global Cardinality constraint (`global_cardinality`)

The `global_cardinality` constraint take two arguments; a collection of variables `VARIABLES` and a collection of values `VALUES`. Each element in `VALUES` have two attributes; `val` and `noccurence`. The constraint ensures that `val` is covered `noccurence` times by the elements in `VARIABLES`.

In MiniZinc, `VARIABLES` is split into two arguments; `cover` and `counts`, where `cover` corresponds to `val` and `counts` corresponds to `noccurence`. Both `val` and `counts` are arrays where the index identifies the element.

2.1.3 Solver

A constraint programming program is consisting of many of these constraints and variables. When the solution 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 the 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. The solver accomplishes this by doing a *search* on the space of possibilities, i.e. the *search space*. The search space is in 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 variables 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.

[14] [7] [12]

2.1.4 Reified

2.2 Job-shop scheduling problem

2.2.1 Flexible job-shop scheduling problem

2.3 MiniZinc

2.4 Solvers

2.4.1 G12

[12]

2.4.2 JaCoP

[11]

2.4.3 Gecode

[10]

2.4.4 OR-tools

[15]

2.4.5 Opturion CPX

[9]

2.4.6 Choco3

[4]

Chapter 3
Model

This model is inspired by Ejenstams work [5]. 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. This is common robot 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 does not incorporate a fixture. We have used a similar way of generalizing tasks by labeling them with *tray*, *fixture*, etc. but extended it.

Before going in to too much detail we will give a brief overview of how the scheduling works. We are working with tasks. A task is a action that manipulates a component in some way and is performed at a certain spacial coordinate in the room. Although, the model does not care about the exact coordinates, but rather the time 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 before hand and put in a matrix which is used to generate the input file for the model. How to do that is described in appendix B.

The robots that the model schedules have arms, these we call machines. And no matter if the assembly has two one-armed robots or one two-armed robot, the number of machines are the same and will be scheduled the same. To compensate for the placement of the machines there are variables that can be set as shown below. The arms can have a set of tools that they can change between depending on what is required by the task. 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, where should we put the change of tools of the machines. If a change occurs between two tasks, it will be noted by the move taking longer than it usually would.

The goal for the assembly is to assemble components. This is components fed to the assembly from the outside, for example button components. The final assembly is the complete assembly of components that make the final product. All the smaller assemblies before that are called sub-assemblies. For reasons explained further down, we will in this thesis call components such as buttons for *primitve* components instead of just components.

All the data used by the model can be generated using the tools described in appendix B.

3.1 Variables

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

3.1.1 Model Variables

- $nbrTasks$
- $nbrMachines$
- $nbrTools$
- $nbrTrays$
- $nbrFixtures$
- $nbrComponents$
- $nbrOutputs$
- $nbrConcurrentGroups$
- $nbrOrderedGroups$
- $tray_t$
- $output_t$
- $fixture_t$
- $componentsUsed_t$
- $mounting$
- $taking$
- $moving$
- $putting$
- $concurrentTasks_k$
- $order_k$
- $toolNeeded_t$
- $changeToolDuration_{tool_1,tool_2}$
- $duration_t$
- $taskSubComponents_t$
- $taskCompleteSubComponents_t$
- $timeMatrix3D_{t1,t2}$

task	task	task	task	sTask1	sTask2	gTask1	gTask2

Figure 3.1: An example of the tasks and start and goal tasks seen as an array for an assembly with 4 tasks and 2 machines

3.1.2 Static variables

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

$$nbrTasks \in \{1, \dots, 2^{32} - 1\} \tag{3.1}$$

$$tasks = \{1, \dots, nbrTasks\} \tag{3.2}$$

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

$$startTasks = \{nbrTasks + 1, \dots, nbrTasks + nbrMachines\} \tag{3.3}$$

$$goalTasks = \{nbrTasks + nbrMachines + 1, \dots, nbrTasks + nbrMachines \times 2\} \tag{3.4}$$

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 much start and goal tasks as there are machines. They are arranged such that first comes all the start tasks 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 something like figure 3.1.

$$allTasks = tasks \cup startTasks \cup goalTasks \tag{3.5}$$

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

$$nbrMachines \in \{1, \dots, 2^{32} - 1\} \tag{3.6}$$

$$machines = \{1, \dots, nbrMachines\} \tag{3.7}$$

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

$$nbrTools \in \{1, \dots, 2^{32} - 1\} \tag{3.8}$$

$$tools = \{1, \dots, nbrTools\} \tag{3.9}$$

$$toolNeeded(t) \in tools, \;\; t \in tasks \tag{3.10}$$

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 task t needs.

$$nbrComponents \in \{1, \ldots, 2^{32} - 1\} \tag{3.11}$$

$$components = \{1, \ldots, nbrComponents\} \tag{3.12}$$

$$componentsUsed(t) \subset components, \ t \in tasks \tag{3.13}$$

$$componentCreated(t) \in components \cup \{0\}, \ t \in tasks \tag{3.14}$$

$nbrComponents$ defines the number of components used. All components needs to be uniquely identified in the assembly, so even if we use 4 screws in an assembly, we need to define all 4 screws. As mentioned before we distinguish between components and *primitve* components. The reason for that 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 is as *compoentCreated* for the task where it is created. This cannot happen anywhere else other than at a mount task, although there is no check in the model for it. If there is not component created at a task, $componentCreated = 0$

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.

$$taskSubComponents(t) \subset components, \ t \in tasks \tag{3.15}$$

$taskSubComponents(t)$ 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.

$$taskCompleteSubComponents(t) \subset components, \ t \in tasks \tag{3.16}$$

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 components themselves. If the origin component is a *primitve* component, the set is empty.

$$subComponents(c) \subset components, \ c \in components \tag{3.17}$$

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.

$$nbrTrays \in \{1, \ldots, 2^{32} - 1\} \tag{3.18}$$

$$trays = \{1, \ldots, nbrTrays\} \tag{3.19}$$

$$tray(t) \in trays \cup \{0\},\ t \in tasks \tag{3.20}$$

The trays available in the assembly, *trays*. Trays are used to hold components until we need them in the assembly. This 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, $tray(t) = 0$.

$$nbrFixtures \in \{1, \ldots, 2^{32} - 1\} \tag{3.21}$$

$$fixtures = \{1, \ldots, nbrFixtures\} \tag{3.22}$$

$$fixture(t) \in fixtures \cup \{0\},\ t \in tasks \tag{3.23}$$

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 will be shown in the assembly example [section?], the fixture can be used for purposes than just holding components.

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

$$nbrOutputs \in \{1, \ldots, 2^{32} - 1\} \tag{3.24}$$

$$outputs = \{1, \ldots, nbrOutputs\} \tag{3.25}$$

$$output(t) \in outputs \cup \{0\},\ t \in tasks \tag{3.26}$$

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, $output(t) = 0$.

$$nbrConcurrentGroups \in \{1, \ldots, 2^{32} - 1\} \tag{3.27}$$

$$concurrentGroups = \{1, \ldots, nbrConcurrentGroups\} \tag{3.28}$$

$$concurrentTasks(k) \subset tasks,\ k \in concurrentGroups \tag{3.29}$$

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.

The *k* set of tasks needing concurrent execution *nbrConcurrentGroups* defines the number of concurrent groups used.

$$nbrOrderedGroups \in \{1, \ldots, 2^{32} - 1\} \tag{3.30}$$

$$orderedGroups = \{1, \ldots, nbrOrderedGroups\} \tag{3.31}$$

$$orderedGroup(k) \subset tasks, \; k \in orderedGroups \tag{3.32}$$

$$ordered(k, i) \in tasks, \; i \in \{1, \ldots, |orderedGroup(k)|\}, \; k \in orderedGroups \tag{3.33}$$

$$orderedSet = \bigcup_{\forall k \in orderedGroups} order(k), \; orderedSet \subset tasks \tag{3.34}$$

$orderedGroup(k)$ is the k:th ordered group specified, there are $nbrOrderedGroups$ ordered groups. An ordered group is an array of tasks that has to come in a very specific order. An example of this could be if an assembly has many move tasks that needs to be performed one after another in order to make an intricate movement. As will be showed in the constraints section, 3.2, we can reason the relation between tasks if they use a certain component and are a certain kind of action. But we can not reason using 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 com in the array. All the tasks in the group will be performed on the same machine, it can not order tasks on different machines.

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 an ordered group.

$tray(t)$, $output(t)$ and $fixture(t)$ can not be set at the same time for a task, since that would mean 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 are that output can not be used by a take task and tray can not be used by a mount task. If ones assembly contains these combinations, the output or tray should be changed to a fixture.

$$mounting \subset tasks \tag{3.35}$$

$$taking \subset tasks \tag{3.36}$$

$$moving \subset tasks \tag{3.37}$$

$$putting \subset tasks \tag{3.38}$$

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

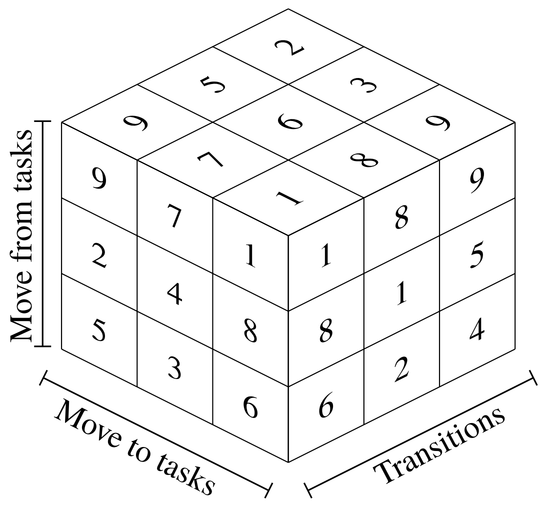

Figure 3.2: The timeMatrix3D

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 components 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.

$$putting(c) \subset putting,\ c \in components \tag{3.39}$$

$$mounting(c) \subset mounting,\ c \in components \tag{3.40}$$

$$taking(c) \subset taking,\ c \in components \tag{3.41}$$

$$moving(c) \subset moving,\ c \in components \tag{3.42}$$

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

$$duration(t) \in \{0, \ldots, 2^{32} - 1\},\ t \in tasks \tag{3.43}$$

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

For the model to decide how long a move between two tasks should be and if there should be a change of tool in between, a matrix is used, $timeMatrix3D$, see figure 3.2. This is a 3-dimensional matrix and contains the times for moving between all the tasks depending

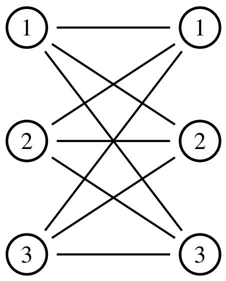

Figure 3.3: All the transitions between the tool states

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.

$$timeMatrixDepth = \frac{n^2 - n + 2}{2}, \ n = nbrTools \tag{3.44}$$

$$\begin{aligned} timeMatrix3D(t(from), t(to), k) \in \{0, \dots, 2^{32} - 1\}, \\ t(from) \in tasks \cup startTasks, \\ t(to) \in tasks, \ k \in \{0, \dots, timeMatrixDepth\} \end{aligned} \tag{3.45}$$

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 too much redundancy.

What we mean with ”different transitions” is easiest to show through an example. Lets say we have 3 tools available for each machine. We consider each tool state as a node in a graph, see figure 3.3, 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 consider the rest of the transitions this way, we will end up with a reduced number of transitions, in our case 4, see figure 3.4

It has been observed that using the reasoning above, *timeMatrixDepth* follows the function 3.44.

$$taskOutOfRange(m) \subset tasks, \ m \in machines \tag{3.46}$$

Depending on the physical layout of the assembly, sometimes not all tasks can be done with 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 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 other than this the machines in the model are identical and has the potential perform the same work.

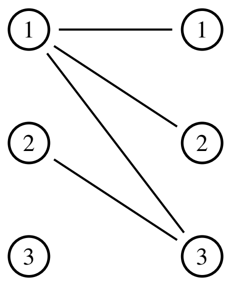

Figure 3.4: The reduced number of transitions between the tool states

3.1.3 Decision variables

Decision variables are variables that can take on many values. It is these values that the solver set out to determine in order to solve the problem.

$$usingMachine(t) \in machines, \ t \in tasks \tag{3.47}$$

The model has to decide which task uses which machine.

$$pred(t) \in allTasks, \ t \in allTasks \tag{3.48}$$

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

$$\begin{aligned} maxE =& (max(\{duration(t) : t \in tasks\}) + \\ & max(\{timeMatrix3D(t_1, t_2, k) : \\ & \quad \forall t_1 \in tasks \cup startTasks, \\ & \quad \forall t_2 \in tasks, \\ & \quad \forall k \in \{0, \ldots, timeMatrixDepth\}\}) \times nbrTasks \end{aligned} \tag{3.49}$$

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.

$$start(t) \in \{0, \ldots, maxE\}, \ t \in allTasks \tag{3.50}$$

$$end(t) = start(t) + duration(t), \ t \in allTasks \tag{3.51}$$

Each task has to start 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.

$$moveDuration(t) \in \{0, \ldots, maxE\}, \ t \in allTasks \tag{3.52}$$

$$moveStart(t) \in \{0, \ldots, maxE\}, \ t \in allTasks \tag{3.53}$$

$$moveEnd(t) = moveStart(t) + moveDuration(t), \ t \in allTasks \tag{3.54}$$

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 is needed for both of the tasks, the duration for the move is a decision variable as opposed to the duration for the task itself.

$$makespan \in \{0, \ldots, maxE\} \tag{3.55}$$

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.

$$toolUsed(t) \in tools, \; t \in allTasks \tag{3.56}$$

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.

3.2 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*, for the MiniZink code see *Appendix B*.

$$(\forall t \in tasks) \; end(t) \leq makespan \tag{3.57}$$

makespan should represent the total time of the whole assembly, since it is that which we want to minimize. That means it should be equal to the largest end time amongst all the tasks. We can enforce that by limiting the end time for each task to be less or equal to the *makespan*.

$$(\forall t \in startTasks \cup goalTasks) \; start(t) = 0 \tag{3.58}$$

$$\begin{aligned} &(\forall m \in machines) \\ &\qquad usingMachine(nbrTasks + m) = m \land \\ &usingMachine(nbrTasks + nbrMachines + m) = m \end{aligned} \tag{3.59}$$

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 the 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.

$$\begin{aligned} &(\forall m \in machines) \\ &\quad (\forall t \in tasksOutOfRange(m)) \\ &usingMachine(t) \neq m \end{aligned} \tag{3.60}$$

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

$$(\forall t \in tasks, \; toolNeeded(t) \neq 0) \; toolUsed(t) = toolNeeded(t) \tag{3.61}$$

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 assign it to *toolUsed* for the tasks where a tool is specified, i.e. *toolNeeded* is not 0.

3.2.1 Precedences

These constraints deals with the order in time in which the tasks as to come.

$$(\forall t \in tasks)\ Start(t) \geq moveEnd(t) \tag{3.62}$$

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.

$$\begin{gathered}(\forall comp \in components)\\ (\forall mountTask \in mounting(comp))\\ (\forall putTask \in putting(comp))\\ end(putTask) \leq moveStart(mountTask)\end{gathered} \tag{3.63}$$

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.

$$\begin{gathered}\forall comp \in components\\ \forall mountTask \in mounting(comp),\\ \forall takeTask \in taking(comp),\\ end(takeTask) \leq moveStart(mountTask)\end{gathered} \tag{3.64}$$

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.

$$\begin{aligned}&\forall comp \in components\\ &(\forall putTask \in putting(comp),\ tray(putTask) > 0)\\ &(\forall takeTask \in taking(comp),\ tray(putTask) = tray(takeTask))\\ &end(putTask) \leq moveStart(takeTask)\end{aligned} \tag{3.65}$$

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 3.65 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.

$$\begin{aligned}&(\forall f \in fixtures)\\ &(\forall putTask \in putting,\ fixture(putTask) = f)\\ &(\forall takeTask \in taking,\ fixture(takeTask) = f\ \wedge\\ &\qquad componentsUsed(putTask) \subset taskSubComponents(takeTask))\\ &end(putTask) \leq moveStart(takeTask),\end{aligned} \tag{3.66}$$

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.

$$\begin{aligned}
&(\forall f \in fixtures)\\
&puts = [put : \forall put \in putting,\ fixture(put) = f],\\
&takesForEachPut = [\{take : \forall take \in taking,\ fixture(take) = f,\\
&componentsUsed(put) \subset taskCompleteSubComponent(take)\} : \forall put \in puts],\\
&takes = [\underset{\forall take \in takesForEachPut(p)}{\arg\min}\ taskCompleteSubComponent(take) :\\
&\qquad \forall p \in \{1, \ldots, |puts|\}],\\
&cumulative([moveStart(task) : \forall task \in puts],\\
&\qquad [abs(end(takes(i)) - moveStart(puts(i))) : \forall i \in \{1, \ldots, |puts|\}],\\
&\qquad [1 : \forall i \in \{1, \ldots, |puts|\}],\\
&\qquad 1)
\end{aligned} \tag{3.67}$$

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.

$$\begin{aligned}
&(\forall group \in \{1, \ldots, nbrConcurrentGroups\})\\
&\quad (\forall t_1 \in concurrentTasks(group))\\
&\quad (\forall t_2 \in concurrentTasks(group)/\{t_1\})\\
&start(t_1) = start(t_2) \wedge\\
&usingMahine(t_1) \neq usingMachine(t_2),
\end{aligned} \tag{3.68}$$

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 3.68.

$$\begin{aligned}
&(\forall t_1 \in tasks,\ componentCreated(t1) > 0)\\
&(\forall t_2 \in tasks,\ componentCreated(t_1) \in compinentUsed(t_2))\\
&moveStart(t_2) \geq end(t_1)
\end{aligned} \tag{3.69}$$

A very logical observation we can do is that components cannot be used before they are created. This is enforced in 3.69.

$$\begin{aligned}
&(\forall precTask \in tasks) \\
&\qquad (\forall t \in tasks,\ precTask \neq t, \\
&\qquad\quad componentUsed(precTask) \cup taskCompleteSubComponent(t) \subset \\
&\qquad\quad taskCompleteSubComponents(t), \\
&\qquad\quad componentsUsed(precTask) \cup taskCompleteSubComponents(t) \neq \emptyset) \\
&end(precTask) \leq moveStart(t),
\end{aligned} \tag{3.70}$$

A similar observation as for 3.69 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 needs to have an end time smaller than the start time of the tasks having the tasks component as sub-component.

$$\begin{aligned}
&(\forall f \in fixtures) \\
&fixtureTasks = [t : \forall t \in tasks,\ fixture(t) = f], \\
&cumulative([start(t) : \forall t \in fixtureTasks], \\
&\qquad [duration(t) : \forall t \in fixtureTasks], \\
&\qquad [1 : t \in fixtureTasks], \\
&\qquad 1)
\end{aligned} \tag{3.71}$$

$$\begin{aligned}
&(\forall tr \in trays) \\
&trayTasks = [t : \forall t \in tasks,\ tray(t) = tr], \\
&cumulative([start(t) : \forall t \in trayTasks], \\
&\qquad [duration(t) : \forall t \in trayTasks], \\
&\qquad [1 : t \in trayTasks], \\
&\qquad 1)
\end{aligned} \tag{3.72}$$

$$\begin{aligned}
&(\forall o \in outputs) \\
&outputTasks = [t : \forall t \in tasks,\ output(t) = o], \\
&cumulative([start(t) : \forall t \in outputTasks], \\
&\qquad [duration(t) : \forall t \in outputTasks], \\
&\qquad [1 : t \in outputTasks], \\
&\qquad 1)
\end{aligned} \tag{3.73}$$

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 3.71, 3.72 and 3.73.

3.2.2 Predecessors

$$\begin{gathered}(\forall startTask \in startTasks/\{nbrTasks + 1\}) \\ pred(startTask) = startTask + nbrMachines - 1\end{gathered} \tag{3.74}$$

$$pred(nbrTasks + 1) = nbrTasks + nbrMachines \times 2 \tag{3.75}$$

$$circuit(\{pred(t) : \forall t \in tasks\}) \tag{3.76}$$

All tasks has to have a predecessor that tells the model what task comes directly before said task 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 3.75. 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.

$$(\forall t \in tasks)\ moveStart(t) \geq end(pred(t)) \tag{3.77}$$

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.

$$\begin{gathered}(\forall t \in tasks \cup goalTasks) \\ usingMachine(t) = usingMachine(pred(t))\end{gathered} \tag{3.78}$$

Another fundamental part is that a predecessor is a task performed on the same machine as the task in question. This is enforced by 3.78.

$$\begin{aligned}&(\forall k \in orderedGroups) \\ &\ (\forall i \in \{1, \ldots, |orderedGroup(k)| - 1\}) \\ &pred(ordered(k, i + 1)) = ordered(k, i)\end{aligned} \tag{3.79}$$

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.

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

$$
\begin{aligned}
&(\forall c \in components)\\
&\quad(\forall mountTask \in mounting(c))\\
&\qquad puts = \{p : \forall p \in putting(c),\\
&\qquad\quad (fixture(p) > 0 \land fixture(p) = fixture(mountTask)) \lor\\
&\qquad\quad (output(p) > 0 \land output(p) = output(mountTask)),\\
&\quad(\forall takeTask \in taking(c),\ takeTask \notin orderedSet,\ puts = \emptyset)\\
&\quad pred(mountTask) = takeTask
\end{aligned}
\tag{3.80}
$$

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 3.79 constraint. So we need to ensure the take task is not in an ordered group either.

$$
\begin{aligned}
&(\forall c \in components,\ moving(c) = \emptyset)\\
&\quad(\forall putTask \in putting(c),\ tray(putTask) = 0)\\
&\quad(\forall takeTask \in taking(c))\\
&\quad pred(putTask) = takeTask
\end{aligned}
\tag{3.81}
$$

As with 3.80, 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 3.80, 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.

$$\begin{aligned}
&(\forall t \in tasks) \\
&\quad k = abs(toolUsed(t) - toolUsed(pred(t))) + 1, \\
&moveDuration(t) = timeMatrix3D(pred(t),\ t,\ k)
\end{aligned} \tag{3.82}$$

3.82 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 $timeMatrix3D$.

3.3 Filter

In [18] [16] [17] 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.

3.3.1 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.

$$\begin{aligned}
&(\forall t \in tasks) \\
&maxMoveDurs(t) = max(\{timeMatrix3D(t, j, k) : \\
&\qquad\qquad \forall j \in tasks, \\
&\qquad\qquad \forall k \in \{1, \ldots, timeMatrixDepth\}, \\
&\qquad\qquad j \neq t\})
\end{aligned} \tag{3.83}$$

$$\begin{aligned}
&(\forall t \in tasks) \\
&minMoveDurs(t) = min(\{timeMatrix3D(t, j, k) : \\
&\qquad\qquad \forall j \in tasks, \\
&\qquad\qquad \forall k \in \{1, \ldots, timeMatrixDepth\}, \\
&\qquad\qquad j \neq t\})
\end{aligned} \tag{3.84}$$

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.

$$maxEnd = \sum_{\forall t \in tasks} duration(t) + maxMoveDurs(t) \tag{3.85}$$

$$minEnd = \frac{\sum_{\forall t \in tasks} duration(t) + minMoveDurs(t)}{nbrMachines} \tag{3.86}$$

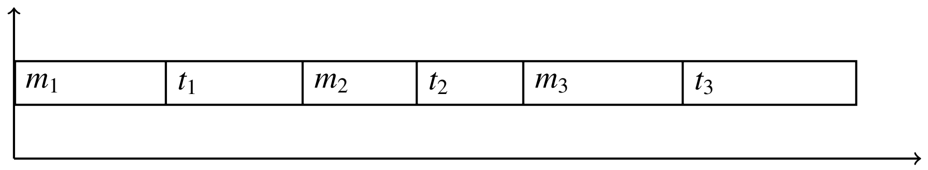

Figure 3.5: The worst case assembly

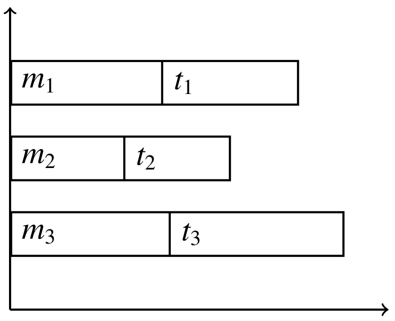

Figure 3.6: The best case assembly

By using the newly created variables we candefine yet another two. These define a new maximum for the total time of the assembly, $maxEnd$. It is similar to $maxE$ in equation 3.49, 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 3.5. 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 3.6. 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 $minEnd$ will always be smaller than the total assembly time.

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

$$\begin{aligned}&(\forall t \in allTasks)\\&start(t) \leq maxEnd - duration(t)\end{aligned} \tag{3.87}$$

$$\begin{aligned}&(\forall t \in Tasks)\\&start(t) \geq minMoveDurs(t)\end{aligned} \tag{3.88}$$

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 3.87 and 3.88 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.

$$(\forall t \in tasks)\ moveStart(t) \leq maxEnd - (duration(t) + minMoveDurs(t)) \tag{3.89}$$

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*.

$$\begin{aligned}makeSpan &\leq maxEnd \land\\ makespan &\geq minEnd\end{aligned} \tag{3.90}$$

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

$$\begin{aligned}&(\forall t \in tasks)\\&(\forall i \in \{0, \ldots, maxMoveDurs(t)\}/\{timeMatrix3D(task, j, k) :\\&\qquad\qquad\qquad\qquad \forall j \in tasks,\\&\qquad\qquad\qquad\qquad \forall k \in \{1, \ldots, timeMatrixDepth\}\\&\qquad\qquad\qquad\qquad t \neq j\})\\&moveDuration(t) \neq i\end{aligned} \tag{3.91}$$

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.

$$\begin{aligned}&(\forall t \in tasks/taking)\\&moveStart(t) \geq min(\{duration(tt) + minMoveDurs(tt) : \forall tt \in taking\})\end{aligned} \tag{3.92}$$

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.

$$\begin{aligned}
&(\forall t \in tasks)\\
&prevTasks = \{task : \forall task \in tasks,\\
&\qquad\qquad componentCreated(task) \in componentsUsed(t)\},\\
&nbrMachines \geq |prevTasks|,\\
&\qquad 0 < |prevTasks|,\\
&start(t) \geq max(\{duration(pt) + minMoveDurs(pt) : \forall pt \in prevTasks\})
\end{aligned} \tag{3.93}$$

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 creates the components that is used in task t, $prevTasks$. These tasks has to come before task t since the component that they create cannot be used before they are created. If the number of machines are 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 proceeding tasks taking the longest to complete.

$$\begin{aligned}
&(\forall t \in tasks)\\
&prevTasks = \{task : \forall task \in tasks,\\
&\qquad\qquad componentCreated(task) \in componentsUsed(t)\},\\
&nbrMachines < |prevTasks|,\\
&start(t) \geq \frac{\left(\sum_{\forall pt \in prevTasks} duration(pt) + minMoveDurs(pt)\right)}{nbrMachines}
\end{aligned} \tag{3.94}$$

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 3.86.

$$\begin{aligned}
&(\forall t \in tasks)\\
&succTasks = \{task : \forall task \in tasks,\\
&componentsUsed(t) \subset taskCompleteSubComponent(task),\\
&componentsUsed(t) \cup taskCompleteSubComponents(task) \neq \emptyset\},\\
&nbrMachines \geq |succTasks|,\\
&\qquad 0 < |succTasks|,\\
&start(t) \leq maxEnd - max(\{duration(st) + minMoveDurs(st) :\\
&\qquad\qquad \forall st \in succTasks\}) - duration(t)
\end{aligned} \tag{3.95}$$

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 3.93 and look at the best case scenario where the tasks are performed concurrently on all machines. The difference here from 3.93 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.

$$\begin{aligned}
&(\forall t \in tasks)\\
&succTasks = \{task : \forall task \in tasks,\\
&componentsUsed(t) \subset taskCompleteSubComponent(task),\\
&componentsUsed(t) \cup taskCompleteSubComponents(task) \neq \emptyset\},\\
&nbrMachines \leq |succTasks|,\\
&start(t) \;\leq maxEnd - \frac{(\sum_{\forall st \in succTasks} duration(st) + minMoveDurs(st))}{nbrMachines}\\
&\qquad\qquad - duration(t)
\end{aligned} \tag{3.96}$$

As with 3.94 we look at the worst case scenario.

3.3.2 Predecessor filter

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

$$(\forall t1, \forall t2 \in taking)\; pred(t1) \neq t2 \tag{3.97}$$

$$(\forall t1, \forall t2 \in putting \cup mounting)\; pred(t1) \neq t2 \tag{3.98}$$

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.

$$\begin{aligned}
&(\forall t \in tasks)\\
&nonPredecessors = \{t_2 : \forall t_2 \in tasks,\\
&componentsUsed(t) \subset taskCompleteSubComponents(t_2) \vee\\
&componentsUsed(t) \subset subComponents(componentCreated(t_2))\}\\
&(\forall nonPred \in nonPredecessors)\\
&pred(t) \neq nonPred,
\end{aligned} \tag{3.99}$$

Using a similar reasoning as in 3.95 and 3.96, 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.

$$\begin{aligned}
&(\forall startTask \in startTasks)\\
&(\forall putTask \in putting)\\
&pred(putTask) \neq startTask
\end{aligned} \tag{3.100}$$

$$\begin{aligned}
&(\forall startTask \in startTasks)\\
&(\forall mountTask \in mounting)\\
&pred(mountTask) \neq startTask
\end{aligned} \tag{3.101}$$

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 no make any difference.

$$\begin{aligned}
&(\forall goalTask \in goalTasks)\\
&(\forall takeTask \in taking)\\
&pred(goalTask) \neq takeTask
\end{aligned} \tag{3.102}$$

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.

$$\begin{aligned}
counts &= \{i : \forall task \in outputTasks,\ i \in \{0, \ldots, 1\}\},\\
outputTasks &= \{task : \forall task \in tasks,\ output(task) > 0\},\\
goalPreds &= \{pred(task) : \forall task \in goalTasks\},\\
&global\_cardinality(goalPreds,\ outputTasks,\ counts) \land\\
&\sum counts > 0
\end{aligned} \tag{3.103}$$

$$\begin{aligned}
counts &= \{i : \forall task \in startTasks,\ i \in \{0, \ldots, 1\}\},\\
takePreds &= \{pred(task) : \forall task \in taking\},\\
&global\_cardinality(takePreds,\ startTasks,\ counts) \land\\
&\sum counts > 0
\end{aligned} \tag{3.104}$$

Continuing the reasoning around what tasks can come first and not we can expand with which tasks has 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 3.103 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 3.104 by placing a constraint over the *pred* variables for the take tasks.

Chapter 4
Assembly

Describing our assembly

Chapter 5
Evaluation

The evaluation of the model

5.1 The Setup

G12

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)	-	-	-	-	-	-	-	-
# of integer variables	174	292	174	313	154	269	154	290
# of boolean variables	162	97	162	97	142	106	142	106
# of arrays	32	46	32	46	30	44	30	44
# of constraints	2555	584	1018	584	2248	558	711	558
% refeid	7.16%	15.92%	17.97%	15.92%	7.25%	17.02%	22.92%	17.02%

JaCoP

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)	635	-	1139619	-	-	-	-	-
# of integer variables	129	223	129	244	129	229	129	250
# of boolean variables	42	27	42	27	42	27	42	27
# of arrays	31	46	31	46	29	43	29	43
# of constraints	2318	425	781	425	2046	416	509	416
% refeid	2.71%	6.35%	8.06%	6.35%	3.07%	6.49%	12.37%	6.49%

Gecode

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)	-	21	-	74019	-	65	-	73496
# of integer variables	129	223	129	244	129	250	129	450
# of boolean variables	42	27	42	27	42	27	42	27
# of arrays	31	50	31	50				
# of constraints	2316	421	779	421	1046	416	509	415
% refeid	2.72%	6.14%	8.08%	6.41%	3.07%	6.49%	12.37%	6.5%

or-tools

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)	257	!	672	!	564	!	600	!
# of integer variables	129	223	129	244	129	229	129	250
# of boolean variables	42	27	42	27	42	27	42	27
# of arrays	31	50	31	50	29	43	29	43
# of constraints	2316	421	779	421	2046	416	509	416
% refeid	2.72%	6.41%	8.08%	6.41%	3.07%	6.49%	12.37%	6.49%

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)	25163591	!	-	!	7332356	!	-	!
# of integer variables	28736	9530	28736	28875	28716	9507	28716	28852
# of boolean variables	57284	18654	57284	57322	57264	18702	57264	57370
# of arrays	32	4694	32	14366	30	28209	30	14382
# of constraints	102519	33021	100982	100675	102212	33024	100675	100678
% refeid	55.91%	56.26%	56.76%	56.86%	56.06%	56.24%	56.92%	56.85%

Opturion CPX - 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)	26318399	!	-	!	7231907	!	-	!
# of integer variables	28736	9530	28736	28875	28716	9507	28716	28852
# of boolean variables	57284	18654	57284	57322	57264	18702	57264	57370
# of arrays	32	4694	32	14366	30	28209	30	14382
# of constraints	102519	33021	100982	100675	102212	33024	100675	100678
% refeid	55.91%	56.26%	56.76%	56.86%	56.06%	56.24%	56.92%	56.85%

Choco3

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)	-	-	-	-	-	-	-	-
# of integer variables	129	223	129	244	129	229	129	250
# of boolean variables	42	27	42	27	42	27	42	27
# of arrays	31	50	31	50	29	43	29	43
# of constraints	2316	421	779	421	2046	416	509	416
% refeid	2.47%	6.41%	8.08%	6.41%	3.07%	6.49%	12.37%	6.49%

Chapter 6
Discussion

The discussion of the results

Concurrent tasks vs. en task med två maskiner. Två tasks ger mer generella constraints, man kan säga att all tasks tar upp en maskin alltid och att man begränsar det med ett constraint att vissa måste ske samtidigt. Man kan ha constraintet att alla tasks kan ha en predecessor. En task gör att man måste kunna säga att tasks kan ta upp till så många maskiner som finns tillgängliga -> fler constraints. Man måste kunna säga att en task ska kunna ha flera predecessors -> fler constraints.

Chapter 7
Conclusions

The conclusions

7.1 Further work

I [ejenstam] kommer de fram till att local search fungerar dåligt.[5] [yuan 2013] menar de att Large neighborhood Search (LNS) tillsammans med Hybrid Harmony Search (HHS) fungerar mycket väl med flexible job shop problem och löser problemet med begränsningen för CP vid dessa problem.[19] Kan vara värt att titta vidare på

Bibliography

[1] ABB. YuMi. `http://new.abb.com/products/robotics/yumi`.

[2] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog, 2nd edition (revision a), February 2012.

[3] Peter Brucker. *Job-shop Scheduling Problem.* Springer, 2009.

[4] Xavier Lorca Charles Prud'homme, Jean-Guillaume Fages. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

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

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

[7] Kim Marriott and Peter J. Stuckey. *Programming with constraints : an introduction*. Cambridge, Mass. : MIT Press, cop., 1998.

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

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

[10] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode*, 2014. Corresponds to Gecode 4.3.2.

[11] Radoslaw Szymanek. JaCoP Overview. `http://jacop.osolpro.com/index.php?option=com_content&view=article&id=19&Itemid=27`, 2010. Accessed: 2015-01-16.

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

[13] Karin Thörnblad, Ann-Brith Strömberg, Michael Patriksson, and Torgny Almgren. An efficient algorithm for solving the flexible job shop scheduling problem. In *25th NOFOMA conference proceedings, June 3-5 2013, Göteborg, Sweden*, page 15, 2013.

[14] Edward Tsang. *Foundations of constraint satisfaction*. Academic Press, 1993.

[15] Nikolaj van Omme, Laurent Perron, and Vincent Furnon. or-tools user's manual. Technical report, Google, 2014.

[16] Petr Vilím. 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, 2002.

[17] Petr Vilím and Roman Barták. 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, May 2002.

[18] Petr Vilím and Roman Barták. Filtering algorithms for batch processing with sequence dependent setup times. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the 6th International Conference on AI Planning and Scheduling, AIPS'02*, pages 312–321. The AAAI Press, 2002.

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

Appendices

Appendix A
Extended Model

This appendix contains constraints that are included in the model, but not as essential for the assembly as the ones in chapter 3.

A.1 Predecessor filter

$$alldifferent(\{pred(t) : \forall t \in tasks\}) \tag{A.1}$$

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.

$$\begin{gathered}
(\forall comp \in components)\\
(\forall mountTask \in mounting(comp))\\
(\forall takeTask \in taking(comp))\\
pred(takeTask) \neq mountTask
\end{gathered} \tag{A.2}$$

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.

$$\begin{gathered}
(\forall comp \in components)\\
(\forall putTask \in puttingcomp,\ tray(putTask) > 0)\\
(\forall takeTask \in taking(comp),\ tray(putTask) = tray(takeTask))\\
pred(putTask) \neq takeTask
\end{gathered} \tag{A.3}$$

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.

$$\begin{aligned}
&(\forall f \in fixtures)\\
&(\forall putTask \in puttingcomp,\ fixture(putTask) = f)\\
&(\forall takeTask \in taking(comp),\ fixture(takeTask) = f,\\
&\qquad componentsUsed(putTask) \subset taskSubComponents(takeTask))\\
&pred(putTask) \neq takeTask
\end{aligned} \tag{A.4}$$

As with constraint 3.66, but we limit the predecessors instead.

$$\begin{aligned}
&(\forall group \in \{1, \dots, nbrConcurrentGroups\})\\
&(\forall t_1 \in concurrentTasks(group))\\
&(\forall t_2 \in concurrentTasks(group)/\{t_1\})\\
&pred(t_1) \neq t_2 \land pred(t_2) \neq t_1
\end{aligned} \tag{A.5}$$

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

$$\begin{aligned}
&(\forall t_1 \in tasks,\ componentCreated(t_1) > 0)\\
&(\forall t_2 \in tasks,\ componentCreated(t_1) \in componentUsed(t_2))\\
&pred(t_1) \neq t_2
\end{aligned} \tag{A.6}$$

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.

$$\begin{aligned}
&(\forall precTask \in tasks)\\
&(\forall t \in tasks,\ precTask \neq t,\\
&componentUsed(precTask) \cup taskCompleteSubComponent(t) \subset\\
&\qquad\qquad taskCompleteSubComponents(t),\\
&componentsUsed(precTask) \cup taskCompleteSubComponents(t) \neq \emptyset\\
&pred(precTask) \neq t
\end{aligned} \tag{A.7}$$

As in 3.70, 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.

$$
\begin{aligned}
&(\forall concGroup \in concurrentTasks,\ |concGroup| = nbrMachines)\\
&\quad concComps = \bigcup_{\forall i \in concGroup} componentsUsed(i),\\
&\quad concSubComps = \bigcup_{\forall i \in concGroup} taskCompleteSubComponents(i),\\
&\quad postTasks = \{postTask : \forall postTask \in tasks,\\
&\qquad concComps \cap taskCompleteSubComponents(postTask) \neq \emptyset\}\\
&\quad preTasks = \{preTask : \forall preTask \in tasks,\\
&\qquad componentsUsed(preTask) \cap concSubComps \neq \emptyset\},\\
&\quad (\forall postTask \in postTasks)\\
&\quad (\forall predTask \in preTasks)\\
&\quad pred(postTask) \neq preTask
\end{aligned}
\tag{A.8}
$$

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
Tool Manuals