CHALMERS

Automatic Generation of Operations for the FLEXA Production System

Master of Science Thesis in Automation

NINA SUNDSTRÖM

Department of Signals and Systems
Division of Automatic Control, Automation and Mechatronics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2010

EX024/2010

Master of Science Thesis in Automation
REPORT NO. EX024/2010

Automatic Generation of Operations for the FLEXA Production System

NINA SUNDSTRÖM

Department of Signals and Systems
Division of Automatic Control, Automation and Mechatronics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2010

Automatic Generation of Operations for the FLEXA Production System
NINA SUNDSTRÖM

Report No. EX024/2010

Department of Signals and Systems
Division of Automatic Control, Automation and Mechatronics
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31-772 1000

Göteborg, Sweden 2010

Abstract

The scope of this thesis was to introduce the possibility to make specifications of a process inside a simulation tool and automatically generate this information to a formal verification tool. The intention was to specify possible locations for different movable resources in a manufacturing system. The positions were to be transformed to transport operations describing the permitted motion for each movable resource in the production cell.

Introducing specifications at an early stage and in a familiar tool would raise the level of abstraction, since the information otherwise would be modeled using formal tools developed by the academic world. In order for this approach to be successful, it had to be possible to retrieve locations from the simulation tool. If so, the locations had to be parsed into transport operations in a format readable by a verification tool.

FLEXA is a research project within the European Seventh Framework Programme. The objective is to utilize automation technology developed by the automotive industry so that the European aero engine industry can meet requirements on flexibility, cost effectiveness and safety aspects. For this thesis, a simplified version of the FLEXA production cell was used in a simulation tool called Process Simulate where a plug-in was integrated using C#. A GUI was developed to guide the user through a few steps that involved creating possible locations and picking locations for specific resources. Transport operations were automatically generated to an .xml file.

The results show that it was possible to interact with the simulation tool, create locations for different resources and generate information in the form of transport operations. The .xml file was written in a .wmod format which was readable by Supremica, a formal verification tool developed at Chalmers University of Technology.

Keywords: Tecnomatix, Process Simulate, Product Recipe, Transport Operation, Automatic generation of operations

Acknowledgements

I would like to express my deepest thanks to my examiner Petter Falkman for his guidance and support throughout this thesis work. I would also like to express my appreciation to Patrik Magnusson for his continuous support, valuable advice and discussions.

Thanks to Fredrik Westman for introducing me to Process Simulate and for answering any questions that I faced with the software during this project.

Sincere thanks to my office mates, Oskar Wigström and Roozbeh Kianfar, for always having a good spirit and the time for a short chat or a cup of coffee.

Nina Sundström
Göteborg, 2010

Contents

Notations

.NET A framework developed by Microsoft. It includes a *Common Language Runtime*(CLR) that manages the execution of programs written in this framework, and a large *Framework Class Library*(FCL) with pre-built code.

action If a guard condition is true and the event occurs, actions in updating the variables may follow.

API Application Programming Interface implemented by a software program enabling the program to interact with other software.

C# Object-oriented programming language developed by Microsoft.

carrier A resource which task is to hold a product, e.g. fixture or basket.

EFA Extended Finite Automata which is an augmentation of an ordinary automaton, where transitions are associated with guards and actions.

event Represents an incident that causes an automaton to move from one state to another.

guard A set of conditions that has to be fulfilled for an event to be enabled.

GUI Graphical User Interface to a computer which makes it possible to interact with a program in more ways than just the textbased.

plug-in A set of software that makes it possible to add new features to a larger software application.

Process Simulate The process simulation application in the Tecnomatix environment.

product recipe The sequence of operations that corresponds to a finished product.

Sequence Planner Modeling and analyzing tool for sequences of a process.

Supremica A formal verification tool developed at Chalmers University of Technology.

Tecnomatix An application for e.g. part manufacturing, resource planning and plant simulation owned by Siemens Product Lifecycle Management Software Inc.

transport operation Description of each carrier's permitted movement in a production system.

XML Widely used programming format. It stands for eXtensible Markup Language.

1 Introduction

Today manufacturers face challenges in delivering products with high quality, competitive in cost and satisfying performance whilst not compromising safety aspects. In order to balance between these requirements, manufacturing processes have become more complex, more automated and flexible. Consequently, virtual manufacturing has become an important part for implementing production cells and improving manufacturing processes.

For the aerospace manufacturing industry, requirements on the development in production has been introduced from demands on new product introduction, new materials used and new regulations on environment. The FLEXA (advanced FLEXible Automation cell) production system is a part of a research project within the European Seventh Framework Programme (FP7). The aim of the FLEXA project is to develop and deliver methods and tools that support the design and the development of flexible automation cells for the European aero engine industry. To meet this goal five major aero engine and component manufacturers, together with six universities and four Society of Manufacturing Engineers companies, have joined forces. Among the universities are Chalmers University of Technology and University West. [1]

This thesis will introduce the concept of product recipes and transport operations in the manufacturing process. The operations that are necessary to be performed in order to have a finished product constitute the product recipe. Transport operations describe the permitted motions for a movable resource inside a production cell. These two definitions will have a strong advantage when e.g. introducing new resources to a manufacturing system.

A simplified version of the FLEXA cell will be used as a virtual model in a simulation tool called Process Simulate [9]. This software is a part of the Tecnomatix application suites owned by Siemens Product Lifecycle Management Software Inc., provided by University West. The intention is to introduce the possibility to define different locations for a resource in the virtual environment and then automatically generate transport operations used for formal verification of the production system.

The motivation for this thesis subject is to take further use of the simulation tool. The aim is to be able to make specifications in the production cell that otherwise would be written as formal automata. By introducing specifications at an earlier stage and in a familiar tool, the level of abstraction would be raised. The goal is for the user to follow a few steps, involving picking locations in a virtual cell, to retrieve transport operations. The information can be saved to an .xml file in a format readable by a tool for formal verification, like Supremica [2]. The result of the verification of transport operations and product recipe would consist of possible routes for the movable resource to take and what routes that may cause blocking situations. With this information it would e.g. be possible to create robot paths in the simulation tool and use optimization to find the optimal path.

1.1 Project Scope

The scope of this project is to automatically create a model of operations for the FLEXA production system. By specifying possible locations for different resources in a virtual production system, transport operations should be created. These should model potential routes of resources and what conditions that have to be fulfilled in order for a transport to occur.

1.2 Purpose

The purpose of this project is to:

- Study the concept of transport operations and product recipes.
- Understand the structure of Process Simulate and how to retrieve information regarding locations for resources using the Tecnomatix .NET API.
- Create a plug-in with a GUI to automatically generate transport operations.

1.3 Aim

This project aims to extend the use of the simulation tool, introducing specifications at an early stage in the production planning process. The intention is to specify possible locations for different resources and generate transport operations between these locations. This raises the level of abstraction since the simulation software is a tool already used in industry. To achieve this aim, a virtual model in Process Simulate is used for automatically generating transport operations to an .xml file. This file can later be opened in a tool for formal verification, such as Supremica, for synthesis and optimization.

1.4 Approach

To interact with Process Simulate v9.1 the Tecnomatix.NET API will be used. Using this API it is possible to create buttons and commands in the Process Simulate environment. The commands can be written in any .NET language such as Visual Basic .NET or C#. However, a prerequisite is to use Microsoft Visual Studio .NET Professional, 2005 edition [7]. Since examples in the API are written in C# and previous work found on the subject also used C#, it is a natural choice to use that programming language. The approach can be represented by a work flow displayed in Figure 1.1. A plug-in will be programmed using

Figure 1.1: The work flow of the project.

C# which parses locations in Process Simulate into transport operations. The information will be written to an .xml file in a .wmod format which can be opened in Supremica. In order for the user to easily achieve the steps before extraction is done, a GUI will be developed. The idea is that the user follow the instructions in the GUI and after a few mouse clicks have completed the task of retrieving the information needed in order to have a model of transport operations.

1.5 Limitations

This thesis work will not investigate the possibility to generate information about product recipes from the simulation tool. Neither will the project consider robots performing the transports.

2 Background

2.1 The FLEXA Production Cell

The FLEXA project aims to create tools and methods to the European aero engine industry to design and develop flexible automation cells. The layout of the FLEXA cell, which has been proposed to be used as an example cell by Volvo Aero, is displayed in Figure 2.1. This cell will be used to manufacture a turbine exhaust case which is a part located in the

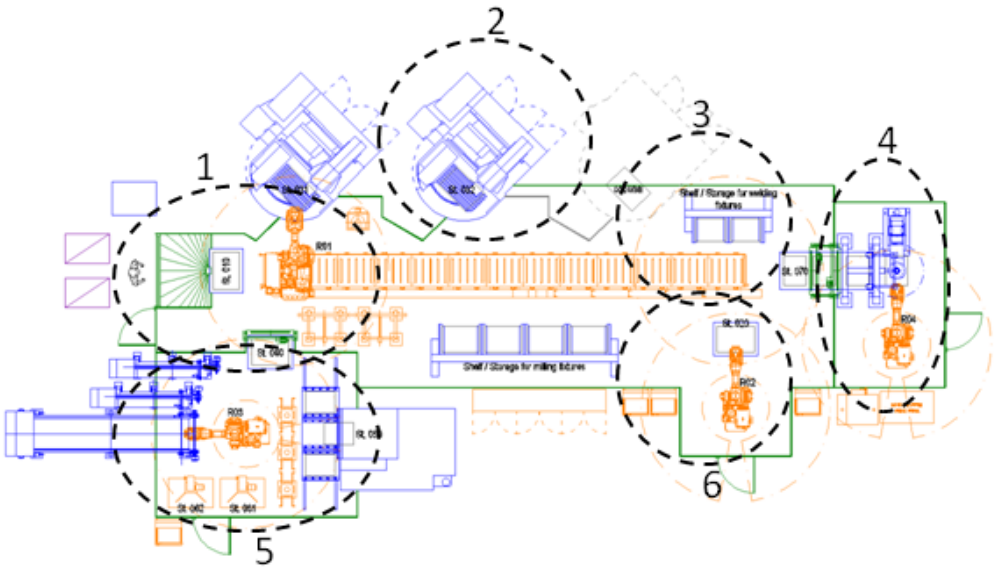


Figure 2.1: The layout of the FLEXA production cell.

exhaust area of an airplane engine. In the current production this part is delivered as one large piece and only a few number of suppliers are able to deliver it. In order to reduce cost the exhaust case will be divided into subparts that will be welded together to form the final larger part. By introducing this new procedure, more suppliers can deliver the subparts resulting in lower cost. The different zones in the cell will perform:

1. Fixturing/Defixturing
2. Measuring and Milling
3. Storage of fixtures
4. Welding
5. Washing and Grinding
6. Measuring

2.2 Product Recipes and Transport Operations

Consider that a product is to be produced in the FLEXA production cell. This product has a given product recipe, i.e. a number of operations that have to be executed in order to have a finished product. The operations could for example be: *Fixate*, *Milling* and *Unfixate*, see Figure 2.2. An operation is realized in e.g. a machine or in some other area in the production system. Products manufactured according to one product recipe are called instances of that recipe which enables reuse of recipes. In order go between two locations of operations in a product recipe a transport operation has to execute. The term location of operation can be described as the physical space where an operation is realized. The

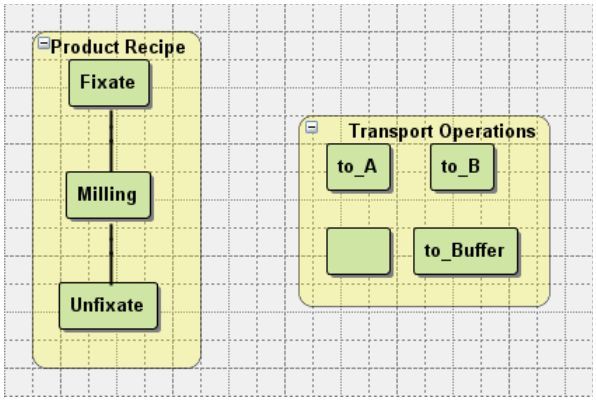


Figure 2.2: A product recipe and transport operations.

following is an example of the conditions that have to be fulfilled in order for a transport operation to be enabled between two locations A and B:

- The product has booked a carrier
- The product is in location A
- Location B is available, i.e. free

A carrier could e.g. be a fixture or a basket. The combination of transport operations and product recipes is represented in Figure 2.3. In this figure alternative routes are introduced. In order to go from operation *Fixate* to operation *Milling* the route through *to_A* could be taken, which for instance could represent a transport operation to e.g. a milling machine A. However, the route to *to_B* could also be chosen, which could represent a transport operation to e.g. a milling machine B. If both machines that can perform the task of *Milling* are busy, the route to a buffer and later to one of the machines could be taken. Next, the *Milling* operation could occur.

The empty box in Figure 2.2 and 2.3 will show the advantage with this approach. If a new resource is introduced to a production cell only transport operations to that machine have

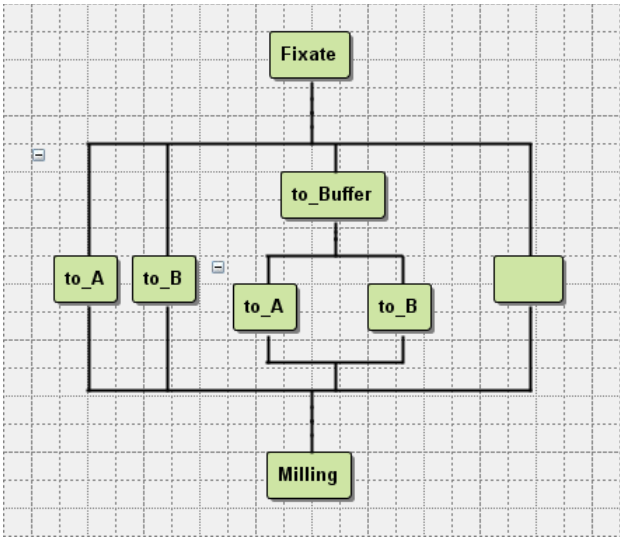


Figure 2.3: The combination of the product recipe and the transport operations.

to be added. In Figure 2.3 an extra alternative shows up as a result of the new machine added to the production system.

With this approach, introducing conditions for a transport operation to occur, logics become a part of the information to be extracted from the production cell. It is therefore appropriate to model the system as Extended Finite Automata (EFA). This framework of modeling is an extension to ordinary automata but with the exception that it introduces integer variables to the model. EFA will be explained more in detail in Section 2.3.

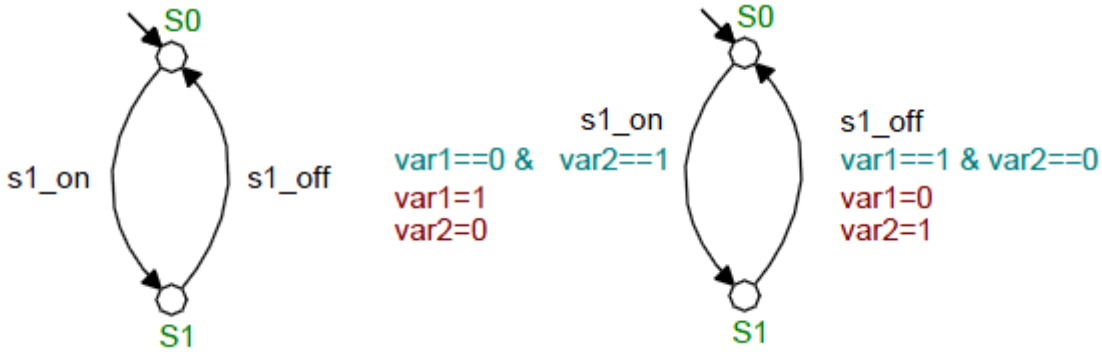
2.3 Extended Finite Automata

Discrete event systems (DES) are systems whose state space is discrete and whose state changes are associated with an event occurring at discrete points in time [3]. The main elements of DES are the discrete state space and the discrete event set. Deterministic Finite Automata (DFA) is a modeling framework for discrete event systems. These automata consist of a finite number of states and transitions between states associated with the occurrences of events. A transition is a directed arc showing in what way an automaton can move from one state to another, or to the same state. An event is connected to a transition and causes the system to move from one state to another [5].

The automaton in Figure 2.4(a) could for instance represent a sensor with two states, off (S0) and on (S1). The event *s1_on* is associated with the transition going from state off to state on, i.e. the sensor is turned on. The opposite holds when the sensor is turned off.

An Extended Finite Automaton (EFA) is an augmentation of an ordinary automaton [10].

For EFA, guards and actions are introduced with transitions. In order for a transition to be enabled the guard formula has to be fulfilled, i.e the condition of the guard has to be true. When the transition is taken, a set of variables may be updated according to the actions. Figure 2.4 displays the difference between the two automata representations. The



(a) DFA (b) EFA

Figure 2.4: Illustration of the differences between (a) Deterministic Finite Automata and (b) Extended Finite Automata. In both automata there are two states, S0 and S1, and two events, *s1_on* and *s1_off*. In (b) the guard formula is on the first row under the event, and the actions consist of the rows under the guard.

EFA in Figure 2.4(b) has two variables constituting the guards and actions. For event *s1_on* to be enabled, the guard formula has to be true, i.e. variable *var1* is 0 and variable *var2* is 1. When the guard condition is true and a transition is taken, the variables are updated. In this example, the values of *var1* and *var2* are set to 1 respective 0. EFA can be modeled in a verification tool called Supremica, developed at the department of Signals and Systems at Chalmers University of Technology.

2.4 Process Simulate

Process Simulate is a part of Tecnomatix application suites owned by Siemens Product Lifecycle Management Software Inc. It enables process planning, resource planning, part planning and simulation of a manufacturing process in a virtual environment. In order to locate where different carriers can be situated inside the FLEXA production cell and to retrieve this information using the Tecnomatix .NET API, it is important to study the structure of the software. The basic configuration of the applications in Tecnomatix is given in Figure 2.5. The software is organized in three layers consisting of a database, a server and clients.

The first layer consists of an Oracle Database Server whose main task is to manage data and ensure the data update is handled correctly. The database is sub-divided into schemas. The eMServer is in the next layer and this is the core element in the configuration. Services are provided by the eMServer to applications/clients, e.g. requests for data from clients are sent forward to the database. The eMServer handles the communication between the clients and the database. The last layer contains the different clients which can represent the applications in the Tecnomatix environment, e.g. Process Simulate. [8]

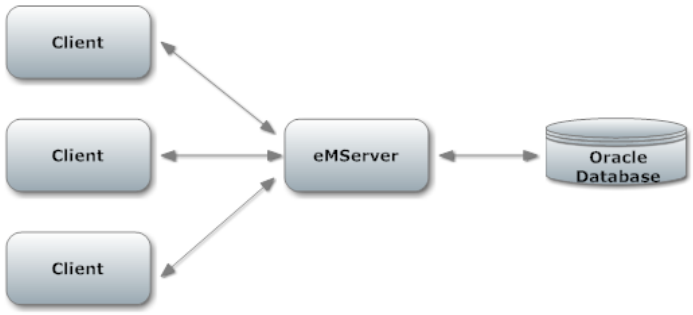


Figure 2.5: The structure of Tecnomatix consisting of three layers; a database, eMS-server and clients [8].

The data structure of the Tecnomatix clients is displayed in Figure 2.6. As mentioned previously, the database is divided into schemas. These consist of projects which are each made up of objects/nodes structured in trees. These nodes are products, operations, resources and manufacturing features that together define the manufacturing process. A tree can only contain a group of the same nodes, e.g. resources. Finally, attributes can be attached to the nodes, e.g. files with 3D data or AutoCAD files. [8][9]

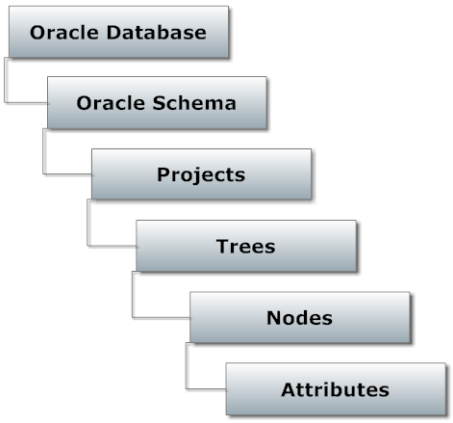


Figure 2.6: The data structure in the Tecnomatix environment.

Retrieving resources and frames in Process Simulate are of great importance for this thesis. Using the Tecnomatix .NET API enables the possibility to function as a client. In the Process Simulate client environment displayed in Figure 2.7, the *Object Tree* contains a hierarchy of the elements displayed in the loaded cell. Example of elements are weld points and paths but also the objects of high interest for this project, i.e. frames and resources. These are located under folders with equivalent names. In parallel with these folders, the *Graphic Viewer* window displaying the production cell is of interest.[9]

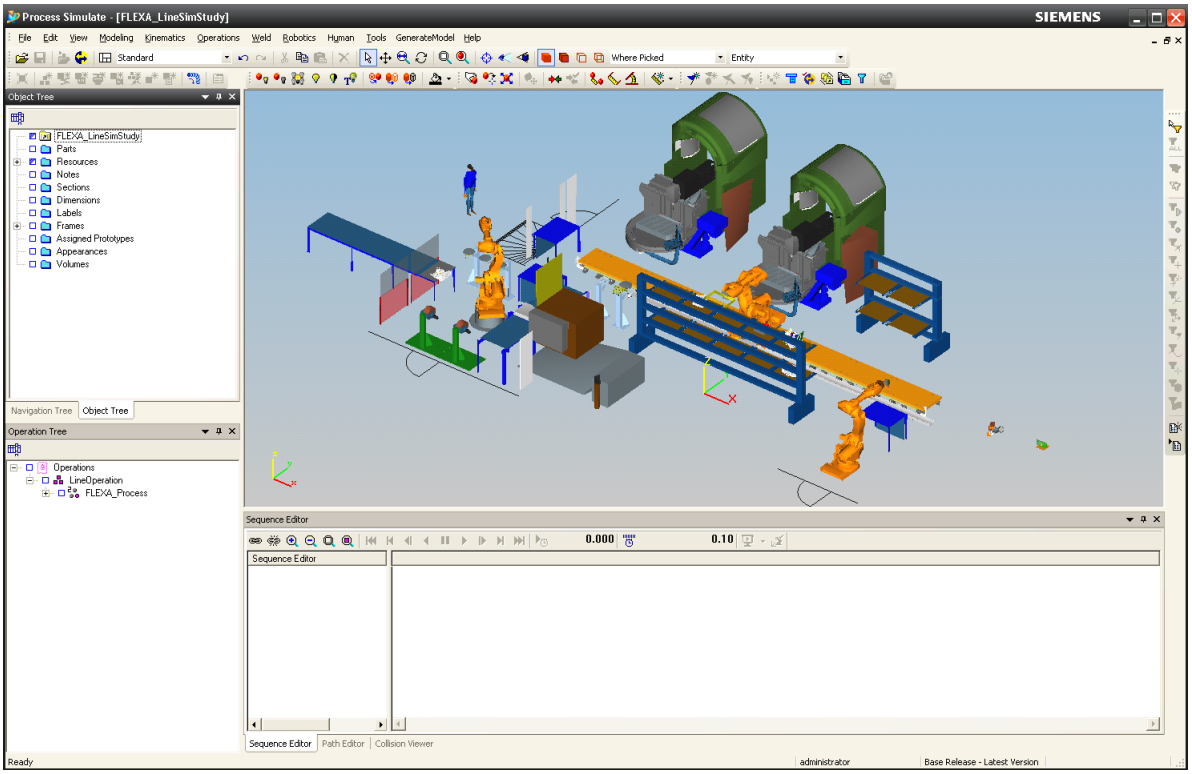


Figure 2.7: The Process Simulate work window. In the work environment viewers such as the *Object Tree Viewer* and the *Graphic Viewer* take up most of the window. The *Object TreeViewer* is located on the upper left side and contains information regarding the loaded cell, e.g. resources displayed in the folder *Resources* and locations of different objects contained in the folder *Frames*.

2.5 Supremica's .wmod format

In order to open the file with transport operations in Supremica the format of the .xml file has to be in .wmod. This section will explain the structure of this format. The .wmod format has a root element named *Module* which has two children nodes, named *EventDecList* and *ComponentList*.

- *EventDecList* contains declarations of all events.
- *ComponentList* contains descriptions of all constituting components of the model, namely automata and variables. The children nodes are
  - *SimpleComponent*
  - *VariableComponent*

The node *SimpleComponent* contains information about states, events and transitions for the given automaton. For each state, incoming and outgoing transitions are given with respective labels, as well as guards and actions for that specific transition.

The node *VariableComponent* contains children nodes *VariableRange*, *VariableInitial* and *VariableMarking*. The first child node defines what values the variable can have. The next child node gives the initial value for the variable while the last child node defines whether the variable is marked and if so, what value it has.

2.6 Sequence Planner

Sequence Planner is an application developed by the department of Signals and Systems at Chalmers University of Technology. The application can be viewed as an interface to Supremica where the formal verification of systems is performed. Supremica returns a result of the verification back to Sequence Planner [6]. In Sequence Planner it is possible to model operations as Sequential Function Charts (SFC) where both parallel and alternative tracks are possible to model. A Sequence of Operations (SOP) can contain several operations. The operations in Figure 2.2 and 2.3 are modeled in this software. The operations in Figure 2.2, representing the product recipe and the transport operations, are SOPs. In Figure 2.3 there are alternative tracks at two locations. Preconditions and postconditions can also be attached to the operations modeled in Sequence Planner defining when an operation is enabled to start and finish.

In this thesis the information extracted from Process Simulate is written to a format readable by Supremica. This is because the intention is to use the simulation tool as the interface generating the transport operations, and afterwards using this information in a verification tool like Supremica. If an intermediate step is included in the project where the user is supposed to visualize the generated operations, Sequence Planner can be appropriate to use. This is simply because the format in Sequence Planner is more adapted for industry than the format in Supremica. In Supremica the transport operations are displayed as automata while Sequence Planner displays them as in Figure 2.2.

3 Method

The intention with the concept of product recipes and transport operations is to improve the flexibility of a system. It should be possible to introduce new resources to a cell without much manual effort or to manufacture multiple products in the same production cell. Imagine if a human operator is supposed to keep track of two different parts, and all different locations these parts can be situated in. As the amount of possible locations increases or if a new product is introduced in the cell, the complexity in keeping track of all possible locations for each product is increased to a level to hard for a human to handle. The virtual production cell used in this project is as mentioned earlier the FLEXA production cell, see Figure 3.1.

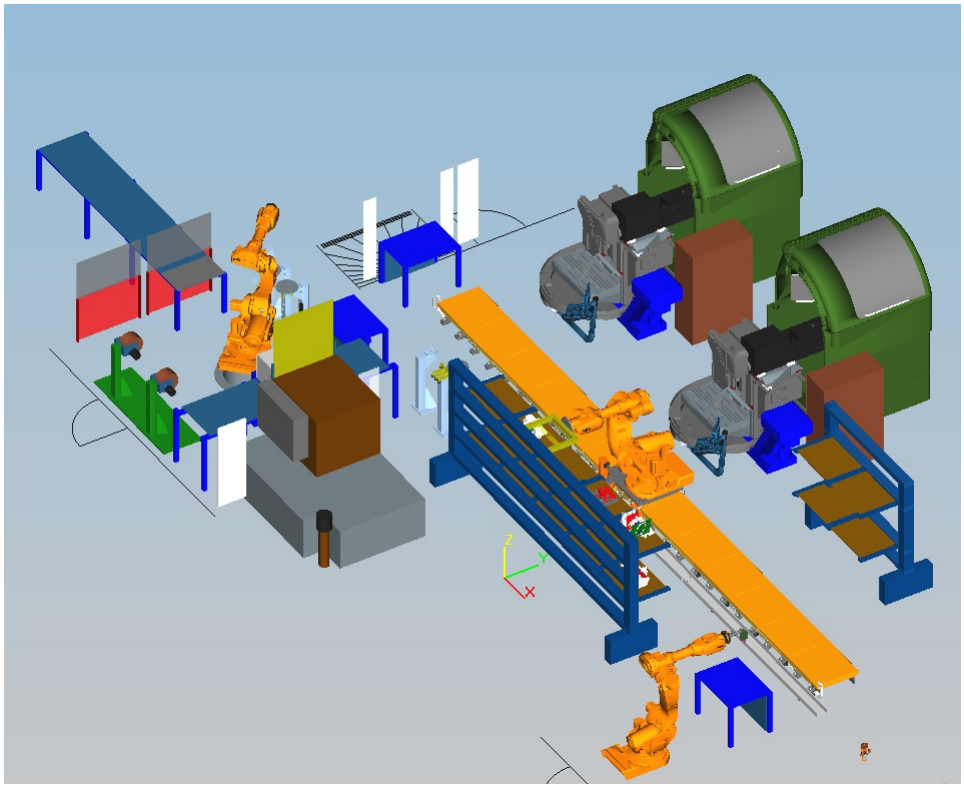


Figure 3.1: The FLEXA Production Cell.

The procedure for automatically generating transport operations can be given in the following steps:

- Create frames in Process Simulate
- Choose possible locations for different carriers
- Parse locations to transport operations written to a .wmod file

Reversed engineering has more or less been used since the information to extract was known but not how to retrieve that specific information from Process Simulate. The first step was therefore to learn more about the software and what options there were to use the Tecnomatix .NET API to extract data. A plug-in was developed using C# to take care of the steps mentioned above. In the following section the plug-in will be described in more detail.

3.1 Developing a plug-in in Process Simulate

A plug-in was developed in Process Simulate which made it possible to extract information from the software. When studying the documentation of the Tecnomatix.NET API, a class was found that enabled the creation of a command button integrated into Process Simulate. The documentation included how to access information about e.g. resources and frames but also how to create new frames [7]. The steps used for creating a plug-in can be described using a flowchart diagram, see Figure 3.2. The first step was to choose a file to extract the information to. In this thesis the file was supposed to be opened in Supremica, thus the filename extension .wmod was attached to the file. The next step in the flowchart diagram was to choose the global set of locations that all carriers could be placed in. When this step was completed, the locations for every carrier were chosen. The final step was to parse the information into a structured .xml file. In the following sections every class starting with *Tx* is a class given in the Tecnomatix .NET API [7].

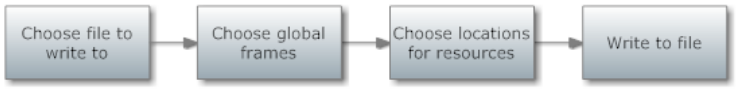


Figure 3.2: The flowchart shows an overview of how the plug-in works.

3.1.1 Creating commands

To construct a button command that is integrated into Process Simulate, it was possible to create a class in C# which implemented a class called *TxButtonCommand*. The result was a custom command introduced in the Process Simulate environment [7]. In this thesis a simple button command was created that appeared in the menu bar. When the button was clicked the command related to the button executed. A well-known example of a button command is the *Help* button that usually displays a GUI with different help options when clicked.

3.1.2 Global locations

The class *TxFrameEditBoxCtrl* could be used in order to choose locations in the Graphic Viewer of Process Simulate. This class has a member event that listens to when a pick

is made. The coordinates of the picked location could be saved as an object of *TxTransformation*. Using *TxFrameCreationData* enabled the creation of a frame in the chosen location. The approach in choosing the global locations for all the possible carriers can be described by the flowchart in Figure 3.3. To avoid errors every frame created had to be unique. Therefore the first step was to choose a name of the frame. If an already existing frame had the same name, another name had to be given. Next, a location could be chosen by picking a location in the Graphic Viewer. [7]

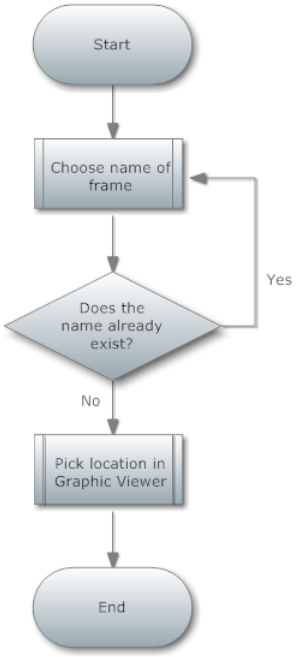


Figure 3.3: The flowchart representing the procedure of choosing global locations.

3.1.3 Locations for resources

When global locations had been chosen, locations for each carrier were picked. These locations were a subset of the global locations. Using *TxApplication.ActiveDocument.PhysicalRoot.GetAllDescendants* it was possible to access all physical objects in the virtual cell in Process Simulate [7]. In order to retrieve the nodes displayed in the folder *Resources* in the Object Tree, all objects were filtered using a class *TxTypeFilter*. If *TxPlanningResource* was used as a filter, the resulting objects were resources and saved for later use.

The flowchart in Figure 3.4 describes the procedure of picking locations for each carrier. First, a resource had to be chosen. If the resource already had been given locations, another resource had to be chosen. Next, the locations for the specific resource had to be chosen. The last step was to pick initial/marked location for the resource. In this thesis,

an assumption was made that each resource had a "home location" where it started and ended. Therefore, the initial and marked location was assumed to be the same. If the initial/marked location already had been picked by another resource, a new location had to be chosen.

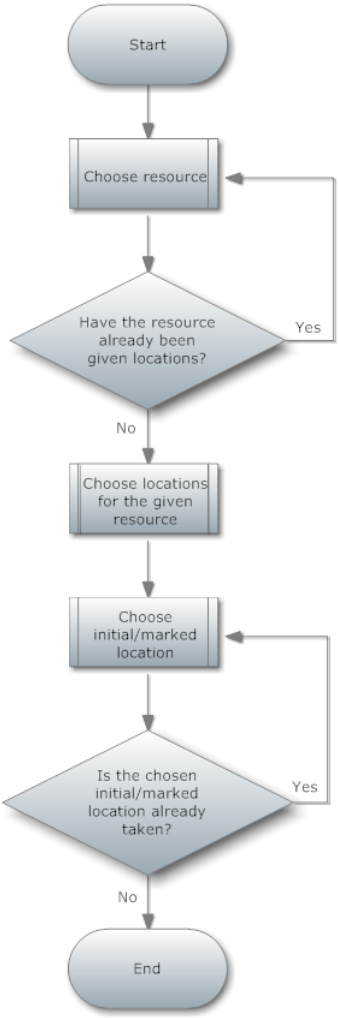


Figure 3.4: The flowchart shows the procedure of choosing locations for each resource.

3.2 Writing the .wmod file

When global locations, resource locations and initial/marked location had been given it was time to parse the locations to transport operations and write the information to an .xml file. Global locations and locations of resources were stored into lists [4]. The name of the initial/marked state was saved for later use. The next step was to write the information to an .xml file in .wmod format. A class called *XmlTextWriter* was used to parse

the information into a structured .xml file [4]. The flowchart in Figure 3.5 represents the steps taken for this procedure. The first step was to loop through the list containing the locations for a resource. Basically, there were two identical lists, one for *from*-locations and one for *to*-locations for a resource. If the *from*-location and *to*-location were identical next element in the list was examined. This because a transport operation was never supposed to occur from one location to the exact same location. If the locations were not identical, an event was created with the name of the resource plus the name of the *from*-location together with the name of the *to*-location, e.g. **Fixture1_in_Sta1_to_Sta2**.

Next, the information for the element *SimpleComponent* described in Section 2.5 was written. If the *from*-location was initial/marked the *VariableInitial* and the *VariableMarking* for the variable *VariableComponent* named after the resource name plus in-location were set to 1. Else the variables were set to 0.

The next step was to write the *VariableComponent* for the *to*-locations. From the beginning all *to*-locations variables were set to zero. When all *to*-locations had been looped through, the next element in the *from*-locations list was examined. The final step was to create a number of *VariableComponent* equal to the number of space resources. These variables were set to 1 if a resource had an initial/marked location in that global position, else they were set to 0.

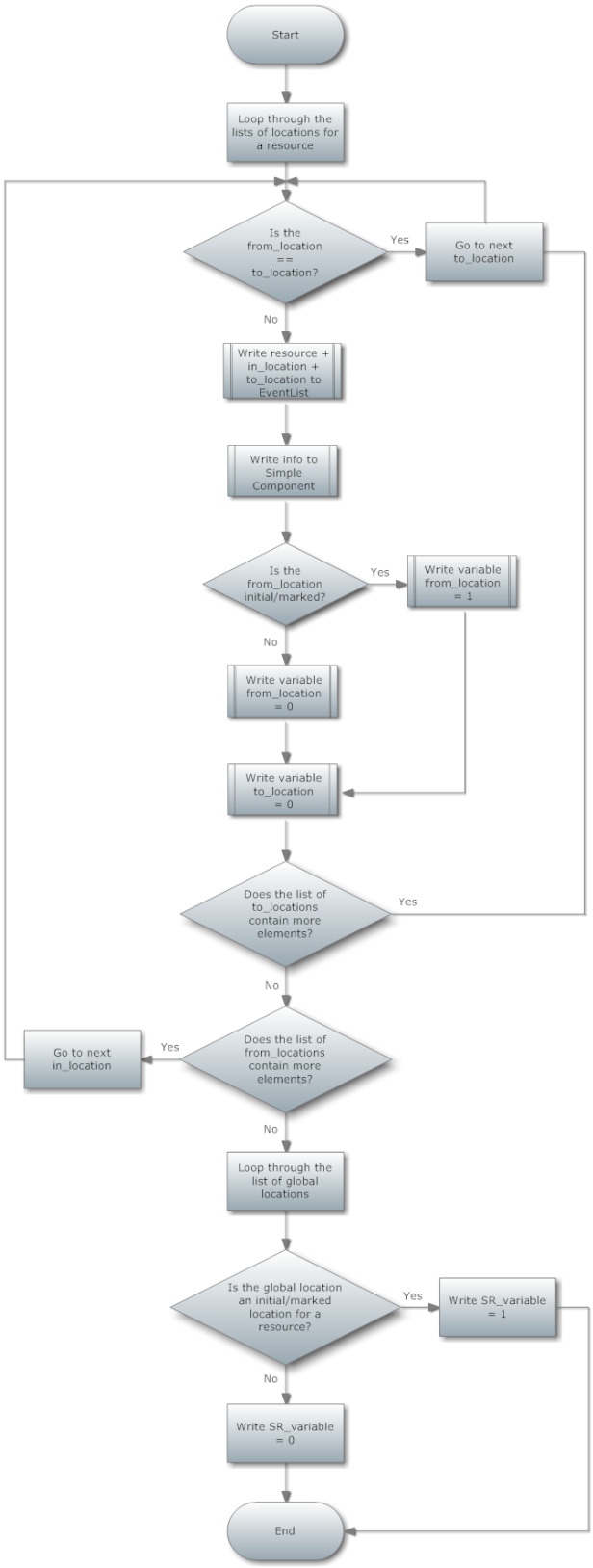


Figure 3.5: The flowchart shows the procedure of writing the information to a .wmod file.

4 Results

The procedure for this project was to develop a plug-in from which it was possible to create a global set of locations in a virtual cell, and later choose a subset of these locations for specific resources. A simple button command was created and implemented in the menu bar of Process Simulate. When the button was clicked, a GUI with three buttons was displayed where the user was instructed through some easy steps in how to create and pick locations. This information was parsed into a structured .xml file which was opened in Supremica. The parsing transformed the information given into transport operations in the form of EFA with guards and actions attached to transitions. The following sections go into details of the results for the different parts. In Appendix A the structure of the final program is described while Appendix B contains the steps for integrating a command into Process Simulate.

4.1 Simple Button Command

The simple button command described in Section 3.1.1 was named GenerateModel and was placed in the menu bar of Process Simulate to the left of the Help button, see Figure 4.1. Once the user clicked on the button the GUI displayed in Figure 4.2 appeared. In

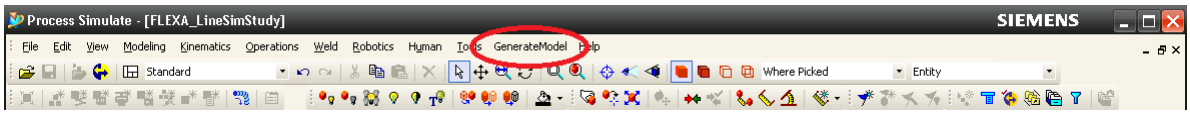


Figure 4.1: The simple command button GenerateModel implemented in the Process Simulate environment.

Appendix B the procedure for integrating a command into Process Simulate is given. The class implementing *TxCommandButton* is described further in Appendix A.

4.2 Graphical User Interface

An user-friendly GUI was developed in C# to make it easy for a Process Simulate user to create locations, choose resource specific locations and finally parse the information to an .xml file. The steps previously mentioned in Section 3.1.2, 3.1.3 and 3.2 have been covered in the GUI in Figure 4.2. This was the main window for the plug-in and was named GenerateModel. If the first button named *Browse* was clicked a file dialog allowed the user to choose a name and directory for the .wmod file to be saved in. As usual, if the user chose an already existing file a message box was displayed asking the user if the file should be replaced. The events of buttons *Pick Location* and *Resources* are described in detail in the following sections. All error messages mentioned in these sections refer to an error represented as a red circle with an exclamation mark in the center and a text box with an explanation of the error.

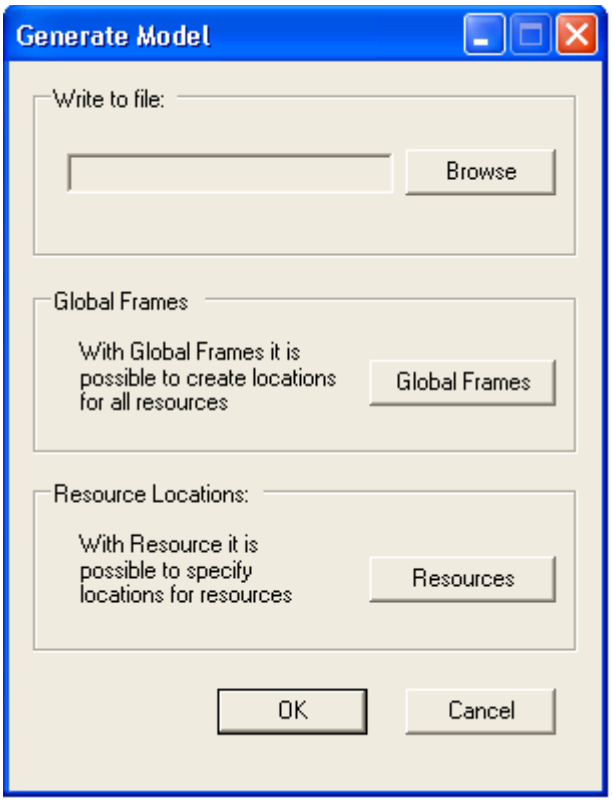


Figure 4.2: The GUI for automatically generating transport operations to a .wmod file.

4.2.1 Defining global locations

If button *Global Frames* was clicked a window with corresponding name was displayed, see Figure 4.3. In this GUI the user had to first choose a name for a location. If a name had

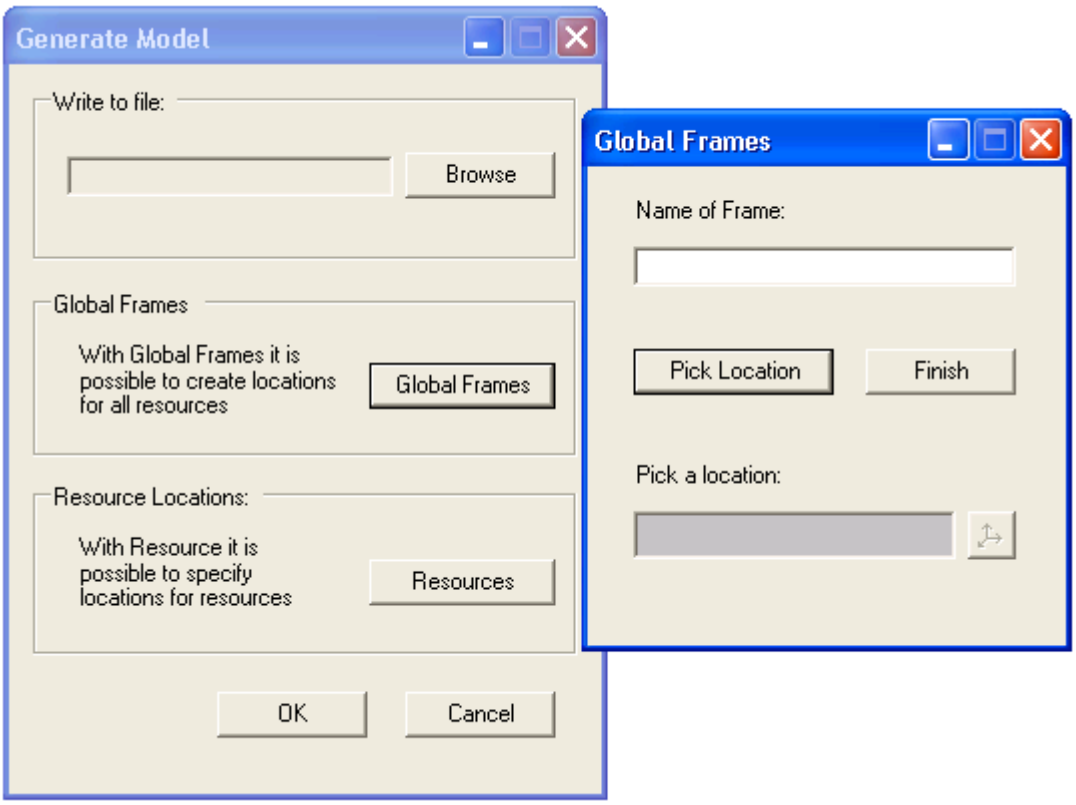


Figure 4.3: The GUI for choosing global locations.

not been given and the *Pick Location* button was clicked, an error was displayed saying that a name had to be given. Else a control was performed to see if a location with the same name already existed. If so, an error message was provided. When *Pick Location* was clicked the mouse pointer was attached with a small axis of coordinates indicating that a pick in the Graphic Viewer could be made in order to choose a location for a frame. When completed, a new frame with the given name was created in the Object Tree. In Graphic Viewer an orange frame appeared where the click occurred, see Figure 4.4. All global locations were created according to this procedure. When the *Finish* button was clicked, the window closed and the main window Generate Model was displayed.

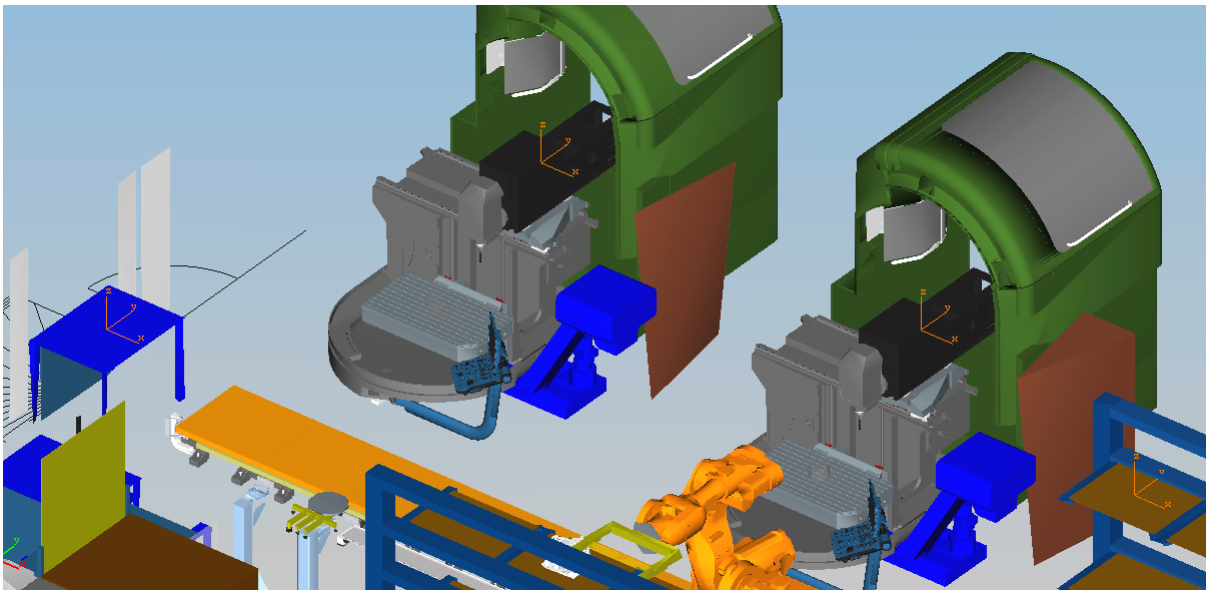


Figure 4.4: The Graphic Viewer with orange frames representing global locations. In this figure, there are frames in both machines, on a table and on a shelf.

4.2.2 Defining locations for resources

When *Resources* was clicked a new window named Resource Locations appeared, see Figure 4.5. The first box displayed all resources in the virtual model and the next box displayed the global locations created in the previous step. Locations picked in the second box were displayed in the third box. This gave the user the possibility to choose the initial/marked location for the resource.

If a resource already had been given locations or if a location in the third box already had been chosen as initial/marked state, error messages were displayed for respective error. If the user only chose one location an error message was also provided since it would have equaled a resource not supposed to move. As a consequence, a transport operation should not have been generated. When the user clicked on *OK* in Resource Location, the main window Generate Model was displayed. The user could continue clicking on *Resources* in order to choose locations for other carriers. When the user clicked on *OK* in the main window it closed and the information was automatically generated to transport operations and written to a .wmod file.

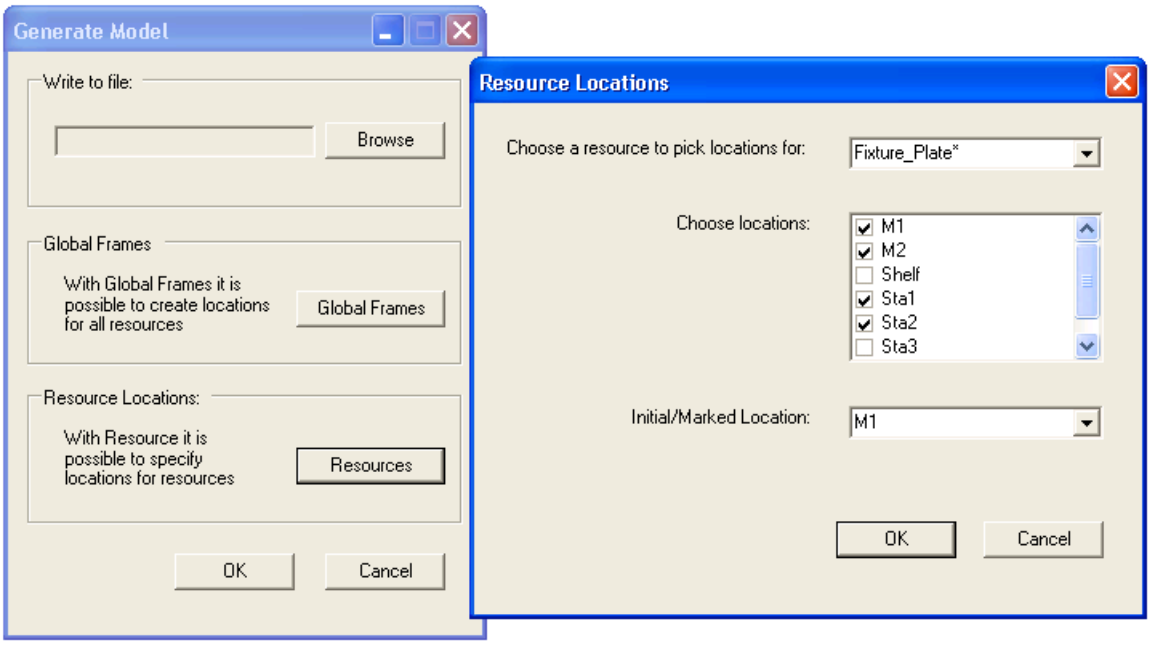


Figure 4.5: The GUI for choosing locations for resources.

4.3 Extracted Information

The plug-in was developed to extract locations for resources and global locations using the GUI. The information was parsed into an .xml file with the .wmod format described in Section 2.5. Once the task was completed the file could be open in a verification tool like Supremica. The two following sections will show the results of the procedure.

4.3.1 .wmod file

Once the user had followed the steps in the GUI the information was written to an .xml file. In Figure 4.6 the resulting file is displayed. In this example file, there are three events declared in *EventDeclList*. Each transport operation is given as a plant in *SimpleComponent* which is a child node to *ComponentList*. The number of *SimpleComponent* equals the number of transport operations. The levels under *NodeList* contain information about the location of the EFA, i.e. the state of an ordinary automaton. The levels under the node *EdgeList* define the transitions, labels of transitions, guards and actions. In this example there is only one transition, else each transition would be represented by a node *Edge*.

In Figure 4.7 variables created from parsing locations into transport operations are displayed. The number of nodes called *VariableComponent* equals the number of variables. As mentioned earlier, *VariableRange* defines what values the variable can have. If the value of the variable Fixture_Plate_in_M1, as in this example, is 0 this means that the resource

```
<?xml version="1.0" standalone="yes"?>
<Module Name="New Module" xmlns:ns2="http://waters.sourceforge.net/xsd/base" xmlns="http://waters.sourceforge.net/xsd/module">
  <EventDeclList>
    <EventDecl Kind="CONTROLLABLE" Name="Fixture_Plate_in_M1_to_M2" />
    <EventDecl Kind="CONTROLLABLE" Name="Fixture_Plate_in_M2_to_M1" />
    <EventDecl Kind="PROPOSITION" Name=":accepting" />
  </EventDeclList>
  <ComponentList>
    <SimpleComponent Kind="PLANT" Name="Fixture_Plate_in_M2_to_M1">
      <Graph>
        <NodeList>
          <SimpleNode Initial="true" Name="Fixture_Plate">
            <EventList>
            <PointGeometry>
            <InitialArrowGeometry>
            <LabelGeometry Anchor="NW">
          </SimpleNode>
        </NodeList>
        <EdgeList>
          <Edge Target="Fixture_Plate" Source="Fixture_Plate">
            <LabelBlock>
              <SimpleIdentifier Name="Fixture_Plate_in_M2_to_M1" />
              <LabelGeometry Anchor="NW">
            </LabelBlock>
            <SplineGeometry>
            <GuardActionBlock>
              <Guards>
                <BinaryExpression Operator="&amp;amp;" Text="Fixture_Plate_in_M2==1 &amp;amp; Fixture_Plate _to_M1==1 &amp;amp; SR_M1==0">
                  <BinaryExpression Operator="&amp;amp;">
                    <BinaryExpression Operator="==">
                  </BinaryExpression>
              </Guards>
              <Actions>
                <BinaryExpression Operator="=">
                <BinaryExpression Operator="=">
                <BinaryExpression Operator="=">
                <BinaryExpression Operator="=">
                <BinaryExpression Operator="=">
              </Actions>
              <LabelGeometry Anchor="NW">
            </GuardActionBlock>
          </Edge>
        </EdgeList>
      </Graph>
    </SimpleComponent>
    <SimpleComponent Kind="PLANT" Name="Fixture_Plate_in_M1_to_M2">
```

Figure 4.6: The resulting xml file after extracting the information from Process Simulate.

named *Fixture_Plate* is not in location **M1**. If the value is 1, the opposite holds, i.e. the specified resource is in location **M1**. *VariableInitial* defines the initial value of the variable and *VariableMarking* defines if the given variable is marked and if so, what value it has. The variable names that start with SR, which is an abbreviation for Space Resource, is variables for each global location. These variables can have value 0 or 1, depending on if a resource has booked the space/location or not.

```
<VariableComponent Name="Fixture_Plate_in_M1">
  <VariableRange>
    <BinaryExpression Operator="..">
      <IntConstant Value="0" />
      <IntConstant Value="1" />
    </BinaryExpression>
  </VariableRange>
  <VariableInitial>
    <BinaryExpression Operator="==">
      <SimpleIdentifier Name="Fixture_Plate_in_M1" />
      <IntConstant Value="1" />
    </BinaryExpression>
  </VariableInitial>
  <VariableMarking>
    <SimpleIdentifier Name=":accepting" />
    <BinaryExpression Operator="==">
      <SimpleIdentifier Name="Fixture_Plate_in_M1" />
      <IntConstant Value="1" />
    </BinaryExpression>
  </VariableMarking>
</VariableComponent>
<VariableComponent Name="Fixture_Plate_to_M1">
<VariableComponent Name="Fixture_Plate_in_M2">
<VariableComponent Name="Fixture_Plate_to_M2">
<VariableComponent Name="SR_M1">
  <VariableRange>
    <BinaryExpression Operator="..">
      <IntConstant Value="0" />
      <IntConstant Value="1" />
    </BinaryExpression>
  </VariableRange>
  <VariableInitial>
    <BinaryExpression Operator="==">
      <SimpleIdentifier Name="SR_M1" />
      <IntConstant Value="1" />
    </BinaryExpression>
  </VariableInitial>
</VariableComponent>
<VariableComponent Name="SR_M2">
```

Figure 4.7: The variables in the resulting .xml file after extracting the information from Process Simulate.

4.3.2 Transport Operations

To visualize how the transport operations would look like as EFA with guards and actions, the xml file was opened in Supremica. Figure 4.8 displays the plant representing a transport operation for a resource named *Reg_Shrmillfix* to go from a location named **Sta3**

to a location named **Sta2**. The guard condition in this plant defines that the resource *Reg_Shrmillfix* has to be in location **Sta3** and want to go to location **Sta2** but also that the space resource for **Sta2** has to be unbooked. It these conditions are fulfilled the event is enabled. The variables are updated when a transition is occurs.

In the given example the actions are to set the variable for resource *Reg_Shrmillfix* to be in the first location **Sta3** to 0, this because by now we have moved from that location. The variable for the same resource to be in location **Sta2** is set to 1 since this is the current location. The variable for the same resource to go to **Sta2** is also set to 0 since this location is now the current location of the resource. The space resources have to be updated. Since the resource is no longer in **Sta3** the variable is set to 0 while the space resource for **Sta2** is set to 1 since this location is occupied by the given resource.

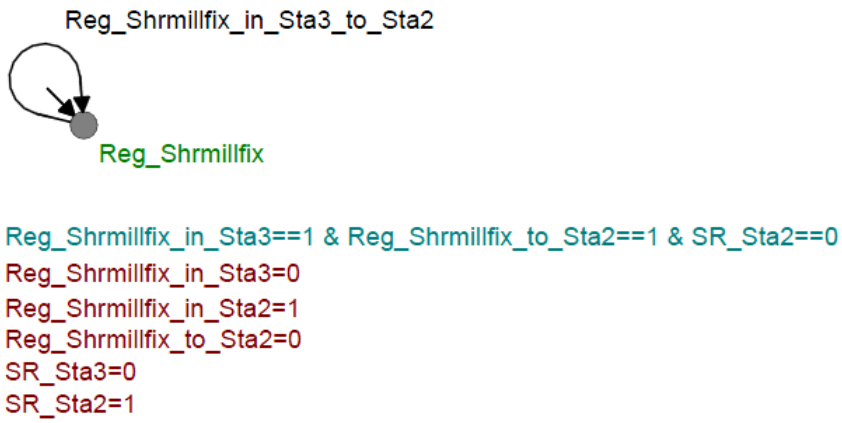


Figure 4.8: The transport operation for carrier *Reg_ Shrmillfix* to go from location **Sta3** to location **Sta2**.

In short, we can describe the transport operation as a single state with a self-loop. For the event of the self-loop to occur the guard conditions have to be fulfilled. When or if they are, the given actions of the EFA will update the variables.

4.4 Hands-on example

To summarize the result of this master thesis, a hands-on example will be given in this section. The production cell given in Figure 3.1 will be used to specify global locations and carrier specific locations. The information is parsed into a .wmod file which is opened in Supremica [2].

The procedure starts with a click on GenerateModel in the menu bar. As a result, the GUI in Figure 4.2 is displayed. By using *Browse* a file can be chosen to save the information to. In Figure 4.9 the first global location is created using the button *Global Frames* on the GUI, i.e. a name is written and a pick is made in Graphic Viewer to choose a location. In this case the name of the location is **Sta010**. An orange frame appears in Graphic Viewer where the pick was made. These steps are repeated until all global locations have been created. In this example there are three global locations positioned as in figure 4.4. The

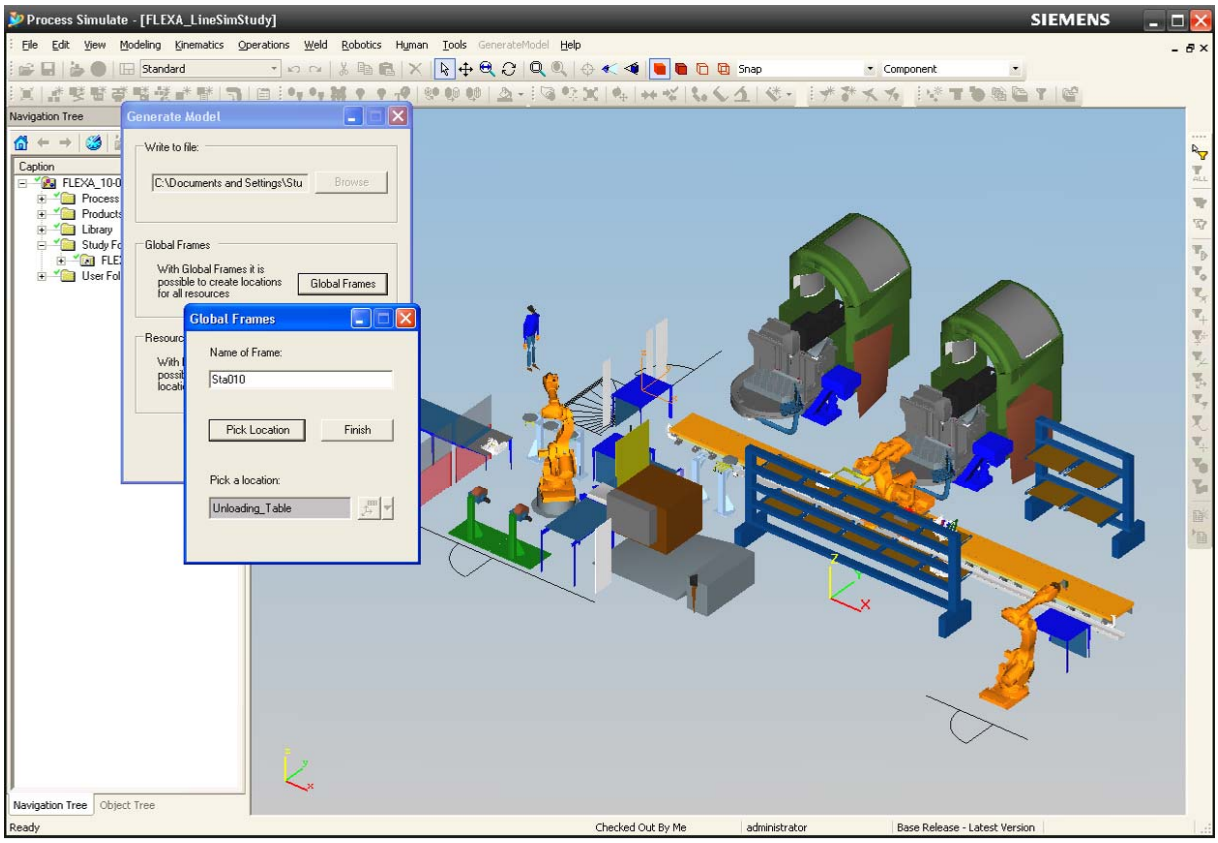


Figure 4.9: The procedure of creating global locations in the cell. First, a name is given then a pick is made in Graphic Viewer in order to choose a location.

name of the first position is as mentioned **Sta010**. The two other positions are located in the machines and are named **M1** and **M2**.

The next step is to choose locations for each carrier using the button *Resources* on the GUI, see Figure 4.2. The resources loaded in the cell are displayed under the first box. First a carrier has to be chosen from the list, in this example the chosen carrier is called *Reg_Hubmillfix*. Next, the locations for this carrier are chosen, in this case **Sta010** and **M1**. The last step is to choose initial/marked location for the carrier according to Figure 4.10. This location is the starting and ending point for the carrier. In this example only one carrier is chosen. To end the procedure the button *OK* on the main window of the GUI is clicked and the information is parsed to the chosen file.

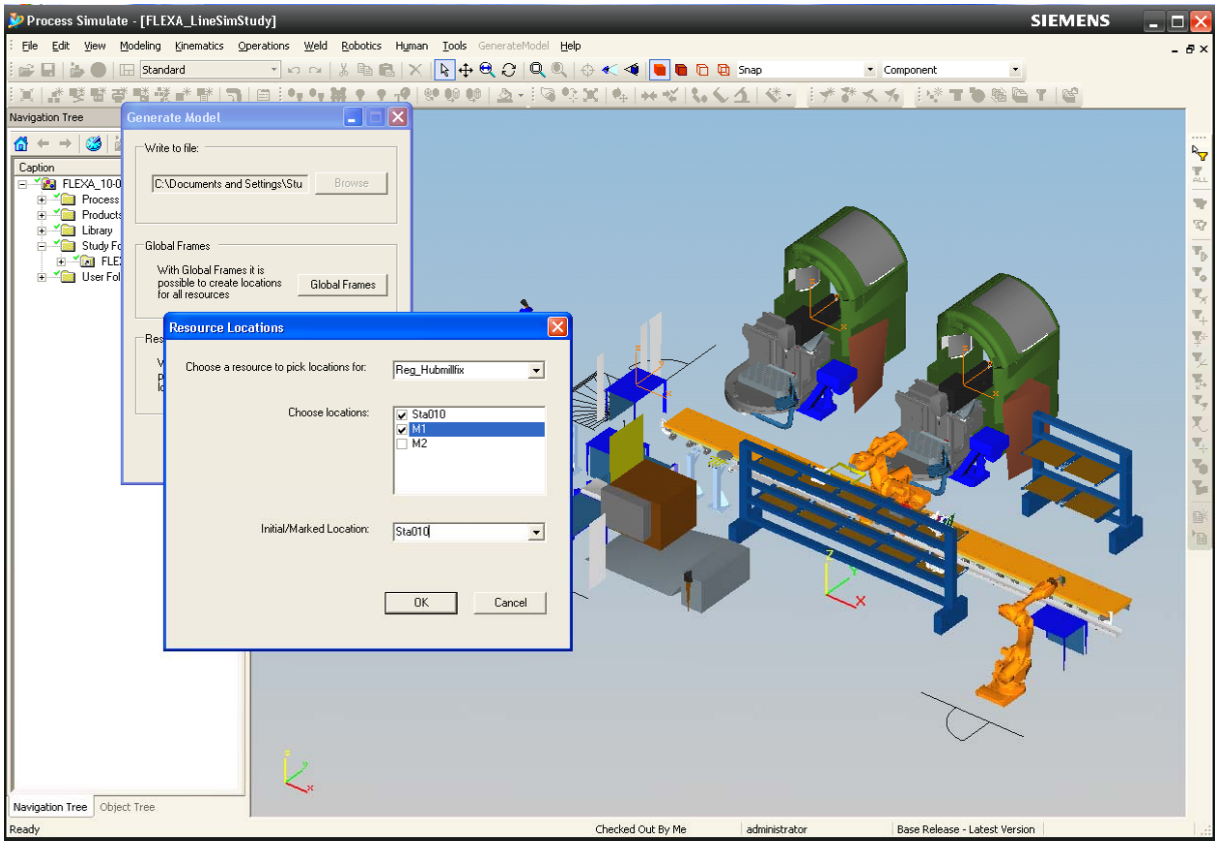


Figure 4.10: The procedure of picking locations for a carrier.

Figure 4.11 displays Supremica loaded with the file used to save the information to. There are two transport operations, one for the case where the carrier is transported from location **Sta010** to location **M1** and one for the opposite case. There are a total of seven variables where four variables describe the current location of the carrier or to where the carrier wants to go. Three variables, beginning with SR, describe the global locations.

In this example, the initial/marked location for carrier *Reg_Hubmillfix* was in **Sta010**. This means that the variable SR_Sta010 will be initialized to 1 and the other two variables beginning with SR will be initialized to 0. The variable Reg_Hubmillfix_in_Sta010 will also be initialized to 1 since it starts in that position while the rest of the variables for *Reg_Hubmillfix* will be initialized to 0.

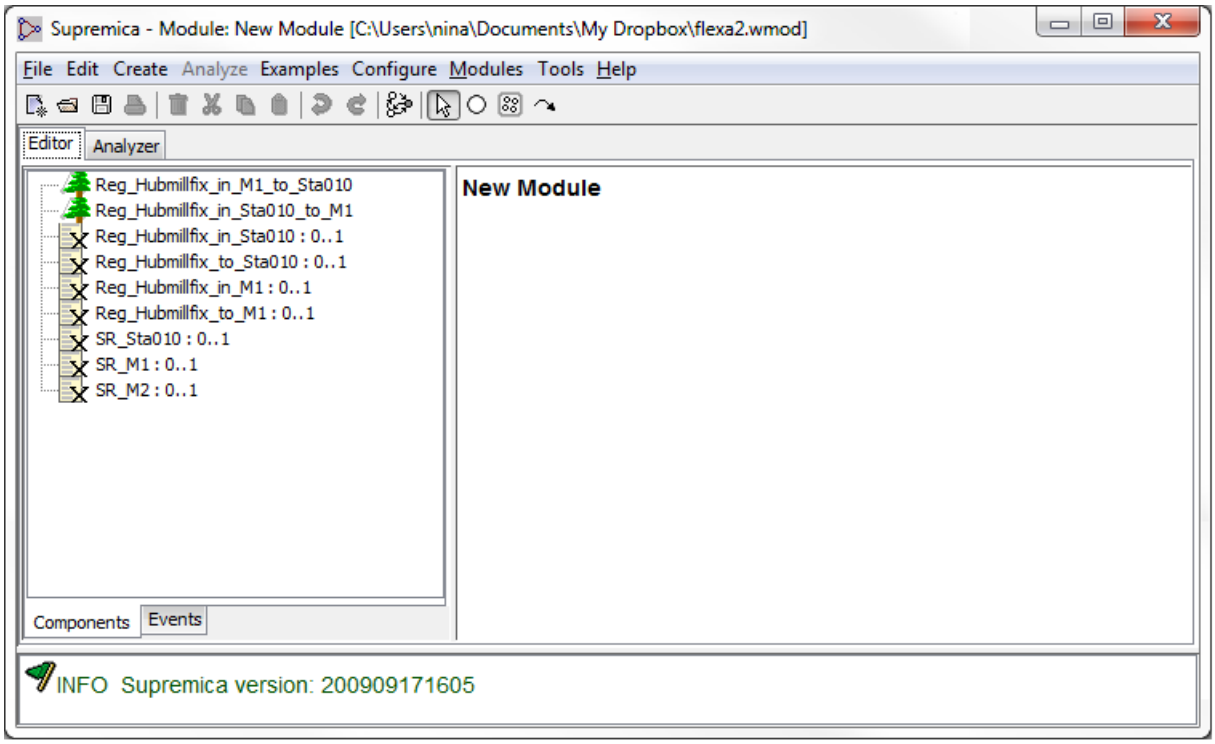


Figure 4.11: Supremica loaded with the file that was used to save the information to.

The transport operation Reg_Hubmillfix_in_M1_to_Sta010 is displayed in Figure 4.12. For a transport to occur, the conditions on the first row below the automaton need to be fulfilled as explained earlier. Once the transition is made, the variables are updated according to the remaining rows.

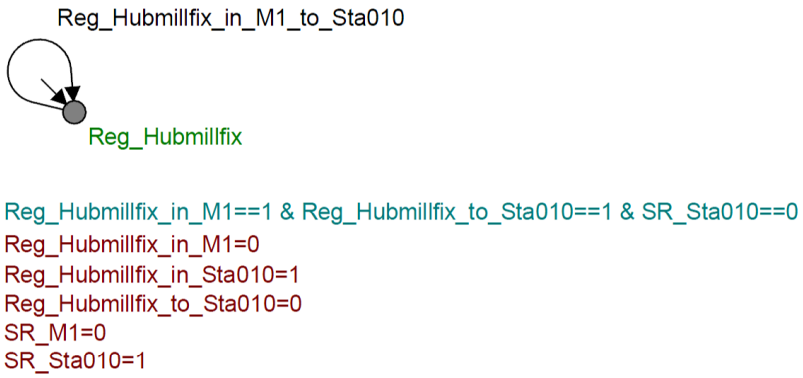


Figure 4.12: The transport operation for carrier *Reg_Hubmillfix* to go from location **M1** to location **Sta010**.

For Reg_Hubmillfix_in_M1_to_Sta010, i.e. for the carrier *Reg_Hubmillfix* to go from **M1** to **Sta010**, the conditions are:

- *Reg_Hubmillfix* has to be in **M1**
- *Reg_Hubmillfix* has a desire to go to **Sta010**
- The space for **Sta010** has to be available

When these conditions are fulfilled and a transition follows, the variables are updated according to:

- *Reg_Hubmillfix* is no longer in **M1**
- *Reg_Hubmillfix* is in **Sta010**
- *Reg_Hubmillfix* no longer has a desire to go to **Sta010**
- The space for **M1** is available
- The space for **Sta010** is no longer available

The opposite holds for transport operation Reg_Hubmillfix_in_Sta010_to_M1.

5 Discussion

This thesis introduced the concept of product recipes and transport operations. Process Simulate was used in order to specify where different carriers could be situated in the cell. With this information it was possible to transform the locations into transport operations. In Figure 4.8 the transport operation was described as an EFA with a single-state and a self-loop. The number of transport operations and variables became quite large for the total system. If one carrier could be in 4 locations, this resulted in 12 transport operations, 8 resource specific variables and 4 space variables representing global variables. However, no restrictions were put on the system regarding what routes to take, making the system flexible. The advantage with this approach is the flexibility it would bring and the possibility to reuse product recipes.

The procedure given in the GUI was to first choose global frames, i.e. all possible locations for the carriers to be in. Another choice would have been to specify each carrier and its different locations in the production cell. The reason to why this approach was not adapted, was because of errors that could be encountered. For instance, if a carrier has been given locations and one of these locations was in a place called A. If locations were to be picked for another carrier and this carrier also could be in A, the user had to pick on the exact same spot given by the preceding carrier. If the user would pick a location just besides the first location A, a new location would be created. These two locations, intended to be the same, would be interpreted as two different locations. Thus, the system would interpret one location as available even though it is actually not.

The transport operation displayed in Figure 4.8 has a redundancy of variables. Instead of introducing space resource variables for global locations, it would be possible to reduce the number of variables. This could be solved by instead of having a space resource for a location A and having to book that variable in order to move there, a control could be implemented in order to see if any of the other carrier in the cell is in that location. If not, the space is available and a move could be made. However, for better understanding when displaying the transport operations the approach with space resources was used for this thesis work.

Another way to reduce the number of variables, is to use an enumeration type for the current location for a carrier. This would result in one less expression for the actions in the EFA. If the transport operation in 4.8 is used as an example, the different locations where the carrier *Reg_Shrmillfix* could be located in are **Sta2** and **Sta3**. The enum becomes {Sta2, Sta3} where the first element is referred to as 0, the second to 1 and so on. The variables Reg_Shrmillfix_in_Sta3=0 and Reg_Shrmillfix_in_Sta2=1 could be bundled up into one expression. Using the enumeration type it would be possible to have a replacing variable Reg_Shrmillfix_in = 0, which would mean that the carrier is located in **Sta2**. For transport operations to be more easily interpreted, this approach was not used in this project.

In the FLEXA Production cell there is one robot performing the transport operations. The scope of this project excluded robots but it should be mentioned here that if more robots were added to the cell which could perform transport operations, it would be necessary to first book a robot before a transport could be executed. In this case, one more state would have to be added to the EFA to model that a robot first has to be booked before a transport operation is executed.

Before the information of locations for carriers is parsed into transport operations and written to an .xml file, controllers should be implemented to the simulation tool. They should perform e.g. reachability and collision tests in order to see if any motions between locations are unfeasible for a robot. If so, those transport operations should not be extracted. With this procedure it is possible to only extract significant information.

6 Conclusion

This thesis work was successful in retrieving locations of carriers from Process Simulate using a plug-in with a GUI that generated transport operations to an .xml file. The project shows that it would be possible to make specification early in a manufacturing process by using a simulation tool.

Information about locations for carriers was retrieved from the production system in order to generate transport operations. However, as mentioned in the discussion, information about product recipes was disregarded. If possible, the product recipe could as a future work be extracted from the cell, raising the level of abstraction even further. The question is where to look for it. Since the operations necessary in order to produce a product is vital for the manufacturing of a cell, it should be possible to find the operations constituting the product recipe.

If the previously mentioned recipes were to be retrieved from the simulation tool, then it would be possible to use both transport operations and product recipes and perform a formal verification in a tool like Supremica in order to see if the production cell conforms to the specifications.

In order to reduce the number of variables for the transport operations, it would be possible to continue this thesis work with the two approaches given in the discussion. These approaches would remove the variables for the space resources and reduce all the variables for a carrier's current location to an enumeration type describing all possible locations for that specific carrier. This would result in one less expression for the action functions in the EFA.

References

[1] FLEXA - Advanced Flexible Automation Cell, *A reseach project within the European Seventh Framework Programme(FP7)*. http://www.flexa-fp7.eu. 2010-05-28.

[2] Supremica. http://www.supremica.org. 2010-05-16.

[3] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer US, 2008.

[4] 2010 Microsoft Corporation. MSDN library. http://msdn.microsoft.com/en-us/library/ms123401(v=MSDN.10).aspx. 2010-05-03.

[5] H. Flordal. Modular controllability verification and synthesis of discrete event systems, 2001.

[6] E. Olsson and C. Thorstensson. Development, implementation and testing of sequence planner, a concept for modeling of automation systems, 2009.

[7] Siemens PLM. *Tecnomatix.NET Manual*.

[8] Siemens PLM. *eMServer Data Importing (via Process Designer) Student Guide*, January 2008.

[9] Siemens PLM. *Process Simulate Basic Student Guide*, January 2008.

[10] M. Sköldenstam, K. Åkesson, and M. Fabian. Modeling of discrete event systems using finite automata with variables. In *Proceedings of the 46th IEEE Conference on Decision and Control*, New Orleans, LA, USA, December 2007.

A Program Structure

The structure of the classes created in Visual Studio 2005 in C# is shown in Figure A.1. The class GenerateModelCommand implemented *TxButtonCommand* and contained the button command explained in Section 3.1.1. This class had a method *Execute* used in order to specify actions that should happen when a button is clicked. Two properties of the class, *Name* and *Category*, were used where the first was the name of the command button visible to a user, and the second the category from where the command was located at in the Customize window for the menu bar, see Figure A.2. In order to integrate the command

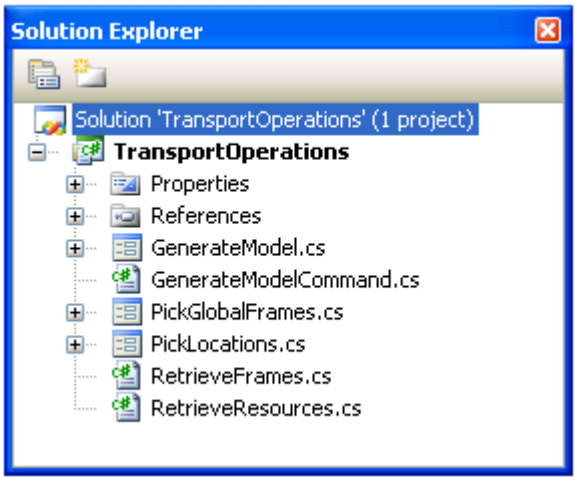


Figure A.1: The classes written in C# in order to create a plug-in and GUI.

into Customize the procedure given in B had to be followed. In the program written for this thesis the *Execute* method started the application GenerateModel which was the GUI displayed in Figure 4.2. The *Pick Location* button was related to the class PickGlobalFrames and the *Resources* button was related to the class PickLocations. The purpose of the two other classes shown in Figure A.1, i.e. RetrieveFrames and RetrieveResources, was to collect information regarding the frames respective resources in the loaded cell in Process Simulate. Information which was used in the other classes.

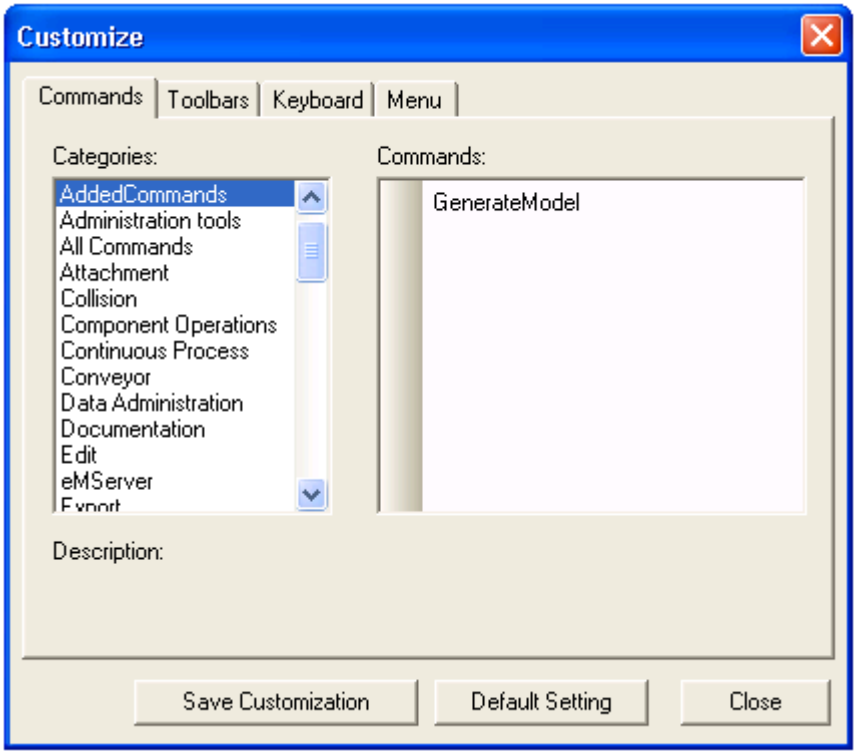


Figure A.2: The customize window in Process Simulate where it is possible to add commands to the menu bar.

B Integrate Command Into Process Simulate

This section will describe the steps that need to be completed in order to have a command button integrated into the menu bar in Process Simulate. The steps are:

- Create a reference to Tecnomatix.Engineering.dll in a project in Visual Studio 2005 C#.
- Create a public class that implements *TxButtonCommand*, e.g. the class GenerateModelCommand described in Section A.
- In the project folder, find the .dll of the project and place it in the Tecnomatix <*Installation dir*>\DotNetCommands\. In order to register the command an application called CommandReg.exe in the Tecnomatix <*Installation dir*> has to be started.
- Open the application. Figure B.1 shows the window displayed when running CommandReg.

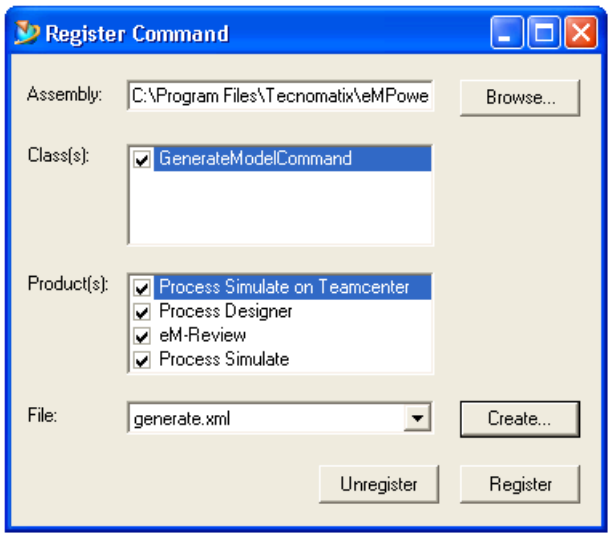


Figure B.1: The CommandReg application window.

Browse to the assembly that was placed in the Tecnomatix <*Installation dir*>\ DotNetCommands\. If there are several classes which implements *TxButtonCommand*, all those classes will be displayed in the Class(s) box. Choose classes with commands that should be registered. In the next box it is possible to choose what products the command is supposted to be integrated to. In the last box it is possible to either choose an existing .xml file or to create a new using the *Create* button to save the command to. When *Register* is clicked a message should appear informing that the registration of the command succeeded.