Task scheduling of multiple industrial robots through constraint programming

Tommy Kvant
ada09tkv@student.lu.se

January 14, 2015

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

Supervisor: Jacek Malec, jacek.malek@cs.lth.se & 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

1.2.1 The setup

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).[2]

[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.[7]

[Garey] shows that job shop problems for size $m \geq 2$ and $n \geq 3$ are NP-complete [4]

[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.[9]

[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

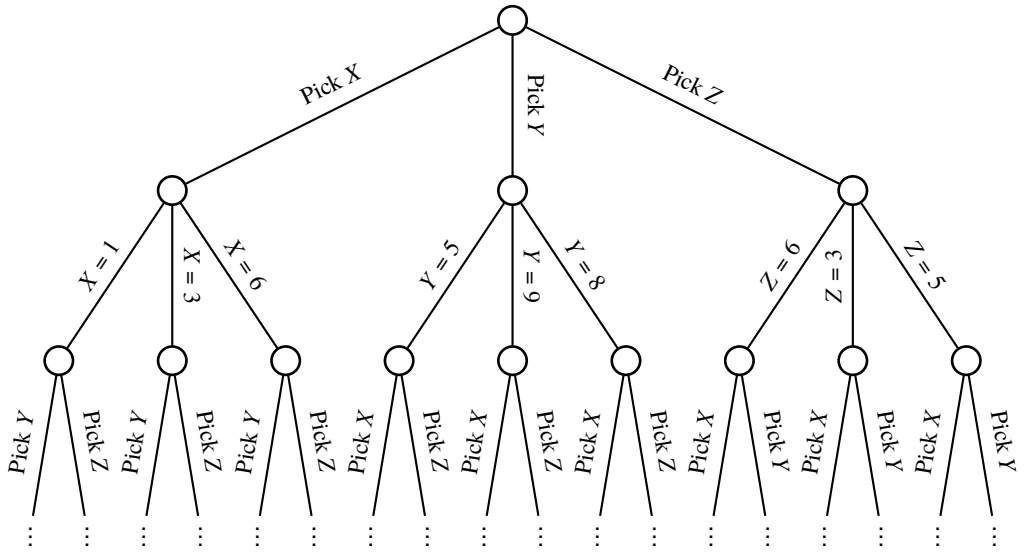

Figure 2.1: The beginning of the search space for the variables X, Y, Z, where $X = \{1, 3, 6\}$ $Y = \{5, 9, 8\}$ $Z = \{6, 3, 5\}$

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

AllDifferent

Circuit

Cumulative

Global_Cardinality

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.

[8] [5] [6]

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

2.4.2 JaCoP

2.4.3 Gecode

2.4.4 OR-tools

2.4.5 Opturion CPX

2.4.6 Choco3

Chapter 3
Model

The most important parts of the model

This model is based on/inspired by the model in [ejenstam]. 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.

Model

Variables

The solver takes a description of the robot cell in the form of a MiniZink data file. The file describes; the number of arms available, the tools available, the trays available, the fixtures available, etc. Here on after called *Cell Variables*. It also sets up a number of *Decision Varibles* which contains a set of values from which each *Decision Variable* can take.

3.0.7 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*.

Chapter 4
Assembly

Describing our assembly

Chapter 5
Evaluation

The evaluation of the model

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.[3] [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.[9] Kan vara värt att titta vidare på

Bibliography

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

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

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

[4] 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.

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

[6] The G12 Team. *Specification of Zinc and MiniZinc v.1.6*. NICTA, Victoria Research Lab, Melbourne, Australia, August 2011.

[7] 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.

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

[9] 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
Model

Beskriv grunder i hur modellen fungerar There are moves put inbetween each task Model deosn't deal with coordinates only times for movement between tasks The data used by the model can be generated using tools described in [appendix ??] The move time depends on if there is a change between the tasks, if there is a task, the change will be longer than usual.
Terminology:
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.

A.0.1 Model varibles

A.0.2 Variables

Static variables

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

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

$$tasks = \{1, \ldots, nbrTasks\} \tag{A.2}$$

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

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 above the tasks to be scheduled. As seen later, this is a convenient way of numbering these tasks.

$$startTasks = \{nbrTasks + 1, \ldots, nbrTasks + nbrMachines\} \tag{A.3}$$

As sources, the start tasks are the predecessors to the first tasks for each machine.

$$goalTasks = \{nbrTasks + nbrMachines + 1, \ldots, nbrTasks + nbrMachines \times 2\} \tag{A.4}$$

As sinks, the goal tasks has the last task as predecessor for each arm.

$$allTasks = tasks \cup startTasks \cup goalTasks \tag{A.5}$$

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

$$nbrMachines \in \{1, \ldots, 2^{32} - 1\} \tag{A.6}$$

$$machines = \{1, \ldots, nbrMachines\} \tag{A.7}$$

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

$$nbrTools \in \{1, \ldots, 2^{32} - 1\} \tag{A.8}$$

$$tools = \{1, \ldots, nbrTools\} \tag{A.9}$$

$$toolNeeded(t) \in tools,\ t \in tasks \tag{A.10}$$

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.

$$nbrComponents \in \{1, \ldots, 2^{32} - 1\} \tag{A.11}$$

$$components = \{1, \ldots, nbrComponents\} \tag{A.12}$$

$$componentsUsed(t) \subset components,\ t \in tasks \tag{A.13}$$

$nbrComponents$ 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.

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.

$$taskSubComponents(t) \subset components,\ t \in tasks \tag{A.14}$$

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.

$$taskCompleteSubComponents(t) \subset components,\ t \in tasks \tag{A.15}$$

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

$$subComponents(c) \subset components,\ c \in components \tag{A.16}$$

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.

$$nbrTrays \in \{1, \ldots, 2^{32} - 1\} \tag{A.17}$$

$$trays = \{1, \ldots, nbrTrays\} \tag{A.18}$$

$$tray(t) \in trays \cup \{0\},\ t \in tasks \tag{A.19}$$

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, $tray(t) = 0$.

$$nbrFixtures \in \{1, \ldots, 2^{32} - 1\} \tag{A.20}$$

$$fixtures = \{1, \ldots, nbrFixtures\} \tag{A.21}$$

$$fixture(t) \in fixtures \cup \{0\},\ t \in tasks \tag{A.22}$$

$fixtures$ 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, $fixture(t) = 0$

$$nbrOutputs \in \{1, \ldots, 2^{32} - 1\} \tag{A.23}$$

$$outputs = \{1, \ldots, nbrOutputs\} \tag{A.24}$$

$$output(t) \in outputs \cup \{0\},\ t \in tasks \tag{A.25}$$

$outputs$ 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, $output(t) = 0$.

$$nbrConcurrentGroups \in \{1, \ldots, 2^{32} - 1\} \tag{A.26}$$

$$concurrentGroups = \{1, \ldots, nbrConcurrentGroups\} \tag{A.27}$$

$$concurrentTasks(k) \subset tasks,\ k \in concurrentGroups \tag{A.28}$$

$concurrentTasks(k)$ 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.

$$nbrOrderedGroups \in \{1, \ldots, 2^{32} - 1\} \tag{A.29}$$

$$orderedGroups = \{1, \ldots, nbrOrderedGroups\} \tag{A.30}$$

$$orderedGroup(k) \subset tasks,\ k \in orderedGroups \tag{A.31}$$

$$ordered(k, i) \in tasks,\ i \in \{1, \ldots, |orderedGroup(k)|\},\ k \in orderedGroups \tag{A.32}$$

$$orderedSet = \bigcup_{\forall k \in orderedGroups} order(k),\ orderedSet \subset tasks \tag{A.33}$$

$orderedGroup(k)$ 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?], 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.

$$mounting \subset tasks \tag{A.34}$$

$$taking \subset tasks \tag{A.35}$$

$$moving \subset tasks \tag{A.36}$$

$$putting \subset tasks \tag{A.37}$$

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.

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.

$$putting(c) \subset putting,\ c \in components \tag{A.38}$$

$$mounting(c) \subset mounting,\ c \in components \tag{A.39}$$

$$taking(c) \subset taking,\ c \in components \tag{A.40}$$

$$moving(c) \subset moving,\ c \in components \tag{A.41}$$

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

$$duration(t) \in \{0, \ldots, 2^{32} - 1\},\ t \in tasks \tag{A.42}$$

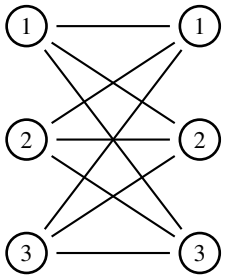

Figure A.1: All the transitions between the tool states

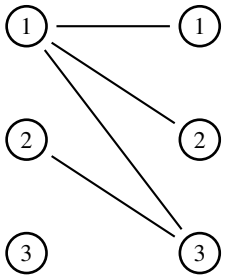

Figure A.2: All the transitions between the tool states

$duration(t)$ 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, $timeMatrix3D$.[picture?] This is a 3d matrix and 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.

$$timeMatrixDepth = \frac{n^2 - n + 2}{2},\ n = nbrTools \tag{A.43}$$

$timeMatrixDepth$ 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 A.1, 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 4 transitions, see figure A.2

$$timeMatrix3D(t(from), t(to), k) \in \{0, \ldots, 2^{32} - 1\},\ t(from) \in tasks \cup startTasks,\ t(to) \in tasks,\ k \in \{0, \ldots, timeMatrixDepth\} \tag{A.44}$$

The time to move from task $t(from)$ to task $t(to)$ changing tool according to k

$$taskOutOfRange(m) \subset tasks,\ m \in machines \tag{A.45}$$

The tasks that cannot be reached by machine m

Decision variables

$$usingMachine(t) \in machines,\ t \in tasks \tag{A.46}$$

The machine task t uses

$$pred(t) \in allTasks,\ t \in allTasks \tag{A.47}$$

The predecessor task of task t

$$\begin{aligned} maxE =& (max(\{duration(t) : t \in tasks\}) + \\ & max(\{timeMatrix3D(t_1, t_2, k) : \forall t_1 \in tasks \cup startTasks, \\ & \forall t_2 \in tasks,\ \forall k \in \{0, \ldots, timeMatrixDepth\}\}) \times nbrTasks \end{aligned} \tag{A.48}$$

Rough upper limit of the total schedule time. Assumes all move times take as long as the longest move time existing in the schedule. And likewise for the task durations.

$$start(t) \in \{0, \ldots, maxE\},\ t \in allTasks \tag{A.49}$$

The start time for task t

$$end(t) = start(t) + duration(t),\ t \in allTasks \tag{A.50}$$

The end time for task t

$$makespan \in \{0, \ldots, maxE\} \tag{A.51}$$

The makespan for the whole schedule, the time to minimize

$$moveDuration(t) \in \{0, \ldots, maxE\},\ t \in allTasks \tag{A.52}$$

The duration of the move to task t from its predecessor

$$moveStart(t) \in \{0, \ldots, maxE\},\ t \in allTasks \tag{A.53}$$

The start time for the move to task t from its predecessor

$$moveEnd(t) = moveStart(t) + moveDuration(t),\ t \in allTasks \tag{A.54}$$

The end time for the move to task t from its predecessor

$$toolUsed(t) \in tools,\ t \in allTasks \tag{A.55}$$

The tool used at task t

A.1 Filter

A.1.1 Domain filter

$$
\begin{aligned}
&(\forall t \in tasks) \\
&maxMoveDurs(t) = max(\{timeMatrix3D(t, j, k) : \\
&\qquad\qquad \forall j \in tasks, \\
&\qquad\qquad \forall k \in \{1, \ldots, timeMatrixDepth\}, \\
&\qquad\qquad j \neq t\})
\end{aligned}
\tag{A.56}
$$

The maximum duration for a move to task t

$$
\begin{aligned}
&(\forall t \in tasks) \\
&minMoveDurs(t) = min(\{timeMatrix3D(t, j, k) : \\
&\qquad\qquad \forall j \in tasks, \\
&\qquad\qquad \forall k \in \{1, \ldots, timeMatrixDepth\}, \\
&\qquad\qquad j \neq t\})
\end{aligned}
\tag{A.57}
$$

The minimum duration for a move to task t

$$
maxEnd = \sum_{\forall t \in tasks} duration(t) + \sum_{\forall t \in tasks} maxMoveDurs(t) \tag{A.58}
$$

The upper limit of the schedule; all tasks is laid out after one after another and the duration between them is the maximum of the moves to them

$$
minEnd = \frac{(\sum_{\forall t \in tasks} duration(t) + \sum_{\forall t \in tasks} minMoveDurs(t))}{nbrMachines} \tag{A.59}
$$

The lower limit of the schedule; the total duration of each task is the duration of the task itself and the minimum duration of a move to the task, and the tasks are scheduled perfectly over all the machines

$$
\begin{aligned}
&(\forall t \in allTasks) \\
&start(t) \leq maxEnd - duration(t) \wedge \\
&end(t) \leq maxEnd
\end{aligned}
\tag{A.60}
$$

Sets the upper limit for the start of each task to be the maximum end minus the duration for the task. Sets the end for each task to be the maximum end

$$
(\forall t \in tasks)\ end(t) \geq duration(t) + minMoveDurs(t) \tag{A.61}
$$

A task can start at its earliest at the time directly after the move to a task, therefore the end of a task can earliest happen after the duration of the task plus the shortest move to it

$$
(\forall t \in tasks)\ moveStart(t) \leq maxEnd - (duration(t) + minMoveDurs(t) \tag{A.62}
$$

A move to a task can start at the latest *maxEnd* but before the duration of the task and before at least the minimum of the move times to the task

$$
\begin{aligned}
&(\forall t \in tasks) \\
&moveDuration(t) \leq maxMoveDurs(t) \wedge \\
&moveDuration(t) \geq minMoveDurs(t)
\end{aligned}
\tag{A.63}
$$

The move duration for task t is limited by $maxMoveDurs$ and $minMoveDurs$

$$
\begin{aligned}
&(\forall t \in tasks) \\
&moveEnd(t) \leq maxEnd - duration(t) \wedge \\
&moveEnd(t) \geq minMoveDurs(t)
\end{aligned}
\tag{A.64}
$$

The end of a move to a task can at the latest come at $maxEnd$ minus the duration of the task. The move to a task can at the earliest happen at time 0, so the end can earliest happen at the shortest move time to the task

$$
\begin{aligned}
makeSpan \leq maxEnd \wedge \\
makespan \geq minEnd
\end{aligned}
\tag{A.65}
$$

Limits the makespan

$$
\begin{aligned}
&(\forall t \in tasks) \\
&(\forall i \in \{0, \ldots, maxMoveDurs(t)\} / \{timeMatrix3D(task, j, k) : \\
&\qquad\qquad \forall j \in tasks, \\
&\qquad\qquad \forall k \in \{1, \ldots, timeMatrixDepth\} \\
&\qquad\qquad t \neq j\}) \\
&moveDuration(t) \neq i,
\end{aligned}
\tag{A.66}
$$

Limits the $moveDuration$ domains to only the values specified in the $timeMatrix3D$

$$
\begin{aligned}
&(\forall t \in tasks/taking) \\
&moveStart(t) \geq min(\{duration(tt) + minMoveDurs(tt) : \\
&\qquad\qquad \forall tt \in taking\})
\end{aligned}
\tag{A.67}
$$

As the schedule has to start with a take task, the move to the other tasks can only start as early as after the shortest move to and execution of one of the take tasks

$$
\begin{aligned}
&(\forall t \in tasks) \\
&prevTasks = \{task : \forall task \in tasks, \\
&\qquad\qquad componentCreated(task) \in componentsUsed(t)\}, \\
&nbrMachines \geq |prevTasks|, \\
&\qquad 0 < |prevTasks|, \\
&start(t) \geq max(\{duration(pt) + minMoveDurs(pt) : \forall pt \in prevTasks\})
\end{aligned}
\tag{A.68}
$$

$prevTasks$ are the tasks for which the task t uses the component created at task $task$, hence the tasks in $prevTasks$ precedes task t. If the number of machines are greater than or equal

to the number of task preceding task t, then the best scheduling is to do all tasks in parallel. If so the earliest task t can start is greater or equal to the maximum of the preceding tasks

$$\begin{aligned}
&(\forall t \in tasks) \\
&prevTasks = \{task : \forall task \in tasks, \\
&\qquad componentCreated(task) \in componentsUsed(t)\}, \\
&nbrMachines < |prevTasks|, \\
&start(t) \geq \frac{\left(\sum_{\forall pt \in prevTasks} duration(pt) + minMoveDurs(pt)\right)}{nbrMachines}
\end{aligned} \tag{A.69}$$

$prevTasks$ are the tasks for which the task t uses the component created at task $task$, hence the tasks in $prevTasks$ precedes task t. If the number of machines are less than the number of tasks preceding task t, then the best we can do is to divide the task times equally on all machines. If the tasks can be divided onto the machines so that the total length of the times on all machines are the same, that time will be equal to the sum/$nbrMachines$. If they don't match up the maximum of these times will be larger than the sum/$nbrMachines$.

$$\begin{aligned}
&(\forall t \in tasks) \\
&succTasks = \{task : \forall task \in tasks, \\
&componentsUsed(t) \subset taskCompleteSubComponent(task), \\
&componentsUsed(t) \cup taskCompleteSubComponents(task) \neq \emptyset\}, \\
&nbrMachines \geq |succTasks|, \\
&\qquad 0 < |succTasks|, \\
&end(t) \leq maxEnd - max(\{duration(st) + minMoveDurs(st) : \\
&\qquad \forall st \in succTasks\})
\end{aligned} \tag{A.70}$$

$succTasks$ are the tasks that has the components used in task t as subcomponents, hence the tasks in $succTasks$ succeeds task t. If the number of machines are greater than or equal to the number of task preceding task t, then the best scheduling is to do all tasks in parallel. If so the latest task t can end is less than or equal to the maximum end of the schedule minus the longest of the succeeding tasks

$$\begin{aligned}
&(\forall t \in tasks) \\
&succTasks = \{task : \forall task \in tasks, \\
&componentsUsed(t) \subset taskCompleteSubComponent(task), \\
&componentsUsed(t) \cup taskCompleteSubComponents(task) \neq \emptyset\}, \\
&nbrMachines \leq |succTasks|, \\
&end(t) \leq maxEnd - \frac{(\sum_{\forall st \in succTasks} duration(st) + minMoveDurs(st))}{nbrMachines}
\end{aligned} \tag{A.71}$$

$succTasks$ are the tasks that has the components used in task t as subcomponents, hence the tasks in $succTasks$ succeeds task t. If the number of machines are less than the number

of tasks preceding task t, then the best we can do is to divide the task times equally on all machines. If the tasks can be divided onto the machines so that the total length of the times on all machines are the same, that time will be equal to the sum/$nbrMachines$. If they don't match up the maximum of these times will be larger than the sum/$nbrMachines$.

A.1.2 Predecessor filter

$$alldifferent(\{pred(t) : \forall t \in tasks\}) \tag{A.72}$$

Helps ensure that no two tasks can have the same predecessor

$$(\forall t1, \forall t2 \in taking)\ pred(t1) \neq t2 \tag{A.73}$$

No two taking tasks can be the predecessor of each other

$$(\forall t1, \forall t2 \in taking)\ pred(t1) \neq t2 \tag{A.74}$$

No two putting tasks can be the predecessor of each other

$$(\forall t1, \forall t2 \in mounting)\ pred(t1) \neq t2 \tag{A.75}$$

No two mounting tasks can be the predecessor of each other

$$\begin{aligned}
&(\forall t \in tasks) \\
&nonPredecessors = \{t_2 : \forall t_2 \in tasks, \\
&componentsUsed(t) \subset taskCompleteSubComponents(t_2) \vee \\
&componentsUsed(t) \subset subComponents(componentCreated(t_2))\} \\
&(\forall nonPred \in nonPredecessors) \\
&pred(t) \neq nonPred,
\end{aligned} \tag{A.76}$$

A task t cannot have task t_2 as predecessor if task t_2 uses a component, or creates a component, that the component task t uses has as a subcomponent

$$\begin{aligned}
&(\forall startTask \in startTasks) \\
&(\forall putTask \in putting) \\
&pred(putTask) \neq startTask
\end{aligned} \tag{A.77}$$

Since a component has to be taken before it can be put anywhere, put tasks cannot be first in the schedule

$$\begin{aligned}
&(\forall startTask \in startTasks) \\
&(\forall mountTask \in mounting) \\
&pred(putTask) \neq startTask
\end{aligned} \tag{A.78}$$

Since a component has to be taken before it can be mounted anywhere, mount tasks cannot be first in the schedule

$$\begin{aligned}&(\forall goalTask \in goalTasks)\\&(\forall takeTask \in taking)\\&pred(goalTask) \neq takeTask\end{aligned} \tag{A.79}$$

Since a schedule has to end with an assembly on the output, a take task cannot be at the end of the assembly

$$\begin{aligned}counts &= \{i : \forall task \in outputTasks,\ i \in \{0, \ldots, 1\}\},\\outputTasks &= \{task : \forall task \in tasks,\ output(task) > 0\},\\goalPreds &= \{pred(task) : \forall task \in goalTasks\},\\&global\_cardinality(goalPreds,\ outputTasks,\ counts) \wedge\\&\sum counts > 0\end{aligned} \tag{A.80}$$

At least one of the output tasks has to be last on one of the circuits

$$\begin{aligned}counts &= \{i : \forall task \in startTasks,\ i \in \{0, \ldots, 1\}\},\\takePreds &= \{pred(task) : \forall task \in taking,\ output(task) = 0\},\\&global\_cardinality(takePreds,\ startTasks,\ counts) \wedge\\&\sum counts > 0\end{aligned} \tag{A.81}$$

At least one of the take tasks, thats not on an output, has to be first on one of the circuits

$$\begin{aligned}&(\forall comp \in components)\\&(\forall mountTask \in mounting(comp))\\&(\forall putTask \in putting(comp))\\&pred(putTask) \neq mountTask\end{aligned} \tag{A.82}$$

If a set of tasks on a component involves a mount task and a put task, the predecessor of the put task cannot be the mount task

$$\begin{aligned}&(\forall comp \in components)\\&(\forall mountTask \in mounting(comp))\\&(\forall takeTask \in taking(comp))\\&pred(takeTask) \neq mountTask\end{aligned} \tag{A.83}$$

If a set of tasks on a component involves a mount task and a take task, the predecessor of the take task cannot be the mount task.

$$\begin{aligned}&(\forall comp \in components)\\&(\forall putTask \in puttingcomp,\ tray(putTask) > 0)\\&(\forall takeTask \in taking(comp),\ tray(putTask) = tray(takeTask))\\&pred(putTask) \leq takeTask\end{aligned} \tag{A.84}$$

If a component has a put and take performed on it in a tray, the predecessor of the put task cannot be the take task.

$$
\begin{aligned}
&(\forall f \in fixtures) \\
&(\forall putTask \in puttingcomp,\ fixture(putTask) = f) \\
&(\forall takeTask \in taking(comp),\ fixture(takeTask) = f, \\
&\qquad componentsUsed(putTask) \subset taskSubComponents(takeTask)) \\
&pred(putTask) \leq takeTask
\end{aligned}
\tag{A.85}
$$

For every put action on a fixture, there is a take action. The predecessor of the put task cannot be the take task.

$$
\begin{aligned}
&(\forall group \in \{1, \ldots, nbrConcurrentGroups\}) \\
&\quad (\forall t_1 \in concurrentTasks(group)) \\
&\quad (\forall t_2 \in concurrentTasks(group)/\{t_1\}) \\
&pred(t_1) \neq t_2 \wedge pred(t_2) \neq t_1
\end{aligned}
\tag{A.86}
$$

Concurrent tasks cannot be predecessor to each other.

$$
\begin{aligned}
&(\forall t_1 \in tasks,\ componentCreated(t_1) > 0) \\
&(\forall t_2 \in tasks,\ componentCreated(t_1) \in compinentUsed(t_2)) \\
&pred(t_1) \neq t_2
\end{aligned}
\tag{A.87}
$$

Components cannot be used before they are created.

$$
\begin{aligned}
&(\forall precTask \in tasks) \\
&(\forall t \in tasks,\ precTask \neq t, \\
&componentUsed(precTask) \cup taskCompleteSubComponent(t) \subset \\
&\qquad taskCompleteSubComponents(t), \\
&componentsUsed(precTask) \cup taskCompleteSubComponents(t) \neq \emptyset \\
&pred(precTask) \neq t
\end{aligned}
\tag{A.88}
$$

Task using a component cannot execute before all the tasks having it as subcomponent.

$$
\begin{aligned}
&(\forall concGroup \in concurrentTasks,\ |concGroup| = nbrMachines) \\
&concComps = \bigcup_{\forall i \in concGroup} componentsUsed(i), \\
&concSubComps = \bigcup_{\forall i \in concGroup} taskCompleteSubComponents(i), \\
&preTasks = \{preTask : \forall preTask \in tasks, \\
&\qquad componentsUsed(preTask) \cap concSubComps \neq \emptyset\}, \\
&(\forall postTask \in postTasks) \\
&(\forall predTask \in preTasks) \\
&pred(postTask) \neq preTask
\end{aligned}
\tag{A.89}
$$

If there is a set of concurrent tasks on a subset of tasks using as many machines as available, the tasks after the concurrent tasks cannot have the tasks before the concurrent tasks as predecessors.

A.2 Constraints

$$(\forall t \in tasks)\ end(t) \leq makespan \tag{A.90}$$

All ends has to be lesser than the total end

$$(\forall t \in startTasks \cup goalTasks)\ start(t) = 0 \tag{A.91}$$

Start and goal tasks are not temporal tasks, i.e. they are timeless. Therefore, their start time is set to 0

$$\begin{aligned}(\forall m \in machines)& \\ usingMachine(nbrTasks + m) = m \wedge& \\ usingMachine(nbrTasks + nbrMachines + m) = m&\end{aligned} \tag{A.92}$$

The start tasks and goal tasks are assigned to machines, thereby there are start and goal tasks assigned to every machine. Because of the way start and goal tasks are created, the start tasks starts with number $nbrTasks + 1$, and the corresponding goal task for a start task can be accessed by $startTask + nbrMachines$.

$$\begin{aligned}(\forall m \in machines)& \\ (\forall t \in tasksOutOfRange(m))& \\ usingMachine(t) \neq m&\end{aligned} \tag{A.93}$$

Setting the tasks that are out of range for each machine

A.2.1 Precedences

$$\begin{aligned}(\forall comp \in components)& \\ (\forall mountTask \in mounting(comp))& \\ (\forall putTask \in putting(comp))& \\ end(putTask) \leq moveStart(mountTask)&\end{aligned} \tag{A.94}$$

If a set of tasks on a component involves a mount task and a put task, the put task has to come before the mount task

$$\begin{aligned}\forall comp \in components& \\ \forall mountTask \in mounting(comp),& \\ \forall takeTask \in taking(comp),& \\ end(takeTask) \leq moveStart(mountTask)&\end{aligned} \tag{A.95}$$

If a set of tasks on a component involves a mount task and a take task, the take task has to come before the mount task

$$\begin{aligned}\forall comp \in components& \\ (\forall putTask \in putting(comp),\ tray(putTask) > 0)& \\ (\forall takeTask \in taking(comp), tray(putTask) = tray(takeTask))& \\ end(putTask) \leq moveStart(takeTask)&\end{aligned} \tag{A.96}$$

If a component has a put and take performed on it in a tray, the put has to come before the take.

$$\begin{aligned}
&(\forall f \in fixtures) \\
&(\forall putTask \in putting(comp),\ fixture(putTask) = f) \\
&(\forall takeTask \in taking(comp),\ fixture(takeTask) = f \wedge \\
&\qquad componentsUsed(putTask) \subset taskSubComponents(takeTask)) \\
&end(putTask) \leq moveStart(takeTask),
\end{aligned} \tag{A.97}$$

For every put action on a fixture, there is a take action. The put action has to come before the take action.

$$\begin{aligned}
&(\forall f \in fixtures) \\
&puts = [put : \forall put \in putting,\ fixture(put) = f], \\
&takesForEachPut = [\{take : \forall take \in taking,\ fixture(take) = f, \\
&componentsUsed(put) \subset taskCompleteSubComponent(take)\} : \forall put \in puts], \\
&takes = [\underset{\forall take \in takesForEachPut(p)}{\arg\min} taskCompleteSubComponent(take) : \\
&\qquad \forall p \in \{1, \ldots, |puts|\}], \\
&cumulative([moveStart(task) : \forall task \in puts], \\
&\qquad [abs(end(takes(i)) - moveStart(puts(i))) : \forall i \in \{1, \ldots, |puts|\}], \\
&\qquad [1 : \forall i \in \{1, \ldots, |puts|\}], \\
&\qquad 1)
\end{aligned} \tag{A.98}$$

The intervals between when components are put and then taken again cannot overlap on the same fixture.

$$\begin{aligned}
&(\forall group \in \{1, \ldots, nbrConcurrentGroups\}) \\
&\quad (\forall t_1 \in concurrentTasks(group)) \\
&\quad (\forall t_2 \in concurrentTasks(group)/\{t_1\}) \\
&start(t_1) = start(t_2) \wedge \\
&usingMahine(t_1) \neq usingMachine(t_2),
\end{aligned} \tag{A.99}$$

Concurrent tasks has to happen at the same time.

$$\begin{aligned}
&(\forall t_1 \in tasks,\ componentCreated(t1) > 0) \\
&(\forall t_2 \in tasks,\ componentCreated(t_1) \in compinentUsed(t_2)) \\
&moveStart(t_2) \geq end(t_1)
\end{aligned} \tag{A.100}$$

Components cannot be used before they are created.

$$
\begin{aligned}
&(\forall precTask \in tasks) \\
&\quad (\forall t \in tasks,\ precTask \neq t, \\
&\qquad componentUsed(precTask) \cup taskCompleteSubComponent(t) \subset \\
&\qquad taskCompleteSubComponents(t), \\
&\qquad componentsUsed(precTask) \cup taskCompleteSubComponents(t) \neq \emptyset) \\
&end(precTask) \leq moveStart(t),
\end{aligned}
\tag{A.101}
$$

Task using a component cannot execute before all the tasks having it as subcomponent.

$$
\begin{aligned}
&(\forall f \in fixtures) \\
&fixtureTasks = [t : \forall t \in tasks,\ fixture(t) = f], \\
&cumulative([start(t) : \forall t \in fixtureTasks], \\
&\qquad [duration(t) : \forall t \in fixtureTasks], \\
&\qquad [1 : t \in fixtureTasks], \\
&\qquad 1)
\end{aligned}
\tag{A.102}
$$

Tasks on the same fixture cannot overlap.

$$
\begin{aligned}
&(\forall tr \in trays) \\
&trayTasks = [t : \forall t \in tasks,\ tray(t) = tr], \\
&cumulative([start(t) : \forall t \in trayTasks], \\
&\qquad [duration(t) : \forall t \in trayTasks], \\
&\qquad [1 : t \in trayTasks], \\
&\qquad 1)
\end{aligned}
\tag{A.103}
$$

Tasks on the same tray cannot overlap.

$$
\begin{aligned}
&(\forall o \in outputs) \\
&outputTasks = [t : \forall t \in tasks,\ output(t) = o], \\
&cumulative([start(t) : \forall t \in outputTasks], \\
&\qquad [duration(t) : \forall t \in outputTasks], \\
&\qquad [1 : t \in outputTasks], \\
&\qquad 1)
\end{aligned}
\tag{A.104}
$$

Tasks on the same output cannot overlap.

$$
(\forall t \in tasks)\ Start(t) \geq moveEnd(t) \tag{A.105}
$$

A task can only start after the move to it.

A.2.2 Predecessors

$$
(\forall t \in tasks)\ moveStart(t) \geq end(pred(t)) \tag{A.106}
$$

A task has to start after its predecessor.

$$\begin{aligned}(\forall startTask \in startTasks/\{nbrTasks+1\})\\ pred(startTask) = startTask + nbrMachines - 1\end{aligned} \tag{A.107}$$

In order to create a circuit containing the sub circuits, for all start tasks, except the first one, the start tasks predecessor is the previous goal task.

$$pred(nbrTasks+1) = nbrTasks + nbrMachines \times 2 \tag{A.108}$$

To complete the circuit, the first start tasks predecessor is the last goal task.

$$circuit(\{pred(t) : \forall t \in tasks\}) \tag{A.109}$$

The predecessors has to form a circuit.

$$\begin{aligned}&(\forall c \in components)\\ &(\forall mountTask \in mounting(c))\\ &\quad puts = \{p : \forall p \in putting(c),\\ &\qquad (fixture(p) > 0 \wedge fixture(p) = fixture(mountTask)) \vee\\ &\qquad (output(p) > 0 \wedge output(p) = output(mountTask)) \vee\\ &\qquad (tray(p) > 0 \wedge tray(p) = tray(mountTask))\},\\ &(\forall takeTask \in taking(c),\ takeTask \notin orderedSet,\ puts = \emptyset)\\ &pred(mountTask) = takeTask\end{aligned} \tag{A.110}$$

If a set of tasks on a component involves a mount and a take task, but no move tasks or put task on the same fixture, tray or output as the mount, the take task is the predecessor of the mount task.

$$\begin{aligned}(\forall c \in components,\ moving(c) = \emptyset)\\ (\forall putTask \in putting(c),\ tray(putTask) = 0)\\ (\forall takeTask \in taking(c))\\ pred(putTask) = takeTask\end{aligned} \tag{A.111}$$

If a set of tasks on a component involves a put task not in a tray and a take task, and there is no moves involved, the take task has to be the predecessor of the put task.

$$\begin{aligned}&(\forall k \in orderedGroups)\\ &(\forall i \in \{1,\ldots,|orderedGroup(k)|-1\})\\ &pred(ordered(k,i+1)) = ordered(k,i)\end{aligned} \tag{A.112}$$

Sets up the predecessors in accordance with the ordered groups.

$$\begin{aligned}(\forall t \in tasks \cup goalTasks)\\ usingMachine(t) = usingMachine(pred(t))\end{aligned} \tag{A.113}$$

A task has to use the same machine as its predecessor.

$$\begin{aligned}(\forall t \in tasks)\\ k = abs(toolUsed(t) - toolUsed(pred(t))) + 1,\\ moveDuration(t) = timeMatrix3D(pred(t),\ t,\ k)\end{aligned} \tag{A.114}$$

Take tasks has to use the same tool as its predecessor or do a change first.

$$(\forall t \in tasks,\ toolNeeded(t) \neq 0)\ toolUsed(t) = toolNeeded(t) \tag{A.115}$$

Set the tool used for each task in accordance with $toolNeeded$.

Appendix B
Tool Manuals