Final Report

Modelling a production cell using the CAMOS language for manufacturing control

Jörg Diederich

LITH-IDA-Ex 97/46

1997-06-05

Abstract

The manufacturing control system CAMOS (Control Application Mediator Object System) solves the data management problems in production processes by using an embedded active object-relational database management system. This thesis investigates the usability of CAMOS for modelling and controlling a production cell of moderate complexity, which was taken from a case study of the Forschungszentrum Informatik (FZI), University of Karlsruhe, Germany. Operating the production cell takes care on safety requirements to avoid damage of machinery and injury of staff. Furthermore, this thesis presents a compiler for the task level programming language, CAMOS(L) and a detailed description of the language. It defines different techniques how to model machines and their behaviour using CAMOS(L).

Acknowledgements

This thesis came into being during my stay abroad in Linköping, Sweden, in the spring term 1997. This stay in Sweden was supported by ERASMUS, the European Community Action Scheme for the Mobility of University Students, both financially and organizationally. Therefore, I want to thank all people from ERASMUS for supporting my stay at Linköping University.

Furthermore, "Tack så mycket" to my supervisor, Esa Falkenroth, for the idea of this thesis, for the uncountable pieces of advice during the genesis of the thesis and for many hours, discussing about it. Also, a big "tack" to Peter Loborg for his simple solutions to many complicated questions.

Finally, I want to thank all the people, which made my stay in Linköping more comfortable, especially the ESN members for the social activities and my peer student Martin for the big help during the first weeks.

Jörg Diederich
Linköping, June 1997

Contents

Chapter 1

Introduction

This thesis investigates the advantages and disadvantages of the manufacturing control system CAMOS (Control Application Mediator Object System) when using it for modelling and controlling a production cell.

Motivation for Manufacturing Control Systems

The requirements on a todays manufacture are different from those twenty years ago. Initially, computers were introduced in the production process to control individual machinery in manufacturing plants, for example a single robot. These single parts of the machinery became more flexible, because the control software could be changed fast. Now there is also a need for a greater flexibility for controlling the whole production process, as the products change faster and therefore the behaviour of the machinery in the production process has to change faster as well.

Manufacturing Control Systems (MCS) realize this flexibility in controlling all machinery of a production process. Having the whole behaviour of the machinery written in software, they also provide an easier maintenance. Controlling machines automatically by computers introduces also a risk: For example, a moving crane cannot perceive if it collides with another machine or injures human staff. Thus, MCSs must also take care on safety requirements.

Other problems such as the large amount of information in MCSs and how to keep this information consistent, are still under research.

CAMOS - a prototype of an MCS

The aim of developing the CAMOS prototype at Linköping University, Sweden, was to investigate how to implement an MCS using next generation object-oriented database technology to cope with the information problems.

The thesis

As the CAMOS system had not been tested before, the task was to use CAMOS for modelling and running an example from the real world and to discuss the usability of CAMOS for real manufacturing plants.

The example has its origin in a case study of the FZI (Forschungszentrum Informatik) at the University of Karlsruhe, Germany [Lew94]. It is a production cell that forges metal blanks in a press, which is loaded and unloaded by a two-armed robot. The FZI provides also a graphical computer simulation of that production cell to verify if the modelled system works.

As there were different possibilities to model the production cell, this thesis includes a discussion about the different modelling techniques as well.

Hypothesis

The hypothesis is that the principles of modelling and controlling a production cell, used in CAMOS, fulfill safety and flexibility requirements.

Restrictions

The focus of this thesis lay mainly on the general usability of CAMOS, as it is the first attempt to use CAMOS for a moderately complex piece of production. No investigations were made if CAMOS is usable for whole production processes. Proofs of the requirements and formal development of the control software mentioned in the case study were omitted as well.

1.1 The genesis of this thesis

After the introduction into the used systems and tools such as AMOS and CAMOS, I wrote a compiler for the proposed language for manufacturing control, CAMOS(L). This included defining the exact syntax of CAMOS(L).

The next task was testing the provided elements of CAMOS(L). I invented some small examples for trying the class modelling facilities such as inheritance and composition. To check the basic functionality of CAMOS, I wrote some operation examples in order to test the primitive suboperations and the operation calls. This included enhancing CAMOS to cope with simultaneous updates and composite objects. The communication protocol between the database and the control system had to be extended to deal with more than one object instance of the same class.

Then I studied the different synchronization techniques for operations (wait conditions and if guards). This included a first test of database queries within CAMOS(L).

After becoming familiar with CAMOS and the programming of CAMOS(L), I modelled the classes and the general behaviour of the production cell. The first test suite for the classes contained some simple, sequentialized operations without taking care of safety requirements.

The final model of the production cell used complex operations, taking care of all the safety requirements and some efficiency requirements.

1.2 The reader's guide to this thesis

This chapter gives a brief introduction about MCSs and CAMOS, followed by the description of the task of this thesis and its restrictions. In chapter 2, the case study from the FZI is introduced with the specifications and requirements on the control system. A detailed description of the production cell and its functionality is given, too. Chapter 3 contains the background theory that is important to understand the next chapters. It is recommended to read this chapter before the following, if terms such as world model, DFA and object-orientation are not known. Chapter 4 describes the state of CAMOS before this work. In chapter 5, the task is analyzed and possible solutions are presented. Chapter 6 describes the syntax and semantics of CAMOS(L). Chapter 7 contains an overview on the modelling techniques I found out during my work with CAMOS(L). In Chapter 8, a description of the implementation of the production cell can be found. Chapter 9 summarizes the results of this thesis. In chapter 10, suggestions on future investigation are made. Unresolved problems are also mentioned there.

Chapter 2

Task description: the production cell

As the main task of this thesis was to model and control the production cell with CAMOS, this chapter contains a detailed description of the production cell of the FZI case study. It shows the different parts of the production cell and the requirements in order to control them.

2.1 The production cell

The production cell exists in a metal plant of a factory in Karlsruhe, Germany, so it is an example from the real world. Its purpose is to forge metal blanks in a press.

One aim of the case study was to have a kind of road map for an industrial practitioner, which software systems can control a production cell, fulfilling certain requirements such as safety or flexibility. Using one specific real example should show advantages and disadvantages of each method and make them comparable, so that conclusions on the usability of each system are possible. This report is a further piece of that road map, showing the advantages and disadvantages of CAMOS.

The production cell consists of the following machinery:

- a **press** to forge the metal blanks
- a **robot** with two arms, each with a magnetic gripper:

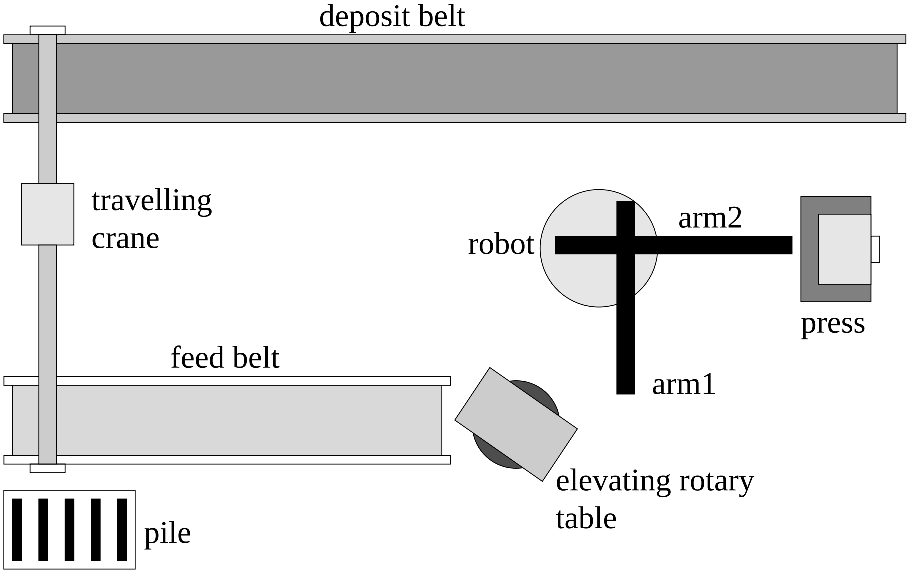

Figure 2.1: The production cell, look from above

  - **arm1** loads the press.
  - **arm2** unloads the press.

  The robot has a rotating base; both robot arms can only move horizontally (extend or retract) and are fixed on the robot base.

- two conveyor belts:
  - a **feed belt** to transport the metal blanks to the robot
  - a **deposit belt** to transport the forged blanks away
- an elevating rotary table (**er-table**) to connect the feed belt with the robot, because the robot arms cannot move vertically
- a **travelling crane** to transport the blanks from the deposit belt to the feed belt

The travelling crane does not belong to the original production cell, but it was added to make the work of the production cell cyclic. Section 2.1.1 about the work of the production cell motivates the introduction of the travelling crane. Furthermore, there is a **pile** with metal blanks at the feed belt.

The detailed description of the production cell can be found in the book about the case study [Lew94, chapter 2].

2.1.1 Description of the work of the production cell

The following enumeration gives a short overview on the basic work of the production cell:

1. A metal blank is put from the pile onto the feed belt.
2. The feed belt transports the blank to the er-table.
3. The table goes in its upper position and rotates clockwise 50 degrees, so that arm1 of the robot can take the blank.
4. The robot rotates, so that arm1 points to the table and picks up the blank. Again, the robot rotates, so that arm1 can deposits the blank in the press, which is in the middle position. After that, the robot turns to a safe position.
5. The press goes up and forges the blank. Afterwards it moves in lower position, so that arm2 of the robot can reach the press.
6. The robot rotates arm2 to the press and picks up the processed blank. Then it rotates to the deposit belt and releases the blank on the belt.
7. The deposit belt transports the blank to the crane and stops there.
8. The crane fetches the blank from the deposit belt and transports it to the feed belt, so that the entire process can start again.

The travelling crane does not belong to the real existing production cell, since it does not make sense to forge a metal blank more than once. For the simulation, the crane provides independence of the production cell against influences from outside. It is not necessary to remove blanks that arrive at the end of the deposit belt or to add new blanks. The five blanks, provided in the simulation, circulate in the simulation until the simulation is stopped.

There is one improvement on the work of the robot, which makes the loading and unloading of the press more efficiently:

At first the robot rotates arm1 to the table in order to take a blank. Then it rotates counterclockwise, until arm2 reaches the press to get the last processed blank out of the press. After that, the robot rotates again counterclockwise, until arm2 reaches the deposit belt to unload the forged blank. Then it rotates counterclockwise, until arm1 reaches the press to be unloaded. Finally, the robot rotates clockwise, until arm1 reaches again the er-table to take another blank.

2.1.2 Controlling the production cell

The production cell consists of 13 actuators (motors and magnets):

- The belts: one motor each for moving the conveyor belts
- The er-table: one motor for elevating and one for rotating the er-table
- The robot arms: one motor for each robot arm to extend or retract the arm and one magnet at each robot arm for gripping blanks
- The robot base: one motor for rotating the robot base
- The press: one motor to open and close the press
- The crane: one magnet for the crane, one motor for moving the crane horizontally and one for elevating the magnet

There are 14 sensors to get information about the state of the machinery.

- The belts: one photoelectric cell at the end of each conveyor belt to measure, if a blank has arrived at the end (light beam of the photocell interrupted) and if it has reached the unload position (light beam not interrupted any longer).
- The er-table: one potentiometer for sensing the rotation and two switches for the upper resp. the lower position.
- The robot: one potentiometer for each arm measuring the extension and one potentiometer measuring the rotation of the robot.
- The press: three switches for the three positions of the press: up, in the middle or down.

- The crane: two switches for the horizontal position (at the feed belt resp. at the deposit belt) and one potentiometer for the elevation of the magnet.

2.1.3 System analysis

Analyzing the production cell results in three statements: It is a reactive system, safety-critical and moderate complex.

The production cell belongs to reactive systems, as the control system must react permanently to changes in the environment. That means also, that it must fulfill certain real-time requirements. For example, the feed belt must stop immediately, when a blank arrives at its end and the er-table is still rotated to the robot. Otherwise, the blank will be dropped at the end of the feed belt.

Then the production cell belongs also to the group of safety-critical systems. Collisions and damage of machinery can occur, if the control software contains errors. Moreover, injury of people is possible, when for example a blank is dropped at the wrong place.

Finally, the system is classified as "moderate complex"; modelling the production cell in one state automaton would lead to not more than 10^{12} different states. That means, this production cell cannot show the behaviour of the controlling system for a whole production process, but it is still large enough to show important advantages and disadvantages.

2.2 Requirements on the software controlling the production cell

The requirements on the control software consist of three parts:

1. Safety properties:
    - Collisions between machines must be avoided.
    - Machines must not move outside their specified operating ranges.
    - Blanks must never be dropped outside designated drop-areas such as above the deposit belt or in the press.

- Blanks must always be sufficiently separated, so that the sensors (e.g. the photocells at the end of both belts) can recognize them as two different objects. It is not allowed to have two blanks on the er-table or in the press as well.

2. Liveness:
   All the blanks inserted in the system will eventually arrive forged at the end of the deposit belt.

3. Flexibility and efficiency:
   It must be easy to enhance the control system in order to use it for similar production cells. Also, small changes in the task of the production cell should not lead to large changes in the control system.

   The time to process a blank in the production cell should be minimal, the throughput of the production cell maximal.

The main focus of the case study lies on the safety requirements, as violation of them can lead to machine damage or injury of human staff. For a detailed description about the safety properties see the task description in [Lew94].

2.3 The graphical simulation

To see, if the control system for the production cell works properly, the FZI provides a graphical simulation programmed in TCL to visualize the work of the production cell. Figure 2.2 shows a screen copy of it.

The simulation was realized with the 13 actuators and 14 sensors as mentioned in section 2.1.2, so that it fits exactly to the specification of the problem. It was programmed for X-Windows and communicates with the control program via UNIX pipes.

2.4 Restrictions for this thesis

The task of this thesis, modelling the production cell with CAMOS, is not exactly the same as the task given in the case study. At first, there are no formal proofs of the methods and no discussion about the liveness. It was not the aim either to program the most efficient solution, but the focus was put

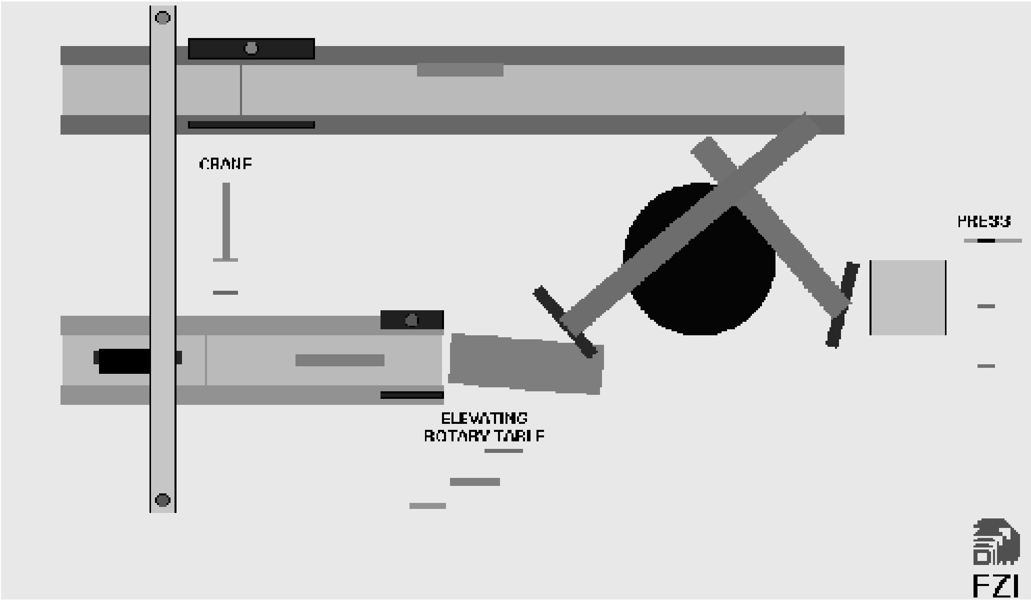

Figure 2.2: The graphical simulation from the FZI

on safety requirements and the usability of CAMOS. Finally, the real-time properties and the reactiveness of the system were guaranteed by moving the real-time kernel into the simulation using guards, provided by the simulation (see section 8.2.1).

Chapter 3

Background and related work

This chapter is about the theory and background information that is necessary to understand the following chapters. Readers familiar with manufacturing control systems, ARAMIS, object relational databases and active databases may continue with reading the next chapter.

3.1 Manufacturing Control Systems (MCS)

Manufacturing control systems (MCS) control parts of or even whole production processes. All data structures and the behaviour of the objects are stored in the MCS. They try to cope with the problems in production processes in a general way, so that they are not limited to a certain production process.

An MCS must fulfill the following properties:

- Support for safety mechanisms to avoid collision of machines or injury of staff
- Increase of the flexibility of the machinery to make changes within the production process easier
- Support for maintenance
- Robust against unexpected events from the environment such as machinery faults, which appear very often in large production processes
- Operate with machines from different manufacturer, which have different user interfaces

- Modularity, so that changing parts of the production process is possible while other parts continue.

- A single and simple interface to all data in the MCS to simplify the work of the production engineers, programming the production process.

3.2 The basics of ARAMIS

The ARAMIS-project (A Robot and Manufacturing Instruction System) implemented a prototype of an MCS continuing a joint research between the Department of Computer Science, Linköping University, and ABB Corporate Research and ABB Robotics in Västerås, Sweden, during the years 1985-1988.

The following sections introduce briefly the basic ideas of ARAMIS, since they were adopted for CAMOS. A detailed description can be found in [Lob94], [LobSkö92] and [LobHol92].

3.2.1 Dividing the problem: A 3-layered architecture

Programming a complex production process is a difficult task because of the large amount of complex information in the production process. This task is two-folded: Programming the general behaviour of the production process and programming the controller of a single machine. Between both an interface can reduce the information about the machines, as programming the general behaviour does not require all the details as needed for programming the controller of a machine.

This separation of programming leads to a three-layered architecture within ARAMIS: The physical level, the control level and the task programming level. It also divides the people, working on the development of the manufacture into three groups: mechanical engineers, control engineers and production engineers. Figure 3.1 shows the architecture of ARAMIS.

The physical level

This level represents the real world in a manufacture, consisting of machines, tools and the materials needed for the production. All information about real world object such as size, weight and material are available in this level. Mechanical engineers use this information to build the real objects for a

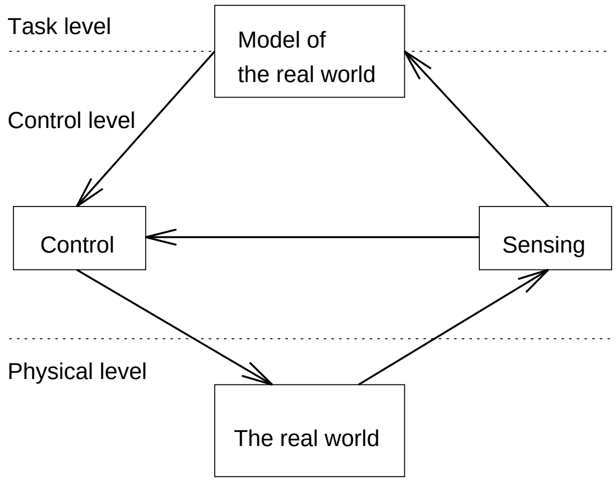

Figure 3.1: The layered architecture of ARAMIS

manufacture (e.g. machines). The physical level does not belong to the ARAMIS system, as there is not any software in this level.

The control level

This level consists of the real-time software that senses the real world, reacts to changes within the real world and gives information about those changes to the task level.

It is important that the software reacts immediately in both directions: to the physical level to keep control over the machines and to the task level to keep the information within the MCS consistent. This level contains the work of control engineers developing traditional control algorithms for the machines.

The task level

On this level production engineers program the general behaviour of the production process with work descriptions and operations in a high-level task-oriented language. This language is especially designed for task level programming and easy to use, so that the production engineers do not need to be experts in programming.

Interfacing the levels

Between these three levels, interfaces give information to the next higher level. They abstract from unnecessary details to reduce the large amount of information in an MCS.

Sensors

Sensors are the interface between the physical and the control level, so that on the control level the machines are only represented by sensor information such as 'motor on' or 'rotation of the table: 55 degrees'. On the one hand, the mechanical engineer, designing the sensors, must take care about providing all information that the control engineer needs to program the control software. On the other hand the designer should not provide unnecessary details in order to keep the amount of information small.

The world model

The **world model** (WM) is the interface between the control level and the task level. It stores all information, provided by the control level, and gives the task level programmer an abstract and single view of the real world. This view presents the real world as a set of **world objects** (WO), which is the abstract view of one single real world object (e.g. a robot).

Introducing discrete states for each WO reduces the number of possible states for a WO. Giving each WO a set of state variables, which have a limited set of allowed values (e.g. an enumeration or a range of integer values), provides these discrete states. Again, the control engineer has to think carefully about two aspects: Providing sufficient information for the task level programmer and reducing the possible state space of the WOs by abstracting from unnecessary details.

The following example shows how an object of the real world can be abstracted as a WO:

A conveyor belt has the possibility to move forward and backward with a speed between zero and 10 meters per second. There are two sensors providing information about the state of the belt to the control level: a switch, which gives the direction of the movement (forward or backward) and one sensor measuring the speed as between 0 and 10.000 mm per second. The task level programmer tells the control engineer that he needs only two forward speeds (10 m/s called high speed and 5 m/s called low speed) and one backward speed (5 m/s, also called low speed). Now the control engineer has different alternatives to create a world object of the conveyor belt:

- mapping both variables directly to the world model
- reducing the speed variable to three states (zero, one and two) and mapping the direction directly to the world model
- creating a new, single state variable, summarizing the states of both state variables together

The first alternative exports $10000 * 2 = 20000$ different states to the task level. It has most states and provides unnecessary details, since the task level programmer needs only three speed levels.

The second alternative reduces the possible state space to $3 * 2 = 6$ and is a good solution according to the demands of the task level programmer. It provides also the possibility to operate the belt in backward direction with high speed.

Finally, the third alternative reduces the state space to four states (high speed forward, low speed forward, stop, low speed backward). It summarizes the values of two real sensors together into one state variable. This summary of the values of several sensors is called creating a **virtual sensor** on the control level.

More examples on how to model world objects can be found in [Lob94, section 2.6].

Summary

Dividing the architecture of an MCS into three levels improves the safety and the flexibility of an MCS. Specialists are available for each level, who know much about safety aspects within their field. Changes within one layer remain local as long as the interface to the other levels does not change.

This architecture solves also the problem of the large amount of information by introducing two interfaces between the three layers as abstraction barriers. Each interface provides only the information, needed by the next layer and abstract from unnecessary details.

3.2.2 Designing the basic behaviour of a world object: DFAs

Having designed the interface between control level and the task level with state variables, a **deterministic finite state automaton** (DFA) uses these state variables to model the basic behaviour of the WO. A **node** of the DFA represents one or more states of the WO, that means a set of possible values for the state variables of the WO. The **edges** or **transitions** between the nodes model the behaviour of the WO: if a transition connects two nodes, the direct state transition between both states is allowed. If there is no edge, the transition is not allowed and the WO refuses to change its state.

Back to the conveyor belt example, which has the following behaviour: When stopped, the belt can move forward or backward with low speed. When moving with low speed, the belt can slow down to stop or when moving forward speed up to high speed. From moving forward with high speed, the belt is allowed to slow down to low speed and for emergency cases also to stop immediately. The belt must not move with high speed directly after having stopped and not switch from moving forward to backward without stopping in between.

The DFA for the modelling with two state variables is shown in figure 3.2.

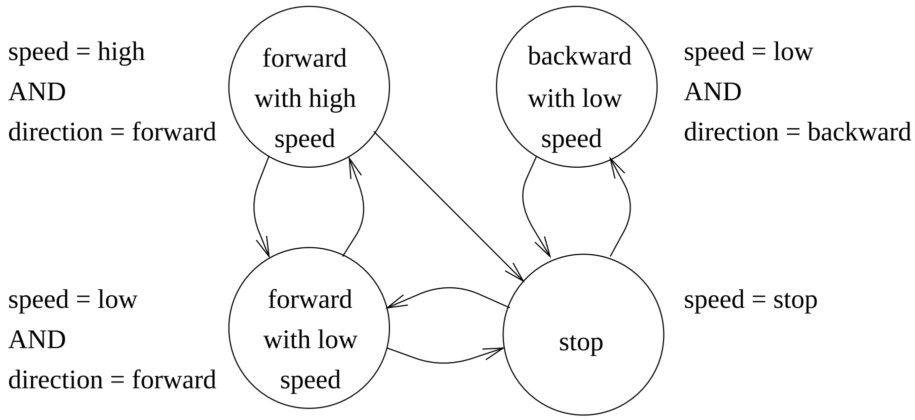

Figure 3.2: The DFA of the conveyor belt with two state variables

Thus, programming the basic behaviour of a world object with DFAs, provides safety methods within the design of the WO. Having omitted a transition between two states in the DFA, makes it impossible in the task level programming to invoke this transition.

3.2.3 Connecting control level and task level: The virtual machine

As the WM is the interface for the task level to get information from the control level, there must be another interface to tell the control level about changes that should be done in the real world. Therefore, there is a **virtual machine** (VM). It is a structural copy of the WM, that means it simply doubles all state variables for all WOs. As the state variables in the WM report about the real state of the WO, the state variables in the VM represent the state, the task level would like to change the WO to. To differ between those two groups of state variables those in the WM will be called actual variables and those in the VM reference variables.

So if the task level wants to invoke a change in the real world, it changes the value of one or more reference variables.

The task of the control level is now not only keeping the WM in the real world state but also guarding the VM about differences to the world model. When having found a change, the control level interprets the change according to the DFA in the WM. That means, it looks up if a transition connects the old state and the new state in the DFA of the WO. If it exists, the control level executes a control algorithm bound to the connecting transition. After the execution the WM is updated according to the new state.

3.2.4 The stability problem

The execution of a control algorithm takes a certain period of time to end successfully. During this time, the VM and the WM remain in an inconsistent state; the VM already contains the new state of the object after the hopefully successful execution of the control algorithm, while the WM still contains the old value. This introduces a new property to all WOs: **stability**. Stability of a WO means that both the WM and the VM contain the same values for the corresponding reference and actual variables. When a WO is instable, the only knowledge about the WO is, that it is somewhere between the both states. So neither the reference nor the actual variable contain the right values and should not be read.

The conveyor belt example should again clarify the problem: The initial state of the belt may be the stop state. Then the task level directs the belt to go forward with low speed and sets the state variables in the VM to the corresponding values. Now the control level perceives the difference between the VM and the WM and executes a control algorithm to speed up the belt. As long as the belt has not reached the low speed, that means five meters per second, the control level will not update the WM. Thus, the reference value and the actual value remain in different states, the WO of the belt is instable. Accessing in this situation the state variables in the VM or the WM would lead to wrong values. The only statement about the state, which can be made, is that the belt is moving forward with a speed between zero and five meters per second.

In general, a WO should *not* be accessed, when it is instable.

3.2.5 The language on the task level

Why having a new programming language on the task level at all? There are many well working multi-purpose languages, but taking such a programming language would introduce too many and unnecessary constructs. Programming the task level would become more complex for the production engineer, who is not a specialist in programming. Thus, the task level language must be easy to use.

The task level language must also support dynamically changes within the machinery, so that the program can run even if some machines in the process are repaired or changed. A conventional language must recompile the program and restart it again.

Furthermore, errors in the machinery or even in the task level program should not lead to a halt of the whole system. It must be possible to rewrite parts of the program code while the other parts are still working. This is very important, because changes in the environment, small improvements or changes of broken machinery, is common in large production processes.

3.2.6 Relation to this work

CAMOS is based on the same three-layered architecture and uses the world model concept from ARAMIS. DFAs model the basic behaviour of a WO using the discrete states provided by the state variables in the WM. The task level language of CAMOS, CAMOS(L), does not contain all the above mentioned requirements on a task level language, but its general concepts allow to introduce the missing requirements.

3.3 Object-oriented databases: Basic ideas

Fundamental knowledge about relational databases is required to understand this section about object-oriented databases. For information about the principles of tables, keys and so on see a book on basics of databases, e.g. [Elm89].

This section focuses on the object-oriented concepts used within CAMOS. For a more detailed description, see an introduction in object-oriented databases, e.g. [Cat91].

3.3.1 Value vs. object orientation

A table is the basic concept of traditional relational databases. One real world object is modeled through one line in one (or more) table. To recognize one entry in the table one or more columns must be declared as keys. That means each value of the key is only allowed to appear once within that table. This concept is called value-oriented, because a real world object, represented through a set of values, is identical to another real world object, represented through the same set of values, although both are two existing objects.

In the object-oriented approach, the system provides a mechanism so that each created object has a unique object-identity, which is hidden from the user. The user of the database does not need to care about how this identity looks like and can assume that two created objects that have identical attributes and attribute values, are not the same.

3.3.2 Type hierarchy

The only "structured" datatype in traditional relational databases is the table having rows and columns. This has been a good solution in order to solve problems with a large amount of data from the same type and structure.

As soon as there is complex data, the time for developing a system using the relational datamodel increases and maintenance or even small changes within the datamodel are time-consuming. This was the motivation to introduce a type hierarchy including classes, encapsulation, methods, aggregation and generalization.

Classes

Classes within the object-oriented databases correspond to tables within the relational databases. Both give the possibility to summarize objects of the same structure together in one data structure. Having a certain number of lines within a table corresponds to having the same number of objects of one class created. An object, which has been created in the database from a certain class, is called **instance** of this class. The difference between instances and classes is that instances are really existent in the database whereas classes are only templates in order to create instances.

So modelling a class means creating a template for a group of objects with the same attributes, differing only in the values of the attributes.

Encapsulation

As the object-oriented paradigm is derived from the abstract data-type concept, one main idea is to keep as much information stored within an object private. That means that only the object itself can access to this information. Access from outside the object is only possible, if the object explicitly allows it. For example, this access could be limited to a write-only access.

The functionality for this object is provided through procedures, which are bound to the object and operate on its private data. These functions are called **methods**. Binding the methods to the object, which the methods change, is the idea of keeping the data and the operations on this data local.

The designer of a class also has to care about the interface to outside the class. The class should have full functionality, but the access to the data of the class should be limited as much as possible. The reason for this limitation is the reuse of classes. When having written a class and designed it in one piece, it is probably a good implementation. But enhancing the class with some more functionality and more data enlarges the possibility of introducing errors. So it is better not to change the class but use generalization instead, which is described in the next section.

Thus, keeping data private and binding methods, providing the functionality of the class, directly to the class is called encapsulation.

Generalization

Now follows a concept to deal with objects with similar, but not the same structure, called generalization (also inheritance, subtyping or subclassing).

It introduces the possibility to reuse former defined classes for the creation of new classes. The new class is called **subclass** from the reused one and the reused class is then called **superclass** for the subclass. By inheriting all the attributes and methods, the new class gets functionality without programming any code. And then new attributes and new methods can be added to the new class to enhance the functionality.

It is also possible to have more than one class as superclass to create a new class. This is **multiple inheritance**.

The generalization provides a better solution as only enhancing the old class, as mentioned in the encapsulation section, since the old class and its methods are reused without changes. Possible appearing errors can only be made in the derived class or its methods.

Overloading

Sometimes a method of a superclass needs to have another functionality in one of the subclasses. Then a new method with the same name, but belonging to the subclass, can overwrite the method of the superclass. The old method is still available for the superclass, but not for the subclass. This is the **overloading** of a method.

Example:

A vehicle simulator should explain the above object-oriented concepts. This simulator shall work with different types of cars, so the class hierarchy may look like in figure 3.3.

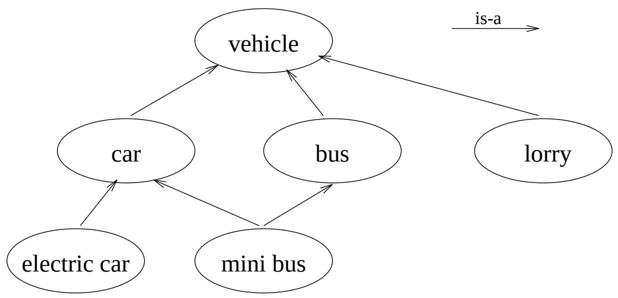

Figure 3.3: A class hierarchy for the example vehicle simulator

The superclass for all classes is the `vehicle` class, which has all common properties of vehicles, for example four wheels, a steering and a motor. Then the methods to operate on those common data must be created such as going right, going left, start the motor, accelerate and so on. For those operations the simulator should work well.

Now, the simulator should also deal with more special vehicles such as lorries, busses or cars. So the classes for these vehicles are derived from the vehicle class, reusing its methods such as starting the motor, because this functionality does not change. These derived classes still are vehicles, but they have also new properties and a new functionality, both bound private. For example, a bus can take passengers, which have to have a ticket to come on board; a lorry has no passengers but two drivers and it can transport goods. Methods could be “control the passengers” for the bus or “unload the goods” for the lorry.

Another special vehicle is the mini bus, multiple inherited from the car and the bus class. It may take passengers, so it has the properties from the bus. It is not necessary to have a bus driving license, but a car driving license. So it has some properties from the car.

The “electric car” class may now use overloading of the “start motor” method: This is necessary, because an electric motor has to be started in another way than a normal combustion motor.

Summary

Thus, creating an object-oriented type hierarchy provides a higher flexibility through reusing of code. Through encapsulation, it also avoids that enhancements of an existing system lead to an increasing number of errors.

3.3.3 Object hierarchy: Aggregation

Type hierarchy methods cannot model all problems from the real world. For example, a car, having two doors, cannot inherit the door class twice. So this class has to take instances of other classes as attributes. This grouping of objects within a class is called **aggregation** and an instantiated object from the class is called **composite object**.

3.3.4 Relation to MCS

Until now there has been nothing said about how the WM data should be stored. As the ARAMIS approach takes an object-oriented view of the real world through the WOs, an object-oriented database stores the WMs data. The design of WOs need not care about the object identity, as mentioned above. Using the possibilities of type hierarchy makes the designing of WOs easier as parts of former designed classes can be reused, and more flexible, when changes have to be done.

3.4 Active Database Management Systems

Active database management systems (ADBMS) were created, when new applications of databases (e.g. MCS) needed a possibility that the database itself can change data within the database. Non-active databases are only passive data repositories, the use must invoke database operations such as insertions, deletions or update of data on his own.

ADBMS introduce a new sort of database concept called active rules. These rules monitor the state of the data in the database and automatically execute one or more user-programmed database operation, when a user-programmed condition is satisfied.

For a detailed introduction in ADBMS see [WidCer96].

3.4.1 The active database system AMOS

The active mediator object system (AMOS) is an object relational database system, developed at the Engineering Databases and Systems Laboratory (EDSLAB), Linköping University, Sweden. It provides mechanisms in order to create classes with object-oriented techniques (e.g. generalization, aggregation) as described in section 3.3 and is a single-threaded multi-user database. The data is held completely in the main memory, so that AMOS is fast enough to cope with the data management problems in complex environments. It also provides active rules as described above and a powerful, relationally complete query language AMOSQL, which was derived from the OSQL dialect, an object-oriented enhancement of SQL.

As it is not necessary to know the details of AMOSQL to understand this thesis, it is left to the interested reader to look in [Kar94] for a detailed description.

3.5 Related work

Much work has been done in the field of MCS during the last years. As my work of this thesis was restricted to the given task, only a few other works crossed my way. Nevertheless, they are briefly mentioned in this section.

3.5.1 ARAMIS

As described in section 3.2 CAMOS adopts the basic principles of ARAMIS such as the world model paradigm. But it contains much more. At first, the control level was fully implemented on a multi-threaded Forth and connected to a robot. So ARAMIS contains much work on real-time properties, which the control level must fulfill. And second, the possibilities for using error recovery techniques in ARAMIS were investigated [Lob94] to show, how the data in an MCS can be used for fast recovery of the production, if an error occurs.

3.5.2 The case study

The main aim of the case study of the FZI [Lew94] was to show, how formal software development can be used in safety- and time-critical systems in the industrial practice. It recommends the use of mathematical methods to prove the safety of the implementation of the production cell.

As the time-critical parts of the systems are within the control level, most of the work of the methods in the case study is done in this level. Therefore, this thesis is not compared to the case study.

Chapter 4

The CAMOS system

4.1 Introduction

CAMOS (Control Application Mediator Object System) is a prototype for an MCS that uses the embedded active object-relational database AMOS to handle the complex data in MCSs. CAMOS was developed at the Real-time Systems Laboratory (RTSLAB), Linköping University, Sweden, as part of a licentiate thesis [Fal96]. Basing on the same principles as ARAMIS (see section 3.2), CAMOS includes the advantages of an ADBMS into the ARAMIS approach. It also introduces a new task level language, CAMOS(L).

4.2 The architecture of CAMOS

CAMOS adopts the three-layer architecture from ARAMIS. Additionally, it uses the AMOS database to store all information about the production process and active rules to connect the database with the control level.

The layered architecture (see section 3.2.1) uses the WM to store the abstracted data from the control level (see section 3.2.1). The VM interfaces the task level and the control level (see section 3.2.3) and DFAs (see section 3.2.2) model the basic behaviour of the WOs. The control level guards and controls the real world. If there is a request for a change in the real world within the virtual machine, it executes one or more control algorithms to change the real world. After the execution, the control level sends an acknowledge message back to keep the WM consistent to the real world.

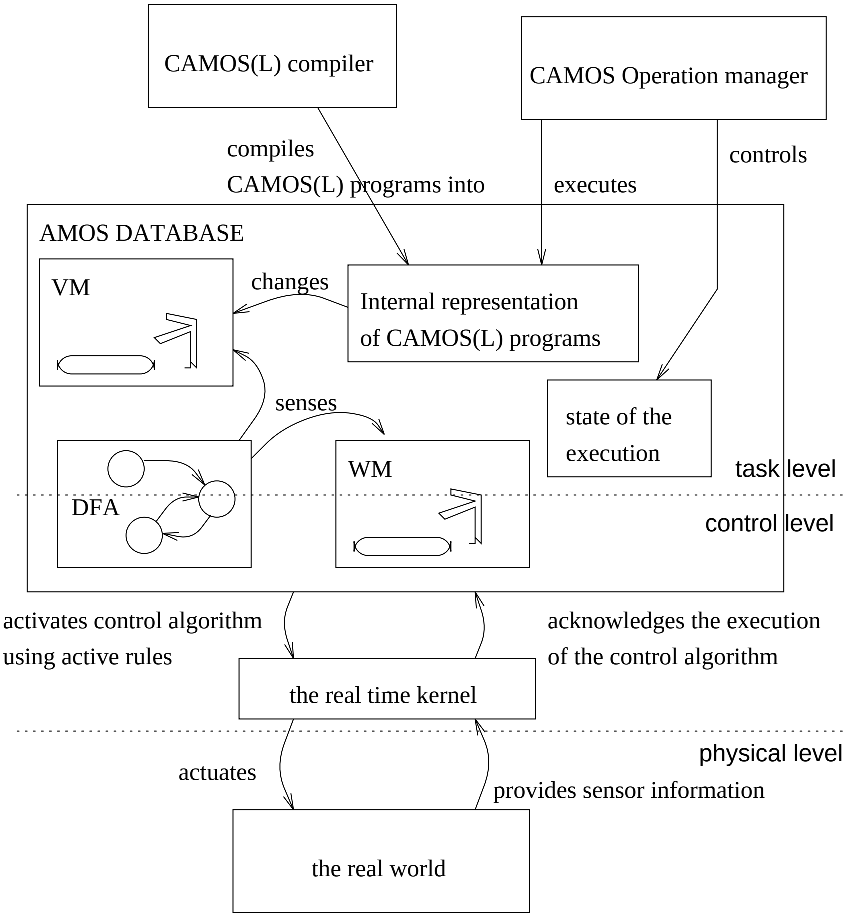

Figure 4.1: A simplified overview on the CAMOS architecture

Figure 4.1 shows now the CAMOS architecture. Basically, the differences to the ARAMIS architecture are two-folded:

1. the integration of a database to store the WM, the VM, the DFAs, the operations and the current execution status: the architecture of CAMOS is database centered.

2. the connection between the task level and the control level using active rules

The following sections describe the four main elements of the CAMOS architecture: the database AMOS, the operation manager, the real-time kernel and the task level language CAMOS(L).

4.2.1 WM stored by the embedded AMOS

As the CAMOS architecture is database centered, all elements of the architecture above the control level deal with the database.

AMOS stores all abstracted information about the real world, that means the WM and the VM (see section 3.2.1, 3.2.3). The object-oriented facilities of AMOSQL support modelling the data structures for the WOs. The DFAs, describing the basic behaviour of WOs (see section 3.2.2), are stored in the database as well, using active rules to model the transitions.

4.2.2 The operation manager

The operation manager was written in the internal LISP language of AMOS to have a tight and therefore fast access to the database. It organizes the execution of the operations that originally were defined in CAMOS(L) but later compiled into an internal representation of operations. CAMOS stores the operations and their current execution status in the database as well, so that even knowledge can be gained about, for example why an operation caused an error. To handle dependencies between operations and suboperations, the operation manager administrates both in different queues according to their status. The principles of the operation manager where mainly adopted from ARAMIS; but the code is much compacter, because the database provides all mechanisms for data management.

The following paragraphs describe briefly the three parts of the operation manager: the interpreter, the scheduler and the rule coordinator. For a detailed description, see [Fal96, section 4.7].

The interpreter

CAMOS(L) programs are compiled into runtime code, database queries and active rules. The interpreter reads the runtime code, containing the suboperations, their guards and their dependencies. It executes the suboperations according to their execution order. It also performs changes in the VM, requested by the primitive suboperations, and works up the acknowledge messages from the real-time kernel.

The scheduler

As the AMOS database is single-threaded, the scheduler has to decide, which operation is allowed to execute, if there is more than one operation to execute at the same time. It is a round-robin scheduler.

The rule coordinator

The rule coordinator activates and deactivates operations according to their wait condition, specified in the CAMOS(L) program. This corresponds to activation and deactivation of active rules, as they are used to implement the wait conditions of operations.

4.2.3 The real-time kernel

This part of CAMOS is implementation depended, because it is the connection to the real machinery. Its purpose is to handle the sensors and actuators, that means initiate changes in the physical level, and to send acknowledge messages to the upper layer, when these changes are done.

For the case study, the real-time kernel is included in the simulation and so executed on the same machine in a separate process. For further details see the description of the simulation in section 2.3 and programming the real-time kernel in section 8.2. The CAMOS system itself does not contain real-time functionality.

4.2.4 The propose for CAMOS(L)

CAMOS(L) is the task level language within CAMOS. It is a successor to the task level language, used in ARAMIS and adopts the principles from them (see also [Lob94] 4.3.1). They are also described in the following section and illustrated through many examples in chapter 6 and 7.

The CAMOS(L) compiler compiles CAMOS(L) programs into database queries, active rules and runtime code, which are then stored in the database and executed by the operation manager. For a detailed description of CAMOS(L) see the following chapter.

Basic ideas of CAMOS(L)

The aims of CAMOS(L) are three-folded:

1. providing mechanisms to coordinate operations in a production process on the task level

2. supporting the data management for the operations

3. supplying the engineers, designing and controlling the production process, with a specialized and easy language, having only few syntactical constructs

The first thing to do when programming a production process on the task level, is to design the WOs and the DFAs for the basic behaviour of the WOs (see section 3.2.1 and section 3.2.2). Then the objects must interact, that means programming the behaviour *between* the different WOs. This is done by creating **operations**. Operations describe the behaviour between WOs on a higher level as DFAs using different synchronization techniques.

The basic structure of operations

An operation has at first a condition, which must be satisfied, before it is allowed to execute. This condition is called the operation's wait condition. Its purpose is to synchronize the operation with other operations.

An operation consists of several suboperations, grouped into guarded sets. Such a guard, a database query returning a boolean value, decides if the guarded set of suboperations is executed or not.

Then there are two different types of suboperations:

- calls to another operation (operation call)

- requests to change the real world (called primitive suboperations).

Basically, all suboperations within one operation may execute in parallel. To serialize suboperations that must not be executed in parallel, dependencies give a possibility to order suboperations.

When all the suboperations with fulfilled guards have ended, the operation ends also, unless it is an iterative operation. This special operation restarts again with checking the wait condition and then executing again all the suboperations with fulfilled conditions.

The proposed syntax of CAMOS(L)

In the proposal for CAMOS(L) [Fal96], there is an example for an operation:

```
CREATE OPERATION
  getPressedFetchNew (robot r, e_r_table t, press p)
AS WHEN
  partsAt(p) AND position(p) = "open"
    AND palletCapacity (currentPallet()) >=
    COUNT ((SELECT b FOR EACH part b
            WHERE position(b) = currentPallet()))
IF NOTANY(partsAt(upperConveyour())) THEN
  1: getPressedPart (r,p)
IF SOME((SELECT b FOR EACH part b
         WHERE name (position (b)) = "e_r_table")) THEN
  2a: getNewPart (r,t);
  2b: SET referenceElevation (t) = 0;
  DEPENDENCIES: (2a, 2b);
IF SOME(partsAt (t)) AND SOME (partsAt(p)) THEN
  DEPENDENCIES: (1, 2a);
```

This creates an operation `getPressedFetchNew` that takes three objects as actual parameters. The wait condition is the database query between the `AS WHEN` and the first `IF` keyword. There are three guarded sets of suboperations, the first containing one suboperation call, the second containing a suboperation call and a primitive suboperation, which are serialized, and the third contains only a dependency for the case, that the first two sets of suboperations are executed.

This operation describes one action between the er-table, the robot and the press in the production cell. Its purpose is to fetch a metal blank from the er-table and to move a forged blank from the press to the deposit belt simultaneously.

As the semantic of this CAMOS(L) operation remains the same in the implementation of the CAMOS(L) compiler, done in this thesis, the detailed description of the single parts of this operation is skipped here. See chapter 6 for a complete description of the syntax and semantics of CAMOS(L) and section 6.1.7 for the changes in the syntax.

4.3 The basic functionality of CAMOS

Having described the basics for CAMOS(L), the execution of a primitive suboperation should explain the work of CAMOS, especially the part, active rules play in the communication between the real-time kernel and the WM.

When the interpreter reaches a primitive suboperation during the execution of an operation, it changes the state variable of the WO in the VM to the wished value. As AMOS is an ADBMS, active rules model the transition within the DFA. Such an active rule triggers, when the node constraint of the target node is fulfilled and the WO has been in the source node before. In this way, the active rules, modelling the transitions, senses the VM and the WM and trigger, when there is a difference between both.

The triggering of a transition executes a procedure, which sends a control algorithm to the real-time kernel. When the execution of the control algorithm has stopped (that means the real world has changed to another state), the control level sends a message back to the database to acknowledge the change of the real world. Now the interpreter updates the WM in the database, so that the difference between WM and VM disappears.

4.4 Summary

To summarize the results of the integration of AMOS in CAMOS, the following list shows the main advantages:

- Using AMOSQL, the query language of AMOS provides a mechanism to access the database from within CAMOS(L)

- Active rules connect the database and the control level sensing automatically the differences between the WM and the VM, so that the programmer do not need to care about, how to send control algorithms to the real-time kernel.

- Using the rollback mechanism of the database gives the possibility to go back in the execution of the task level program, when an error has occurred.

- Having all data and information not only about the WM, but also about the state of execution within the database provides a uniform access to all information with the same query language. That means, the integration of AMOS in CAMOS provides a homogeneous programming environment, although the information stored in the database is heterogeneous.

The use of active rules for communication between the database and the control level may introduce problems such as rule interaction, rule cycles or cascading rule triggering. As these problems are solved within CAMOS, there is no further discussion about them. Details can be found in [Fal96].

Chapter 5

Task analysis

This chapter contains at first an analysis of the state of the CAMOS system before the beginning of this work. After that, an analysis of the problems given through the task follows and a discussion about the possible methods to solve the problems.

5.1 The current state of CAMOS

The problem on CAMOS was that it was discussed theoretically and only tested with a very small example in the internal language of CAMOS (the er-table from the FZI case study); it has never been tested on a more complex system. So there were no statements on the usability of CAMOS, not even for an example of moderate size.

In addition to that, CAMOS(L) existed only in theory. So the mechanisms of this language had never been checked in practice for their usability. Also, the detailed syntax for CAMOS(L) did not exist, only one example (see section 4.2.4), given in [Fal96].

5.2 Partitioning the task

The task of this thesis was to model and program the production cell, taken from the FZI case study, with CAMOS. The first choice to do this was using the proposed CAMOS(L) language or the internal representation. Secondly, it had to be decided to enhance CAMOS or to use the current state of it. Third, the different alternatives for modelling in CAMOS(L) were examined for their applicability. Finally, the different alternatives to model the production cell were discussed.

Internal representation vs. CAMOS(L)

There were two ways to program the production cell: using the internal representation of CAMOS or the proposed language CAMOS(L).

The following example shows the internal representation of the operation `moveToEnd`:

```
/* The argumentaccessfunctions for the operation MOVETOEND */
create function MOVETOENDarg1 () -> BELT as
  select obj for each BELT obj
    where obj = args (1, current ());
create function wcMOVETOENDarg1 (wcmon id) -> BELT as
  select obj for each BELT obj
    where obj = args (1, activation (id));
create function parentMOVETOENDarg1 () -> BELT as
  select obj for each BELT obj
    where obj = args (1, caller (current ()));
create function MOVETOENDlabel1arg () -> BELT as
  select parentMOVETOENDarg1 ();

/* The rules for the if-clauses */
create function ruleMOVETOEND1 () -> boolean as
  select where  true ;

/* The operation itself and the accessfunctions */
create operation (name) instances :MOVETOEND ("MOVETOEND");
create function MOVETOEND () -> operation;
set MOVETOEND () = :MOVETOEND;

 /* Some helpfunctions to keep the waitcondition short !! */
create function MOVETOENDwchelp1 (wcmon id) -> boolean as
  select true where
  loadedbeg (wcMOVETOENDarg1 (id)) = 1 and
  loadedend (wcMOVETOENDarg1 (id)) = 0
;
```

```
create function MOVETOENDwchelp2 (wcmon id) -> boolean as
  select true where MOVETOENDwchelp1 (id) and
  stable (wcMOVETOENDarg1 (id));

/* The waitcondition for the operation */
create rule MOVETOENDWCRULE (wcmon id) as
  when active (id) and MOVETOENDwchelp2 (id) do wakeUp (id);
set wcRule (:MOVETOEND) = getrule ("MOVETOENDWCRULE");

/* The operation rules for operation MOVETOEND*/
set rules (:MOVETOEND) =
 [["MOVETOENDif1", getfunction (".ruleMOVETOEND1"),
[1, [:PUT, getfunction (".MOVETOENDlabel1arg"),
           [getfunction ("LOADEDBEG"), 0],
            [getfunction ("LOADEDEND"), 1]]]]];
```

Although this is computer generated code, which would be compacter, if written by hand, this shows, that it is not convenient to program directly the internal representation of CAMOS(L).

The CAMOS(L) code for this example operation looks like this:

```
CREATE OPERATION moveToEnd (belt .b)
AS WHEN
  loadedbeg (.b) = 1 AND loadedend (.b) = 0
DO
IF true THEN
    1: PUT .b: loadedbeg = 0,
               loadedend = 1;
ENDIF
```

The `moveToEnd` operation moves a metal blank, which was put onto a conveyor belt, to the end of the belt if there is not already a blank at the end. Section 8.1.1 describes the work of this operation more detailed.

The CAMOS(L) code is much compacter and easier to read and to change. And there is even possible to introduce encapsulation in CAMOS(L), which is not realized in the internal representation.

Thus, I decided to use CAMOS(L) and write a compiler for it, because the internal representation of CAMOS is too complicated to program. The complicated syntax leads to programming errors, which are not easy to find.

5.2.1 Writing the compiler for CAMOS(L)

Having decided to write the compiler there were two alternatives: Using LISP or C as programming language and integrating the compiler in CAMOS or not.

Programming in LISP vs. programming in C

The use of the LISP language, provided by AMOS, would make the whole CAMOS system programmed in one language. LISP provides easy methods to program the lists needed for the compiler, too.

The C version of the compiler could use existing tools for developing compilers (flex and bison), so that it is easier both to write and to maintain the compiler.

These tools were the reason, why I wrote the compiler in C. Furthermore, I was used to program in C.

Integration in CAMOS vs. separation

The next choice was integrating the compiler into CAMOS to have a direct access to the database or making the compiler stand-alone, translating the file with the CAMOS(L) program into another file with the internal representation.

The internal version would have direct access to the AMOS parser in order to parse database queries in CAMOS(L). Furthermore, the result of the compilation could be read immediately into the database, showing all errors in the programming directly after the compilation.

The external version would provide a better possibility to test the compiler without using the database, because AMOS is a research prototype and still changing. Therefore, it is easier to separate errors in the compiler from errors in the database. It would also be possible to watch the result of the compilation in the output file independently from the database. The problem with the external version would be, that the database queries cannot be checked for syntactical errors, as the AMOS parser is not included in the external version.

In spite of this disadvantage I decided to write the compiler externally because of the debugging features. Besides, the AMOS database provides a simple possibility to include the external version into CAMOS, so that this can also be done afterwards.

5.2.2 Enhancing the current CAMOS system

The next decision was doing changes in CAMOS or not. Taking the system "as it is" would have the advantage that I would not have to deal with implementation details in LISP.

But there were too many restrictions in the existing implementation. The mechanisms to work with composite objects, multiple primitive suboperations and database operations existed only in parts.

This restrictions were the reason, why I decided to do the changes. Doing the changes would give me also a better understanding on how the different parts of CAMOS, as mentioned in chapter 4, work.

5.2.3 Modelling techniques in common

Now the different possibilities to model objects and operations in CAMOS(L) were examined. Using composite objects introduces aggregation in the modelling, but then problems with stability of composite objects arise. Operations can be synchronized using ordering of suboperations or wait conditions, which causes problems with concurrency of operations.

A detailed discussion about the different modelling techniques and their problems can be found in chapter 7.

5.2.4 Modelling the production cell

Modelling the objects of the production cell provides several alternatives: using simple, container or composite classes. For the operations it had to be decided, which synchronizations techniques could be used.

The results of the modelling and a discussion about the advantages and disadvantages can be found in chapter 8.

5.3 Summary

The task is now divided into the following parts:

1. Writing an external compiler for CAMOS(L) using flex and bison

2. Enhancing the implementation of CAMOS for composite objects, multiputs and database variables

3. Discussing the different alternatives of modelling classes and operations in CAMOS(L) in general
4. Modelling and programming the production cell in CAMOS(L)

Chapter 6

CAMOS(L): A language for manufacturing control

The first section of this chapter gives a description of the CAMOS(L) syntax, the second section provides an introduction into the semantics of the syntactical constructs.

6.1 The syntax of CAMOS(L)

The description of the syntax is divided into five main sections: syntax of defining classes, object and operations, the syntax of database queries and the syntax of the so-called AMOSQL mode. The changes to the proposed syntax in [Fal96] follow in the last part of this section. Appendix B contains the complete syntax in EBNF form.

6.1.1 General notes on the syntax

The compiler converts every character except within database queries and in the AMOSQL mode to **upper case**. So creating a class named "table" and another one named "Table" results in a compiler error. It also does not matter, if the keywords of CAMOS(L) are written upper case or lower case or even mixed.

Comments within a CAMOS(L) program start with "*" and end with the next "*\", the same mechanism as programming in C.

An **identifier**, used for variables, class name etc., starts with a character or an underscore followed by more characters, numbers or underscores.

6.1.2 Classes

The following example shows the syntax of a CAMOS(L) class. Keywords are written in upper case.

```
CREATE CLASS magnet SUBTYPE OF camosobject
  STATEVARS: state: integer = 0
  NODES: 1 "magnet on" ((state (.obj) = 1)),
         2 "magnet off" ((state (.obj) = 0))
  TRANSITIONS:  1 -> 2 "magnetSwitchOff" ((.obj)),
                2 -> 1 "magnetSwitchOn" ((.obj));

CREATE CLASS crane SUBTYPE OF lifter
  STATEVARS: h_position: integer = 1
  OBJECTVARS: magnet: magnet
  DBVARS: NumberOfLifts: integer = 0,
          maintenance: charstring = "FALSE"
  NODES: 1 "left position" ((h_position (.obj) = 1)),
         2 "right position" ((h_position (.obj) = 0))
  TRANSITIONS:  1 -> 2 "craneGoRight" ((.obj)),
                2 -> 1 "craneGoLeft" ((.obj));
```

The magnet class is defined to have the example complete, because the object variable magnet in the crane class refers to a class named magnet. The following sections and chapters will continue this example.

This example defines one class named crane, which is derived from a superclass named lifter. The name for the class has to be unique that means there must not be another class with the same name. It is also possible to have more than one superclass, then commas separate the identifiers for the superclasses.

Next, three different lists of variables can be defined, where again commas separate the elements of each list:

- State variables, preceded by the keyword `STATEVARS:`
- Object variables, preceded by the keyword `OBJECTVARS:`

- Database variables, preceded by the keyword `DBVARS`:

The definition of **state variables** and **database variables** starts with an identifier for the name of the variable. This identifier must be unique in all three lists of variables. The type of the variable follows after the colon. For **object variables** this must be a CAMOS(L) class defined before; for state variables and database variables there are two types implemented:

- `Integer` values range between 2147483647 and -2147483648. A "+" before positive integers is not allowed.
- `Charstring` values are enclosed by double quotation marks and have a maximum length of 200 characters.

The initialization of state variables or database variables is optional. Object variables are not allowed to be initialized.

Finally, the DFA for the class is defined through nodes and transitions.

- A **node definition** contains three parts:
  1. the unique number of the node
  2. a string in order to describe this node
  3. a boolean database query, describing the node constraint. Any identifier starting with a "." in this query refers to the instance of this class.
- A **transition definition** has also three parts:
  1. the number of the start node and of the end node for the transition, separated by "$->$"
  2. a string, representing the algorithm, which will be executed, when the transition triggers
  3. a database query, resulting in an object. The name of this object will be sent together with the algorithm to the control level to let the control level know, which object of the class should be changed.

The base class: camosobject

The only class provided by CAMOS, is the `camosobject` class. This is the base class in order to create a class hierarchy with CAMOS(L), that means all CAMOS(L) classes have this class as superclass. It provides some (read only) variables:

- currentstate: contains the number of the current node in the DFA of the object
- stable: true, if the object is in a stable state; false, if not (see section 3.2.4 for a description of stability)
- name: contains the name of the object, given in the object definition. Right now this is only for internal use.

6.1.3 Objects

CAMOS(L) also provides support for creating objects and initializing the variables:

```
CREATE OBJECT crane (h_position, magnet, currentstate)
                    crane42 (1, cranemagnet, 1);
```

This creates an object of the class crane with the name "crane42". Again this name must be unique for all objects. Setting the system-provided variable currentstate to a certain node number signals that the object should have an active DFA and that this node should be the initial state. Otherwise, the object does not have an active DFA.

The variables, given in the list enclosed by parenthesis after the class name, are initialized to the values, given in the list after the object name, so in the example:

- h_position = 1 and currentstate = 1, which means that the start node in the objects DFA is node one
- magnet = cranemagnet, an object, which must have been instantiated before

The initialization in the object instantiation overwrites the default values in the class definition. The initialization is optional, so without the example looks like this:

```
CREATE OBJECT crane crane42;
```

6.1.4 Operations

The following example should give an overview about the syntax of operations in CAMOS(L):

```
CREATE OPERATION MoveCrane (crane .c)
AS WHEN
  numberOfLifts (.c) < 50
DO
IF position (.c) = 0 THEN
  1: PUT .c: h_position = 1;
  2: magnetOff( ((cranemagnet (.c) )) );
  DEPENDENCIES: (1,2);
ENDIF
IF position (.c) = 1 THEN
  2: PUT .c: h_position = 0;
  3: magnetOn( ((cranemagnet (.c) )) );
  DEPENDENCIES: (2,3);
ENDIF
IF numberOfLifts (.c) = 40 THEN
  3: SET .c: maintenance = "TRUE";
ENDIF
```

Names of operations, here `MoveCrane`, must again be unique [1]. A list of formal parameter follows, enclosed in parenthesis. A **formal parameter** consists of the name of the parameter's class and of an identifier, to which the object is referred to within the operation. This identifier must begin with a "." to recognize it in database queries. Formal parameter can only be classes, defined in CAMOS(L); other database classes or base types such as integer are not allowed.

The **wait condition** is enclosed between the keywords `AS WHEN` and `DO`. It is a database query returning a boolean result. If the operation should start working immediately, the wait condition is simply: `AS WHEN true DO`.

Finally an operation contains a non-empty set of **if-statements**. Each if-statement has an if clause, called **if guard**, enclosed between `IF` and `THEN`. This is also a database query returning a boolean result. The body of an if-statement contains one or more suboperation, preceded by a label, which is a

[1] This makes overloading of CAMOS(L) operations not possible right now.

positive integer number followed by a colon. Instead of labels suboperations also a `DEPENDENCIES:` statement is possible.

There are three different suboperations: Primitive suboperations, operation calls and database suboperations.

Primitive suboperations are applied on a list of state variables of one object to change its state. They begin with the keyword `PUT`, followed by an object from the formal parameter list, on which this primitive operation should be applied on. After the colon follows a list of assignments, which variables of the given object should have which values. The variable must be a state variable, the value can either be an integer or a charstring (depending on the type of the state variable) or even a database query[2], enclosed in double parenthesis. The programmer has to ensure that the database query result is from the appropriate type. When there is more than one state variable to change in a single suboperation, the operation is called multiple primitive suboperation, or shorter **multi-put**.

Operation calls consist of the name of another operation followed by the formal parameter list for this operation. Each parameter of the list can either be a formal parameter of the calling operation or a database query, enclosed in double parenthesis, returning an object of the type, defined in the formal parameter list.

Database suboperations start with the keyword `SET`. Their syntax is the same syntax as having primitive operations, but applied on object variables or database variables. Multiple assignments are not allowed as well.

The `DEPENDENCIES:` **statement** introduces the possibility to order the execution of the suboperations. After the keyword follows a non-empty list of label pairs, separated by a comma and enclosed by parenthesis.

6.1.5 Database queries

As said in the description of the syntax of classes and operations, there are possibilities to use database queries within CAMOS(L). The syntax of those queries was adopted from AMOS and corresponds to a SELECT statement in AMOS, but without the leading SELECT keyword.

[2] Using database queries here is dangerous: the stability of formal parameter, used within these database queries, is not checked. So it is recommended, not to refer to state variables in this context.

The only thing the compiler does within these queries is replacing the formal parameter name by an internal name. There is no other checking of syntactical errors, so errors within queries appear only, when loading the compiled program into CAMOS, not during the compilation. The programmer must also take care that the return value of the query is from the appropriate type and is a single value, not a bag of values.

Accessing the classes, defined in CAMOS(L), is the same as having created them with the `create type` statement in AMOSQL. Access to the created objects is possible by adding "()" behind the name. For the crane example from above, "crane42()" will return the crane object and "h_position (crane42())" will return the current value of the horizontal position of the crane, which is stored in the VM.

An example for a query in the wait condition of an operation could look like this:

```
CREATE OPERATION maintainCrane (crane .c)
AS WHEN
  MAX ((SELECT numbersOfLifts (cra)
        FOR EACH crane cra)) > 45
DO
```

This operation starts working, if one of the cranes has done more than 45 lifts. As seen in this example, also the use of aggregation operators such as `max`, `count` or `some` is allowed. For a complete description of the syntax of queries in AMOS, see the AMOS manual [Kar94].

6.1.6 The AMOSQL mode

CAMOS(L) does not contain all the constructs for programming CAMOS, but the missing ones are so similar to the constructs in AMOSQL, the query language of the AMOS database, that I decided to give a direct connection to the database instead of including all these constructs in CAMOS(L). The AMOSQL mode is used for two main tasks: activating workers and overloading functions; but it can contain any AMOSQL statement.

The `:AMOSQL` keyword switches the compiler into the AMOSQL mode. Everything between this keyword and the `:CAMOS` keyword will be copied directly into the output file of the compiler, keeping the order of the execution within AMOS. For example, if a class is defined before the insertion of the AMOSQL mode and an operation was defined after, the class will be known within the AMOSQL mode, but the operation not.

Activating workers

Operations that are not activated by other workers, are called **top level worker** or simply **worker**. They wait for their wait condition, when the compiled CAMOS(L) program is loaded into CAMOS and the operation manager was started. Activating the example operation from section 6.1.5 as top level worker looks like this:

```
:AMOSQL
activateWorker (maintainCrane(),
                [getfunction(".crane42")],
                "cyclic");
:CAMOS
```

This example activates an operation called `maintainCrane` as a top level worker. The operation has one formal parameter, and for this example an instance of the crane class, crane42, is given to the worker. The "cyclic" statement at the end signals that this worker is an **iterative worker**. Non-iterative worker have an empty string there.

Overloading functions

As AMOS is an object relational database system, it is possible to overload functions. For example, when creating composite (see section 7.1.4) or container classes (see section 7.1.3), the `stable` function, provided by the base class `camosobject` can be overloaded, for example:

```
:AMOSQL
create function stable (crane c) -> BOOLEAN as
  select true where stable (magnet (o))
              AND (notany (lockedby (o)));
:CAMOS
```

Doing this still poses some problems. The overloading of AMOS does not work properly in all cases due to bugs in AMOS, but adding the overloaded functions at the end of the CAMOS(L) program does work. Furthermore, a deep knowledge on the internal structure of the CAMOS datatypes and functions and even on how to program in AMOSQL is needed.

The advantages and disadvantages for overloading the `stable` function are discussed in chapter 7 as well as an example for a composite resp. a container class, where this overloading is used,

6.1.7 Changes to the proposed CAMOS(L) syntax

There are differences between the proposed CAMOS(L) syntax in [Fal96], described in section 4.2.4, and the realized syntax. This section gives an overview about the changes and a motivation for the changes.

Changes in the class syntax

In the proposal for CAMOS(L) there is no own syntax in order to define classes. This was done in the AMOSQL mode using AMOSQL constructs. I introduced a new class syntax for two reasons: At first to have control about syntactical errors during compiling the CAMOS(L) program, not only when loading the compiled program into CAMOS. And second, and that is the main reason, to have an easy and short syntax in order to create the nodes and transitions of the DFAs. The original code for defining only one transition from the operation example in section 6.1.2 looks like this:

```
/* Creating the transition-rules */
create rule CRANETransFr1To2 (CRANE o) as
when currentState (o) = 1 and
h_position (o) = 0
do begin
set lockedBy(o) = current ();
algorithm (name(o), "craneGoRight");
set currentState (o) = 3
end;
activate CRANETransFr1To2 (:CRANE42);
```

In this way, the new class syntax provides an easier and faster way to create classes and DFAs. It also hides implementation details from the programmer, who does not need to deal with activating the locking mechanism (the `set lockedby` statement in the example) or with the syntax of how to call control algorithms in the real-time kernel.

Changes in the object syntax

The object syntax was introduced for the same reasons as the class syntax: to have the possibility to check more errors in the compiler and to make the programming shorter and easier.

Changes in the operation syntax

The proposed syntax of the example, given in section 4.2.4 has changed now to the following syntax:

```
CREATE OPERATION
  getPressedFetchNew (robot .r, e_r_table .t, press .p)
AS WHEN
  partsAt(.p) AND position(.p) = "open"
    AND palletCapacity (currentPallet()) >=
    COUNT ((SELECT b FOR EACH part b
      WHERE position(b) = currentPallet()))
DO
IF NOTANY(partsAt(upperConveyour())) THEN
  1: getPressedPart (.r,.p);
ENDIF
IF SOME((SELECT b FOR EACH part b
         WHERE name (position (b)) = "e_r_table")) THEN
  2: getNewPart (.r,.t);
  3: PUT .t: referenceElevation = 0;
  DEPENDENCIES: (2, 3);
ENDIF
IF SOME(partsAt (.t)) AND SOME (partsAt(.p)) THEN
  DEPENDENCIES: (1, 2);
ENDIF
```

There are the following differences:

- The formal parameter in the parameter list have to begin with a ".". The compiler has to differ between formal parameters in database queries, which are translated into an internal representation, and local variables such as the 'b' variable in the wait condition of the above example operation. These local variables are not translated but only copied.

- To have a possibility to recover from syntactical errors within the parser, the keywords `DO` and `ENDIF` terminate the wait condition resp. the if-statement.
- Parameters of operation calls can now contain database queries, enclosed in double parenthesis, as well.
- The labels contain no characters anymore.
- The syntax for primitive operations was completely renewed. The keyword `SET` changed to `PUT`, because the former fits better to database operations. In order to deal with multi-puts, the object that should be changed, was put before the list of variable-value pairs, because multi-puts are only allowed to be used on a single object. The value of a variable-value pair can now be a database query, enclosed in parenthesis. For a complete description of the syntax, see section 6.1.4.
- Database operations are new, their syntax is equivalent to the syntax of the new primitive operation.

6.2 The semantics of CAMOS(L)

To explain the meaning of the different parts of the language, the syntactical constructs of CAMOS(L) are explained in this section.

6.2.1 Classes

As said in section 3.3.2, classes summarize objects with similar properties in one structure. Generalization helps to build type hierarchies and so to reuse classes, designed before.

In CAMOS(L), generalization is provided through the list of superclasses, given in the class definition (see section 6.1.2). The defined class inherits all variables from the superclasses and from the superclasses of the superclasses up till the base class. Multiple inheritance is possible, but the programmer has to take care about cyclic inheritance.

The three variable types have three different meanings, which was the reason for separating them.

- State variables model with the nodes and the transitions the basic behaviour of the class. They are stored in the WM and the VM, so that the control level can detect differences and invoke changes in the real world.
- Object variables introduce object aggregation into the class definition (see section 3.3.3).
- Database variables store additional information about the class. They are not used in order to model the DFA, but can appear in database queries such as in wait conditions or if guards. They are only stored in the VM.

The basic behavior of a class

To design the basic behaviour for classes, DFAs are used in CAMOS(L). For an overview on DFAs, see section 3.2.2.

Back to the crane example from section 6.1.2 on page 50. The crane has one state variable h_position having only two values: 0 and 1, meaning the crane being in position 0 (right) or 1 (left).

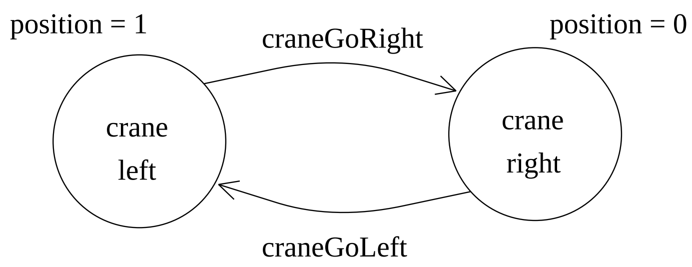

Figure 6.1: The DFA of the example class crane

This results in a DFA with two nodes (see figure 6.1), each having one value of the state variable as node constraint. Of course, this is a very simple example. The node constraints can be more complicated, especially referring to more than one state variable.

The syntax for the transitions describes, what the control level must do, when several conditions are satisfied: The start node defines the node, which node constraint has to be fulfilled, before the transition triggers. The end node defines the node, which node constraint must be fulfilled to make the transition trigger. The transition triggers if both are satisfied and sends the string with the control algorithm's name to the control level. The name of the object, given through the database query, is sent to the control level to separate different objects from the same class. Afterwards, the object is in the new state, the transition leads to.

More examples and different possibilities to model classes and DFAs are in section 7.1.

Relation to the WM concept

Relating to the VM and the WM concept, the semantic of transitions means that the node constraint of the start node refers to the value of the state variables in the WM (the now-state). The node constraint of the end node refers to the state of the same state variables, but in the VM (the current desired state). So summarizing this paragraph, a transition watches differences between the VM and the WM and triggers, if both assigned node constraints are fulfilled. (See also section 3.2.3)

To make the programming of CAMOS easier, the difference between VM and WM is hidden from the task level programmer. The state variables in the class definition contain the so-called reference variables (see section 3.2.3) that means the variables with the state, wished by the programmer. The actual variables are created automatically from the compiler and are only internally used in order to let the transitions detect differences between VM and WM (see above).

Hiding the actual variables does not affect the task level programming, because the wait condition of an operation checks automatically, if every object of the formal parameter list is stable (for an explanation of stability see section 3.2.3). So when entering an operation and evaluating the if guards, the reference and actual variables contain the same values.

6.2.2 Objects

Creating an object means creating one instance of the class. Initializing the variables of the object provides the possibility to set the object in a certain start state. When instantiating an object with a DFA (see page 52), the programmer must ensure that the initial state of the object (given through the currentstate variable) corresponds to the appropriate node constraint.

6.2.3 Operations

As the operation is the basic concept of the language ([Fal96], p.55), its general semantic has been described in the basics of the language in section 4.2.4. I decided to describe the whole semantics here again for two reasons: first to have all the semantics in one section and second to add refinements to the proposed semantics.

Operations introduce procedural abstraction. They contain a list of suboperations, which are possible concurrent or which must be executed in a certain order.

Formal parameter

The reason for introducing formal parameter was at first the reuse of code, that means calling the same operation with different parameters instead of writing several operations. Also, the formal parameters introduce a kind of locality of data: the scope for the objects, given to the formal parameter, is local to this operation. So there is no need to access to global database variables and this should not be done to avoid problems with instable objects. For a detailed description on that, see the following section on wait conditions.

The wait condition

The wait condition in an operation can synchronize the operation with the external environment. The start of the operation is delayed until the wait condition is true. Wait conditions include the check of the stability of all formal parameters, otherwise the value for the parameters are not corresponding to the values in the WM (see section 3.2.4).

The wait condition does not guarantee the stability of the objects from the formal parameter list *during* the execution of the operation. It is possible and for efficiency reasons wished that the execution of a database or primitive suboperation leads immediately to the execution of another operation.

The following example explains this:

```
CREATE OPERATION doFirst (object .o)
AS WHEN
...
   4: PUT .o: state = 1;
...

CREATE OPERATION doSecond (object .o)
AS WHEN
  state (.o) = 1
DO
IF true THEN
  1: PUT .o: state = 2;
ENDIF
```

When the `doFirst` operation finishes the primitive suboperation and the object is stable, the wait condition for the second operation is fulfilled and `doSecond` will execute, beginning also with a primitive suboperation on the same state variables as the first operation. So the object becomes instable again, although the first operation has not ended yet.

This example shows also, how the synchronization of two operations with wait conditions works.

The if clauses: guarding groups of suboperation

To provide alternative paths of executing an operation, if-statements have been introduced to group a set of suboperations together. When the wait condition of an operation is true, CAMOS checks *all* the if clauses of the suboperations. Those sets of suboperations, which have a fulfilled if clause are queued for execution. The if clauses do *not* contain an implicit order as in other programming languages. For example:

```
CREATE OPERATION showIfClause (crane c)
AS WHEN loaded (.c) = 0 DO
IF true THEN
  1: SET .c: loaded = 1;
```

```
ENDIF
IF loaded (.c) = 1 THEN
...
ENDIF
```

The second if statement will never be executed, because the wait condition of the operation ensures that the loaded variable of the crane is always zero. The if clause is not checked, when the `SET` finishes, it is checked together with all other if clauses, when the operation starts.

Dependencies: Making suboperations sequential

To introduce ordering of suboperations, dependencies between suboperations can be given. A list such as `DEPENDENCIES: (1,2), (2,3), (3,4);` means that the suboperation with the label one must have ended its execution, before suboperation two can start, two must be ready, before three can start and so on. In this case, the four suboperations are executed one after the other, that means sequentially. Another list like `DEPENDENCIES: (1,2), (1,3), (1,4)` means that the first operation must end, before the other three can start. As there is nothing said about dependencies between the last three operations, no assumptions can be made about that. They may execute in parallel or in an order, the CAMOS system decides to give them.

Primitive suboperations

Primitive suboperations provide the possibility to change the state of an object from the formal parameter list by changing the VM in the database though a database update statement (see section 4.3). This will lead to a call of a control algorithm in the real-time kernel for this changed object, if there is a transition between the old and the new state. Primitive suboperations are the most important parts of an operation, because they provide the *only* possibility to invoke a change in the real world through control algorithms.

As it is only allowed to change an object from the formal parameter list, a kind of encapsulation is introduced. Only own or inherited state variables from objects from the formal parameter list are allowed to be changed. So it is possible to recognize only by reading the formal parameter list of an operation, what objects at maximum are changed within this operation. The formal parameter list gives a kind of characterization of the operations content.

Multiple primitive suboperations

Multi-puts become useful, when a class has more than one state variable. An example illustrates the motivation for multi-puts:

```
CREATE CLASS platform SUBTYPE OF camosobject
  STATEVARS: x_position: integer = 0,
             y_position: integer = 0
  NODES: 1 "position 0,0" ((x_position (.obj) = 0
                      AND y_position (.obj) = 0)),
         2 "position 0,1"((x_position (.obj) = 0
                      AND y_position (.obj) = 1)),
         3 "position 1,0"((x_position (.obj) = 1
                      AND y_position (.obj) = 0)),
         4 "position 1,1"((x_position (.obj) = 1
                      AND y_position (.obj) = 0))
  TRANSITIONS:
...
```

This platform class has two motors, one for the x and one for the y direction. It has two positions in each direction (zero and one), that means four positions. The problem arises, when having a transition between the (0,0) and the (1,1) position. When there are no multi-puts, a decision has to be made, if the x_position or the y_position state variable should be changed first. But as the transitions are active rules from AMOS, the transition to the (0,1) position will trigger as soon as the y_position is changed and the primitive suboperation ends, even if the primitive suboperation changing the x_position follows immediately. But the aim was to go *directly* to the (1,1) position.

So multi-puts provide a possibility to change more than one state variable in the VM, before any transition can trigger and detect the difference between VM and WM.

Database suboperations

Database suboperations change database variables, which bind additional information to the object without causing changes in the real-time kernel. As wait conditions can contain database variables, database suboperations may lead to the activation of another operation. So they can synchronize the operations in the same way as primitive suboperations.

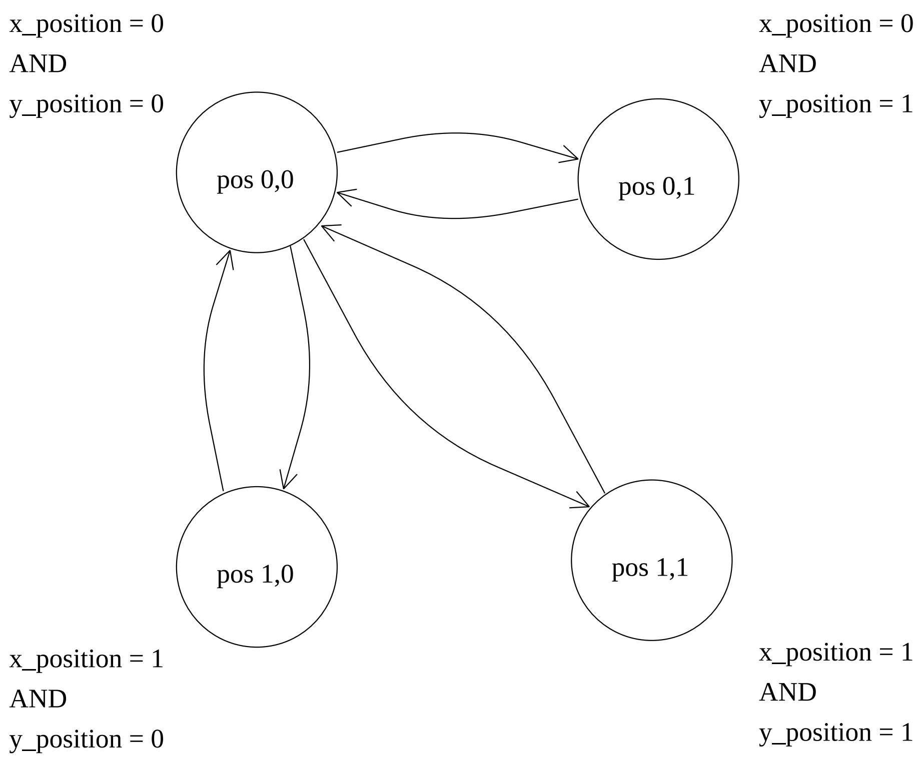

Figure 6.2: The DFA of the example class platform

Operation calls

Operation calls make it possible to create a hierarchy within the operations. Starting from the top level worker, which is activated separately (see section 6.1.6), operations are called, which again call other operations, until at the base of the hierarchy primitive suboperations or database suboperations are reached. So operation calls provide a procedural grouping of primitive and database suboperations. With `DEPENDENCIES:` statements it is even possible to make those operation calls sequential.

Operation calls are especially important, when a class is composed of other objects. To change the state of such a subobject, an access operation must be called (see section 7.3.1 for a detailed description and examples).

In the parameter list of an operation call, the database queries provide all possibilities of a relational complete database query language. It is possible not only to give objects from the formal parameter list or subobjects to the operation call, there could also be a database wide search for objects with a certain attribute:

```
CREATE OPERATION fetchMaterial (crane .c, material .m)
AS WHEN
  true
DO
IF true THEN
  1: moveCrane (.c, ((position (m)
            FOR EACH material m WHERE m = .m)));
ENDIF
```

This example operation calls an operation named `moveCrane`, which takes two arguments: the first is the crane, which should be moved, and the second is the position, where it should move to. This operation call moves the crane to the position, where the type of material is placed that was given in the formal parameter list.

6.3 Summary

CAMOS(L) is a compact task level language with a syntax similar to imperative languages such as PASCAL or C. It provides syntactical constructs to create classes and operations. Creating classes includes also the modelling of the basic behaviour of the class through DFAs. Operations can be scheduled with dependencies and if guards, wait conditions synchronize the operation with the external environment. Finally, using database queries introduces the possibilities of a relational complete query language into wait conditions, if guards and operation calls.

Chapter 7

Modelling techniques using object-oriented facilities

This chapter introduces a new abstraction level above the semantics of the syntactical constructs of CAMOS(L). It subdivides the modelling of classes and operations into different groups of classes and operations.

The basic work of DFAs and operations has already been examined in [Lob94] and is described in section 3.2.2 and section 4.2.4. Using the object-oriented facilities of CAMOS(L) and trying to model a more complex WM introduces new modelling techniques, which are described in the following sections and discussed afterwards.

7.1 Designing the classes

When modelling the real world, the task level programmer must extract the required information and abstract from unnecessary details (see section 3.2.1). Having the possibility for object-oriented modelling in CAMOS, the programmer must also think about grouping WOs with similar properties together to one class and creating a class hierarchy.

Now, classes can be divided into four different groups: basic classes, derived classes, container classes and composite classes, as described in the following sections.

7.1.1 Basic classes

Basic classes are atomic classes that cannot be subdivided into smaller parts. As they are the basic bricks in order to build the class hierarchy, they do not make use of generalization and do not contain object variables. They are derived from the base class of CAMOS(L), the `camosobject` class.

7.1.2 Derived classes

Derived classes use the generalization (see section 3.3.2) concepts. They inherit state variables from their superclasses and introduce some new ones to define, what is different to the superclass. They do not inherit the DFAs from the superclasses for two reasons:

- Creating a new class by deriving means that the properties of the superclasses should be reused, but also new properties should be added. These new properties usually influence the behaviour of the new class, so the old DFA has to be changed in anyway.
- Using multiple inheritance raises the question, how to combine the multiple DFAs.

Besides, derived classes can reuse the operations from the superclasses. For example, having a class named car, a derived class named fastCar and an operation driveCar, which takes a car as formal parameter, it is also possible to call driveCar with an instance of the fastCar class.

And finally, the derived class can overload operations from the superclasses to make them inaccessible for the new class (see section 3.3.2).[1]

7.1.3 Container classes

Container classes make use of the aggregation concept (see section 3.3.3). They have object variables, which refer to other, predefined classes by importing a link to an instantiated object into the instance of a container class. Opposite to composite classes, they reuse the whole functionality of the included objects, that means the subobjects run with their own DFAs.

The difference between having a container class with subobjects or all object independently is more a question of design than of functionality: The

[1] Overloading in not yet implemented in CAMOS(L).

objects are logically grouped into one class, but behave, as if they were independent. The subobjects do not know that they are linked into another object.

Container classes provide also a simplification of operation calls: when calling a suboperation or activating a worker it is not necessary to have all objects in the formal parameter list, the instance of the container class is sufficient. Suboperation calls can refer to the subobjects using the database queries in the formal parameter list.

This should be illustrated by an example. The class `crane`, defined in section 6.1.2, becomes a container class with the following object creation:

```
CREATE OBJECT magnet (currentstate) cranemagnet (1);
CREATE OBJECT crane (h_position, magnet, currentstate)
              crane42 (1, cranemagnet, 1);

CREATE OPERATION craneMoveNGrip (crane .c)
AS WHEN
  true
DO
...
IF h_position (.c) = 1 THEN
  1: magnetOn ( ((magnet (.c))) );
ENDIF
...
```

The first `CREATE OBJECT` instantiates an object of a magnet class, setting its currentstate to initialize the DFA (see section 6.1.3 for a description of currentstate). The second `CREATE OBJECT` instantiates the crane as well with its own DFA and with the cranemagnet as subobject.

Stability of container classes

Using container classes instead of independent objects provides also a change in the functionality through overloading the stable function of the container class (see section 3.2.4).

```
:AMOSQL
create function stable (camosobject o) -> BOOLEAN as
  select true where notany (lockedby (o));
```

```
create function stable (crane c) -> BOOLEAN as
  select true where stable (magnet (o))
              AND (notany (lockedby (o)));
:CAMOS
```

The first function is the original definition of the stable function in CAMOS. It defines the crane as stable, when there is no difference between the crane's state variables in the WM and the VM. The second function overloads the original stable function. To be stable the magnet of the crane must be stable as well.

There are reasons why it is necessary to change the meaning of stability for a container object. For the crane example this could be that it is not allowed to change the state of the magnet when the crane is still moving. Then the second function is the better alternative to design the stability of the crane class.

For the example of the `craneMoveNGrip` operation, the first stability definition means that the system enters the operation even if the magnet is instable. When calling the `magnetOn` suboperation, the wait condition of this operation will check the stability of the magnet, but suboperations of `craneMoveNGrip`, which have no dependency to the `magnetOn` suboperation, can execute in parallel. The second stability definition causes the `craneMoveNGrip` operation to delay all suboperations, until the magnet is stable as well.

7.1.4 Composite classes

Composite classes differ from container classes in their definition of behaviour: The composite class controls the behaviour of the subobjects, including the state variables of the subobjects into the node constraints of the composite class.

The following example is a composite version of the crane example:

```
CREATE CLASS crane SUBTYPE OF lifter
  STATEVARS: h_position: integer = 1
  OBJECTVARS: magnet: magnet
  DBVARS: NumberOfLifts: integer = 0,
          maintenance: charstring = "FALSE"
```

```
NODES: 1 "left position, magnet off"
         ((h_position (.obj) = 1
           AND state (magnet (.o)) = 0)),
       2 "right position, magnet off"
         ((h_position (.obj) = 0
           AND state (magnet (.o)) = 0))
       3 "right position, magnet on"
         ((h_position (.obj) = 0
           AND state (magnet (.o)) = 1)),
TRANSITIONS: 1 -> 2 "craneGoRight" ((.obj)),
             2 -> 1 "craneGoLeft" ((.obj)),
             2 -> 3 "magnetSwitchOn" ((magnet(.obj))),
             3 -> 2 "magnetSwitchOff" ((magnet(.obj)));
```

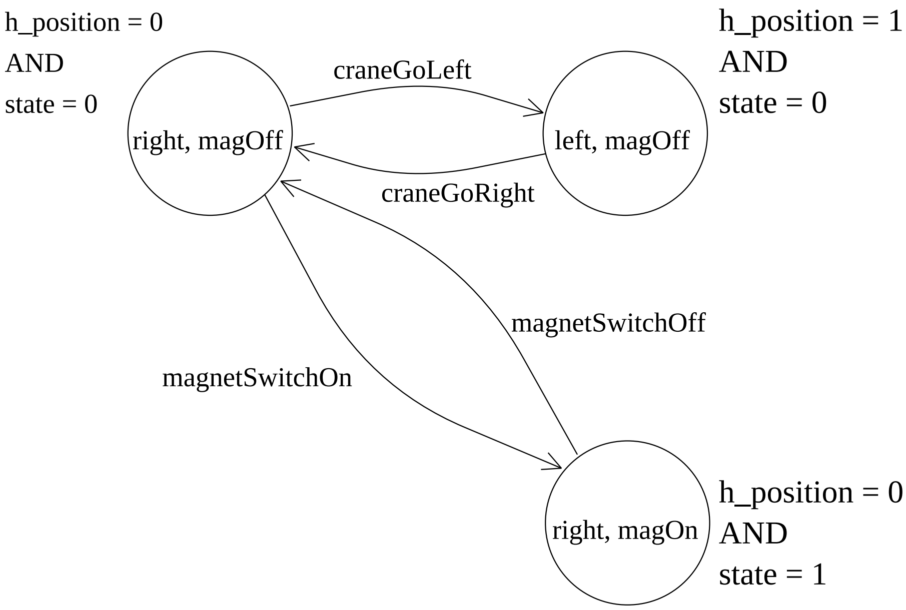

Figure 7.1: The DFA of the crane as composite object

The only changes in the class definition are one more node and two more transitions, resulting in a new DFA. Figure 7.1 illustrates the composite behaviour: The magnet is only allowed to be switched on, when the crane is in the right position, and the crane is only allowed to move from right to left position, when the magnet is switched off.

As the subobject's behaviour is included into the DFA of the composite class, the stable function for the composite class *must* be overloaded. Otherwise, changing the subobject would be allowed, even if the composite object itself changes its state. For the crane, this is the overloaded version of the stable function as described in section 7.1.3.

When creating instances of composite objects, the programmer must ensure that the subobjects of composite objects do not initialize the currentstate variable, otherwise there are two concurrent DFAs operating on the subobjects: the subobject's DFA itself and the DFA of the composite object.

For the crane example, the instantiation looks like this:

```
CREATE OBJECT magnet (state) cranemagnet (0)

CREATE OBJECT crane (h_position, magnet, currentstate)
              crane42 (1, cranemagnet, 3);
```

Again the programmer must ensure that the initial values for the state variables of the cranemagnet *and* the crane itself correspond to the initial state of the crane's DFA, given through the currentstate initialization.

7.1.5 Comparing multiple derived classes and composite classes

As classes using multiple inheritance and composite classes have a common base, this section compares both.

At first both classes reuse former defined state variables to design their own behaviour. Both do not use the DFAs, which where defined in the superclasses resp. subobjects. Inherited state variables are treated, as if they were defined directly in the class, whereas the state variables of subobjects cannot be accessed directly, for example in primitive operations (see primitive suboperation in section 6.2.3 and encapsulation in section 3.3.2). The only way of accessing state variables of subobjects is using operation calls to access operations (see section 7.3.1). Finally, it is not possible to inherit one class more than once. For example, realizing a crane with two magnets is only possible using composite classes.

7.1.6 Summary of the class groupings

The previous section provides only the groups of classes, I found during my work at this thesis. Of course, combinations of the groups such as classes, inheriting state variables and having subobjects, are possible.

7.2 Virtual sensors on the task level

Changing sensor values within the control level to give different information as measured by the sensors to the WM is called creating a virtual sensor (see section 3.2.1). It is also possible to create virtual sensors using information provided by the task level itself.

The following example clarifies the problem: A buffer is feed by a magnetic gripper with bolts. Every time, 500 bolts are in the buffer, it should empty itself through opening a drop side. The buffer has no sensors, but there must be one sensor measuring, if the buffer is full or not.

This knowledge can be gained from the ordering of operations: Every time, the gripper feeds the buffer, the buffer contains one more bolt. A database variable stores this information; it is set to zero, when the buffer empties itself and increased by one, when the gripper feeds the buffer. The only thing to do now, is guarding the value of the database variable and setting the state variable, representing the virtual sensor (here bufferFull), to "full", when the database variable equals 500. This change in the VM lets a transition trigger and an algorithm executes in the control level; in the example this will be the opening of the drop side.

```
CREATE OPERATION emptyBuffer (buffer .buf)
AS WHEN
  numbersOfBolts (.buf) = 500
DO
IF dropSideOpen (.buf) = 0 THEN
  1: PUT .buf: bufferFull = 1;
  2: PUT .buf: bufferFull = 0;
  DEPENDENCIES: (1,2);
ENDIF
```

The database variable alone cannot solve this problem. Database variables cannot be used in DFAs and so the activation of control level algorithms (here opening the drop side) is not possible.

Thus, virtual sensors on the task level provide status information in order to model the DFA of a class. When using them, the programmer has to ensure the correctness of the virtual sensing. For this example, it would not make sense to increase the database variable every time, the gripper switches its magnet off. It could also be possible that the gripper is not above the buffer.

7.3 Modelling the operations

After modelling the classes and their basic behaviour, the work of the machinery must be described, that means controlling of the production process. Depending on the purpose of the operation, there are three different types of operations.

7.3.1 Access operations

Access operations provide a guarded access only to change the state of one object. They are the bricks in order to create an operation hierarchy in the same way as basic classes are bricks in order to build a class hierarchy.

The `magnetOn` function from the composite crane (see section 7.1.4) is such an access operation:

```
CREATE OPERATION magnetOn (magnet .m)
AS WHEN
  true
DO
IF state(.m) != 1 THEN
  1: PUT .m: state = 1;
ENDIF
```

Access operations have several properties:

- no wait condition, because they give only guided access to a state change.
- only one formal parameter, because the access is only for one class
- no calls of suboperation

- no use as top level worker, only for exporting basic functionality to other, higher abstracted operations
- all primitive suboperations are guarded by if clauses, so that it is ensured that after the execution of an access operation the object is in the wished state.

Access operation can also have alternative paths of execution, for example:

```
CREATE OPERATION toggleMagnet (magnet .m)
AS WHEN
  true
DO
IF state(.m) != 1 THEN
  1: PUT .m: state = 1;
ENDIF
IF state(.m) != 0 THEN
  1: PUT .m: state = 0;
ENDIF
```

This is a combined variant of `magnetOn` and `magnetOff`.

7.3.2 Simple operations

Simple operations are the next higher abstracted operations above access operations. They have the following properties:

- The wait condition is not only `true` and synchronizes the operation to external events.
- The operation does not access to external information, as there is only the object itself in the formal parameter list.
- The operation can be used as top level worker, if the behaviour of the object, modelled by the operation, is independent from external information.

An example for a simple operation for the crane class is:

```
CREATE OPERATION moveCraneWhenUnloaded (crane .c)
AS WHEN
  state (magnet (.c)) = 0
DO
IF h_position(.c) != 0 THEN
  1: PUT .c: h_position = 0;
ENDIF
```

Activated as top level worker, this operation starts working every time, when the crane's magnet has been switched off by another operation. It moves the crane back to the right position. Called by another operations, the wait condition ensures that the crane is not moved, before the magnet is off. So even if the programmer forgot to introduce dependencies in the calling operation, to switch off the magnet first, and then moves the crane, both operations will work properly.

7.3.3 Complex operations

Complex operations are the highest abstraction level for operations. They make use of access and simple operations and are mostly top level worker. Their main task is to synchronize the work of different parts of the machinery using dependencies or wait conditions (see section 4.2.4). So they have more than one formal parameter in the parameter list.

Synchronization with dependencies

The following example shows an operation, in which the suboperations are synchronized with dependencies. It coordinates the crane with the platform example from section 6.2.3.

```
CREATE OPERATION craneGetFromPlatform (crane .c,
                                       platform .p)
AS WHEN
  loaded (.p) = 1 AND loaded (.c) = 0
DO
IF true THEN
  1: returnPlatform (.p);
  2: moveCraneToPlatform (.c, .p);
  3: lowerMagnet(.c);
```

```
  4: switchOn ( ((magnet (.c))) );
  5: liftMagnet (.c);
  6: SET .c: loaded = 1;
  7: SET .p: loaded = 0;
  8: movePlatformToPos1_1 (.p);
  9: moveCraneAwayfromPlatform (.c);
  DEPENDENCIES: (1,2), (2,3), (3,4), (4,5),
                (5,6), (5,7), (6,9), (7,8);
ENDIF
```

As the names of the suboperations indicate, the crane should move to the platform, get some material from it and move away again. The first suboperation `returnPlatform` moves the platform to the (0,0)-position, the second suboperation `moveCraneToPlatform` moves the crane to the (0,0) position of the platform. The (1,2) number pair in the `DEPENDENCIES:` statement orders the execution of the first two suboperations, the first must be finished before the second is started.

Another type of ordering expresses the (5,6), (5,7) dependency: The suboperations six and seven must only wait until the fifth suboperation is ready. Since there is no dependency between the sixth and the seventh suboperation, both may be executed in parallel after suboperation 5 has finished.

Thus, dependencies give the possibility to order suboperations within one operation, which must not execute in parallel. For the above example, it is necessary, that the magnet is on (suboperation 4), before the crane lifts the magnet (suboperation 5), so that it is ensured that the magnet holds the material.

Synchronization with wait conditions

The same task could also be programmed with several operations, synchronized by wait conditions:

```
CREATE OPERATION returnPlatformWhenLoaded (platform .p)
AS WHEN
   loaded (.p) = 1
   AND x_position (.p) != 0 AND y_position (.p) != 0
DO
IF true THEN
  1: returnPlatform(.p);
```

```
ENDIF

CREATE OPERATION moveCraneToLoad (crane .c, platform .p)
AS WHEN
  loaded (.c) = 0
DO
IF true THEN
  1: moveCraneToPlatform (.c, .p);
ENDIF

CREATE OPERATION loadCrane (crane .c, platform .p)
AS WHEN
  loaded (.c) = 0 AND loaded (.p) = 0
   AND h_position (.c) = 1 /* at the platform */
   AND x_position (.p) = 0 AND y_position (.p) = 0
DO
IF true THEN
  1: lowerMagnet(.c);
  2: switchOn ( ((magnet (.c))) );
  3: liftMagnet (.c);
  4: SET .c: loaded = 1;
  5: SET .p: loaded = 0;
  DEPENDENCIES: (1,2), (2,3), (3,4), (3,5);
ENDIF

CREATE OPERATION moveUnloadedPlatform (platform .p)
AS WHEN
  loaded (.p) = 0
   AND x_position (.p) != 1 AND y_position (.p) != 1
DO
IF true THEN
  1: movePlatformToPos1_1 (.p);
ENDIF

CREATE OPERATION returnCrane (crane .c)
AS WHEN
  loaded (.c) = 1
DO
```

```
IF true THEN
  1: moveCraneAwayFromPlatform (.c);
ENDIF
```

In this example, the different tasks of the platform and the crane are separate.

The `returnPlatformWhenLoaded` operation returns the platform always, when it is loaded and not at the unload position (0,0), the `moveUnloaded-Platform` does the opposite, when the platform is unloaded and not at position (1,1).

For the crane, the moving operations are also realized separately: The `moveCraneToLoad` operation move the crane always to the platform, when it is unloaded, and the `returnCrane` operation moves the crane away, when it is loaded.

The `loadCrane` operation models the interaction between the crane and the platform.

Instead of returning the platform at the beginning of the `craneGetFrom-Platform` operation from the previous example, the platform returns always, when it is loaded and not at the unload position (0,0). This is the same, when moving the platform to position (1,1) after the unloading: In the first example the platform goes to position (1,1), when the crane has unloaded the platform, in the second example the platform goes there, when it is unloaded and not already at that position. The movement of the crane is also realized in two separate operations: The moving task for the crane is in this example: Go to the platform, when you are unloaded, return from the platform when you are loaded.

7.3.4 Top level workers

There are two possibilities to execute simple or complex operations: Either the operation is called by another operation (suboperation call) or it is activated as iterative or non-iterative top level worker (see section 6.1.6 and section 4.2.4). Iterative worker restart with checking the wait condition, when one execution has finished. Non-iterative worker execute only once.

From the second example above, the move operations for the platform could be activated as iterative top level workers.

7.4 Discussion of the modelling techniques

In this section the advantages and disadvantages of the different modelling techniques are discussed to clarify the different purposes of them.

7.4.1 Classes

Creating a class hierarchy introduces the advantages of object-oriented design to the modelling of production processes:

- The real world can be modelled from an object-oriented view, so that the translation from the real world to the abstracted WM gets easier (see also section 3.2.1).

- Properties given through variables and operations of already created classes can be reused through generalization, so that enhancing the system gets easier and faster.

- Using aggregation of objects in container resp. composite classes provides a grouping of objects in one class.

Container classes vs. composite classes

Container classes reuse the subobjects without changing their behaviour. That means that designing the DFA of the container class must only take care of the states, provided by the new state variables. Nothing must be changed in the subobjects DFAs, but this moves the problem to another level: The behaviour of a container class will be different from the behaviour of the single subobjects, otherwise it makes no sense to group them together. Then operations have to model the combined behaviour of the container class and of the subobjects using wait conditions and if guards.

Using composite classes instead restricts the behaviour of the class within the DFA and provides a higher safety against programming errors in operations. The composite crane in section 7.1.4 is an example of higher safety through DFAs: It is only possible to switch the magnet on or off, when the crane is in the right position, and moving the crane is only allowed, when the magnet is off. Thus, it is ensured that an operation cannot do something violating these restrictions. However, designing the DFA of a composite class is more difficult than reusing the old DFAs, since the composite DFA includes the state variables of all subobjects in the node constraints. So the possible number of nodes raises exponentially with each new subobject.

7.4.2 Operations

Comparing the modelling of a production process with CAMOS(L) to the design of a production process without computer aid, the design of the classes could be building the machinery and giving them an interface to control them (e.g. switches). Then the design of operations is like creating tasks for each human worker in the production process. That means dividing the complete task of the production process into smaller subtasks, easier to handle for the human worker and also for the programmer of the production process.

Access operations

Using access operations has three main advantages:

1. Having all primitive suboperations guarded raises the safety of the system against programming errors. It is always ensured that the state of the system is the wished one; the system will not stop because of an unguarded suboperation.

2. Maintenance of the system gets easier, as a change of the primitive suboperation has to be done only at one place.

3. Supporting the idea of encapsulation introduces a locality of changes that means primitive suboperations are only executed within access operations.

The disadvantage of them is that they slow down the execution of operations of the task level language. They do not affect the real-time behaviour of the control system, as the real-time kernel is independent, but the execution of operations gets slower.

A real need for using access operations exists only for composite classes to have access to the subobject's state (see the example in section 7.1.4), using database queries. Everywhere else primitive suboperations can be used directly without calling access operations.

Thus, the use of access operations is recommended because of the advantages, given above, as long as the system's speed is not affected.

In terms of human workers, access operations describe the work for apprentices in the production process. Their task is very simple and they do not have any responsibilities for safety. Also, they do not have other workers, which they can direct.

Simple Operations

Simple operations describe the work, normal human workers would do in a production process. As they are limited to only one object, they do not require deeper knowledge of the production process. They are easy to do and therefore safe. They can make use of the access operation, so they can "direct" the apprentices.

Simple operations are easy to program as they affect only one object. Synchronization to external events is only possible in connection with complex operations, which change the internal state of the object from outside simple operations. Thus, simple operations are safe to use, but as they make use of the wait condition, concurrency problems can occur (see section 7.4.3).

Complex Operations

Complex operations model the intelligent behaviour of the production process. They are the "masters" of the production, advising and synchronizing the tasks of normal workers and apprentices.

Programming complex operations should contain that tasks are divided into smaller subtasks. It is easier to check the functionality of a small operation than of a large one, especially if the operation affects many objects. That corresponds to the fact that even masters cannot do tasks of any size and any complexity without making mistakes. Furthermore, tasks should be limited to one place to keep the purpose of the operation clear. This is the same for human worker: It is more efficient to have everything, needed for the task, at one place than to have everything wide spread.

The examples from section 7.3.3 show the difference.

In the second example, the tasks of the crane are separate: moving the crane to and from the platform is independent from how the crane is loaded between both move operations. The moving task for the crane is not as specific as in the first example. The operations could also be reused in order to coordinate other tasks between the platform and the crane.

For the platform, the tasks are separate as well: It does not matter, if a crane or another machine loads or unloads the platform, it will move to position (0,0), when it is loaded, and to position (1,1), when it is unloaded.

In this way, all moving operations become now simple operations; the only remaining complex operation is the `loadCrane` operation, containing the interaction between the crane and the platform. The suboperations correspond to the suboperations three till seven in the first example and are also serialized with dependencies, but this complex operation is now smaller and the move operations are separated from the loading operations.

Thus, using dependencies for synchronization makes the execution of an operation deterministic, whereas wait conditions provide more flexibility.

7.4.3 Concurrency between operations

There is one very important thing, a programmer has to take care of, when designing the wait condition of operations: It must be avoided that two operations try to change the same state variable of one object simultaneously. This is because CAMOS does not support a locking mechanism for write access. The wait condition of an operation guarantees *only* that the object is stable at the beginning of the operation, and the if guards are checked at the same time to choose the execution path.

So if a primitive suboperation ends, another operation may start because of the satisfied wait condition and is executed then in parallel. It is also possible that the wait conditions of two operations are fulfilled simultaneously, so both operations will be activated and queued for execution. Then all primitive operations will be executed, even if the same primitive operation has been executed from another operation before. This will lead to the system halting, because no transition in a DFA triggers, when the object is already in the wished state, but the system will wait for a reaction, as it has sent the order to change the real world.

An easy example for this contains the following two workers:

```
CREATE OPERATION move1 (crane .c)
AS WHEN
  loaded (.c) = 1
DO
IF true THEN
...
```

```
  1: PUT .c: h_position = 1;
...
ENDIF

CREATE OPERATION move2 (crane .c, platform .p)
AS WHEN
  loaded (.c) = 1 AND x_position (.p) = 1
DO
IF true THEN
...
  1: PUT .c: h_position = 1;
...
ENDIF
```

As soon as the second wait condition is fulfilled, both operations will be activated simultaneously. Regardlessly, which primitive suboperation will be executed first, the first will lead to a triggering transition, so that a control algorithm in the real-time kernel will be executed. After the execution, the control level sends an acknowledge message to update the world model (see also section 3.2.3 and section 4.3). The second execution of the primitive suboperation will not lead to a triggering transition, but as CAMOS does not know, if a transition triggers or not, it still waits for the acknowledge from the control level.

So when designing an operation, the programmer must be aware, which wait conditions of which operations can be fulfilled, when a primitive suboperations is executed and what state variables of what object those operations change. To say it again: It must not be allowed for two operations, to change one state variable of the same object at the same time.

Comparing this to the human controlled production cell, the problem is clear: It is not possible for two human workers to change one object simultaneously, but working on one object one after the other is possible.

7.4.4 Database queries

Database queries in classes or operations can find every object and test each condition within the WM, as AMOSQL is a relationally complete database query language.

They can also violate the encapsulation ideas (see primitive suboperation in section 6.2.3 and encapsulation in section 3.3.2). Using a query in a suboperation such as:

```
CREATE OPERATION databaseQueryProblem (platform .p)
...
  10: moveCrane ( ((c FOR EACH crane c
                    WHERE h_position (c) = 1)) );
...
```

results in moving a crane, which does not appear in the formal parameter list. This example introduces also the problem of bag results in CAMOS: What should the system do, if the query results in more than one crane instance?

The direct access to an object (see section 6.1.5) such as "crane42()" introduces the same violation of the encapsulation idea; so it is recommended using those constructs (if at all) only in complex operations, to keep simple operations simple.

Thus, he programmer has to take care that queries do never result in more than one object and check again that no concurrency problems occur.

7.5 Summary

Dividing the problem into objects and operations on those objects, grouping objects together to classes and then creating hierarchies of classes and operations increases the flexibility of programming a task level language such as CAMOS(L). Using encapsulation and most restricting DFAs provides safety, although the development of the DFAs becomes more difficult.

The following directives summarize how to program operations:

- Use guards intensively! As safety plays an important role in production processes, it is better to have too many than too few guards.
- Keep the operations local! That means, do not try to control tasks, which are spread all over the production process within one operation.
- Keep operations small! The bigger an operation is, the more programming errors can appear and the more difficult are changes within the operation.
- Watch out for concurrent operations!

Chapter 8

Implementation of the production cell

This chapter describes the implementation of the production cell when using CAMOS(L). Only the most interesting examples to explain the different modelling techniques (see chapter 7) are mentioned. The complete source code of the production cell is in appendix C.

8.1 Designing the classes

The first step when modelling the production cell is building a type hierarchy (see figure 8.1). As there are only six different objects in the production cell, the possibilities for using generalization are small.

8.1.1 Basic classes: Building reusable types

The `actuator` class was introduced to emphasize that all classes, derived from it, are active objects. If the world model should also contain passive objects such as the blanks or other material, a "passivator" class would be the base class for them. As the `actuator` class has no variables, this is the only purpose.

In the belt class those properties of the two conveyor belts are modelled, which both have in common:

```
CREATE CLASS belt SUBTYPE OF actuator
    STATEVARS: loadedend: integer = 0,
```

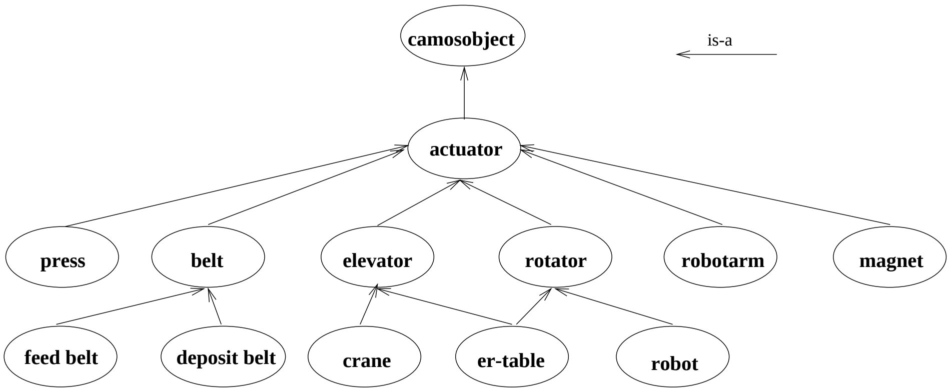

Figure 8.1: The class hierarchy for the production cell

```
                loadedbeg: integer = 0;

CREATE OPERATION moveToEnd (belt .b)
AS WHEN
  loadedbeg (.b) = 1 and loadedend (.b) = 0
DO
IF true THEN
    1: PUT .b: loadedbeg = 0,
               loadedend = 1;
ENDIF
```

At first both belts provide a photocell at the end, which indicates if there is a blank in the photocell or not. This sensor was directly mapped to the world model, whereas zero means no blank in the photocell and one means that there is a blank.

Secondly both belts need to have a virtual sensor, measuring if there is a blank at the beginning of the belt. This should make it easier to avoid double loading of the belts. Furthermore, one simple operation is provided, which forms a common behaviour of a belt: if it is loaded at the beginning and there is nothing in the photocell, it transports the loaded blank to the end of the belt.

8.1.2 A first example class: the crane

```
CREATE CLASS crane SUBTYPE OF elevator
    STATEVARS:      h_position: integer = 0
    OBJECTVARS:     craneMagnet: magnet
    DBVARS:         craneloaded: integer = 0
    NODES:
      1 "Init"
      ((h_position (.obj) = 0)),
      2 "At depBelt and down"
      ((h_position (.obj) = 2 AND elevation (.obj) = 0)),
      3 "At depBelt and up"
      ((h_position (.obj) = 2 AND elevation (.obj) = 1)),
      4 "At feedbelt and up"
      ((h_position (.obj) = 1 AND elevation (.obj) = 1))
    TRANSITIONS:
            1 -> 3 "craneToDepbelt" ((.obj)),
            2 -> 3 "craneUp" ((.obj)),
            3 -> 4 "craneToFeedbelt" ((.obj)),
            4 -> 3 "craneToDepbelt" ((.obj)),
            3 -> 2 "craneDown" ((.obj));
```

The crane inherits the elevation state variable from the elevator class to model that the magnet can be lifted up (elevation $= 1$) and lowered down (elevation $= 0$). This modelling in two states abstracts from the fact that the sensor measures the elevation of the magnet in real values, but these two states are the only interesting states.

The crane has also a state variable for the horizontal position, which is a direct mapping from the control level to the world model: One means the crane is above the feed belt and two means the crane is above the deposit belt; zero had to be introduced, since the initial position of the crane is between both belts. The fourth possible state, when having two sensors (two sensors with two states each: 2^2 combined states), means that the crane is at both belts at the same time. This is not possible.

Since the crane contains also an object variable (a magnet), from which the DFA is reused, it is even a container class.

The DFA of the crane is shown in figure 8.2. It fulfills one safety requirement given in the task description: To avoid a collision between the magnet

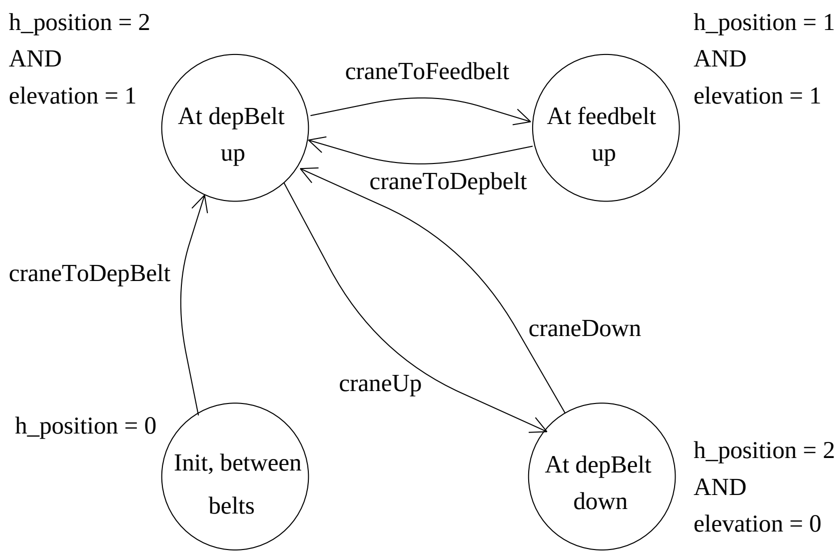

Figure 8.2: The DFA of the crane in the production cell

and the feed belt the crane must go to the deposit belt before lowering the magnet and must also lift up the magnet before going to the feed belt.

8.1.3 Derived classes: the belts

At first I thought to model both belts as one class, but then I found a small but significant difference: the feed belt unloads itself actively through moving a blank over the end of itself, when the er-table is there to get it. The deposit belt is passively unloaded by the crane. It must never move a blank over the end of itself. So I modelled the above mentioned basic class for both belts and then derived two new classes from it.

Both belts include a maximum of safety within their DFAs to show examples, how complex even objects with a simple task can be.

The feed belt

```
CREATE CLASS feedbelt SUBTYPE OF belt
    STATEVARS: pileEmpty: integer = 0 /* virtual sensor */
    DBVARS: blanksfrompile: integer = 0
...
```

The feed belt has another problem, the deposit belt does not have: There are two possibilities of loading it with two different control algorithms to call. To start the simulation the feed belt is loaded from a pile with five blanks. When all the blanks are within the production cell, the crane loads the feed belt as soon as there are blanks at the deposit belt. So another virtual sensor measures, if the pile is empty (`pileEmpty` $= 1$) or not (`pileEmpty` $= 0$). The database variable `blanksfrompile` counts, how many blanks have been put on the feed belt. This mechanism is the same as in the buffer example in section 7.2.

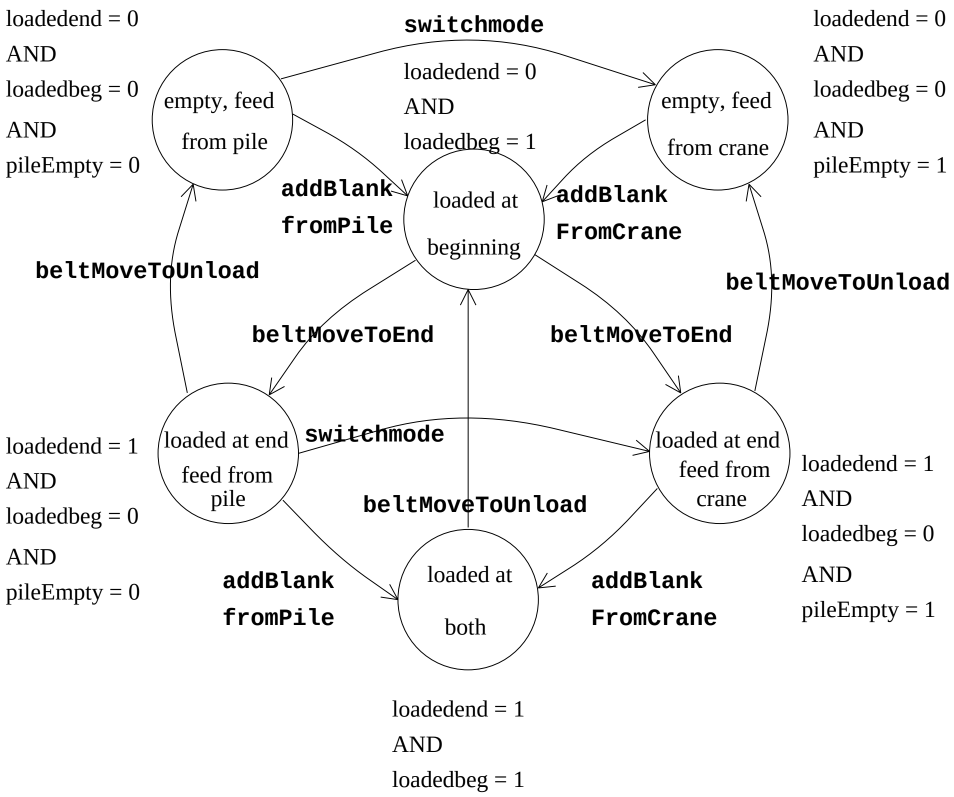

Figure 8.3: The DFA of the feed belt in the production cell

To solve the problem with the different control algorithms two nodes in the DFA (see figure 8.3) are doubled: The nodes, when the feed belt has no blank at the beginning ("empty", "loaded at end"). If the pile is not empty, the control algorithm **addBlankFromPile** is called to load the feed belt. Otherwise, the control algorithm **addBlankFromCrane** is called. The control algorithm **switchmode** is necessary, to acknowledge a change of the virtual sensor to the world model. As the pile is never filled again, there is no transition back; when the pile is empty, only the right part of the DFA is active.

This paragraph describes the behaviour of the feed belt. Loading the feed belt is done either from the pile, if it is not empty, or from the crane. Being loaded at the beginning, the feed belt transports the blank to the end, that means until the beam of the photocell is interrupted. Then there are two possibilities: Either another blank is put at the beginning of the feed belt, then the belt is loaded at both possible positions, or the belt will move again, until the blank leaves the end of the belt.

It needs further explanation, that the belt goes into the "loaded at beginning" state after unloading from the "loaded double" state: The task description says that the belts should not be loaded again until the last blank has arrived at the photocell. So if the feed belt is loaded at both positions and moves the blank at the end out of the photocell, the blank from the beginning position has not yet arrived at the photocell. That means, the feed belt is not yet in the "loaded at end" state and it is not allowed to load the feed belt again. That is exactly the description of the "loaded at beginning" state and so the feed belt will change to this state after unloading one blank from the "loaded double" state.

The deposit belt

```
CREATE CLASS depbelt SUBTYPE OF belt
    STATEVARS: atunload: integer = 0  /* virtual sensor */
...
```

Opposite to the feed belt, the deposit belt has three possible positions for blanks: at the beginning of the belt, at the photocell and behind the photocell, where the blanks are unloaded by the crane. Another virtual sensor provides this information about the "atunload" position, that means behind the photocell.

Figure 8.4 shows the DFA of the deposit belt. The basic sequence is: loading a blank at the beginning of the belt, moving it to the end (that means to the photocell) Then the belt moves the blank to the unload position, where the crane catches the blank, so that the belt is empty again. Alternative paths in the DFA cope with having two blanks on the belt at the same time.

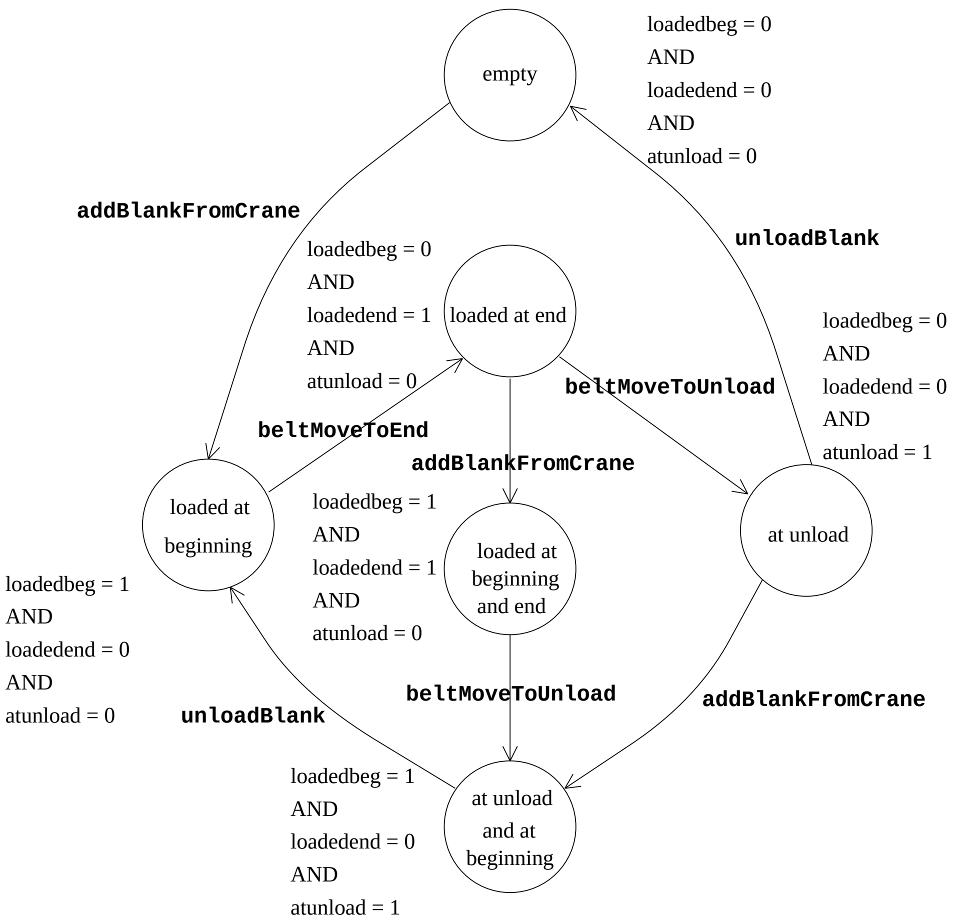

Figure 8.4: The DFA of the deposit belt in the production cell

It must always be ensured that no blank will be transported further than to the unload position. That is why the only transitions leading away from the states, where the deposit belt is loaded at the end, are transitions that send an algorithm to the control level to unload the blank at the end or to load another blank at the beginning, but no moving algorithms.

8.1.4 Composite or container classes: the robot

The first thought about the robot was making it as secure as the belt and so modelling it as composite class with one DFA for all state variables, but the possible state space is too large. The rotation of the robot has six different states, the extension for each arm three and the magnet at each arm two states. That results in $6*[(3*2)*2] = 216$ possible nodes for an all-including DFA. So I decided to model it as a container class, trying to provide safety in the operations.

```
CREATE CLASS magnet SUBTYPE OF actuator
    STATEVARS: magnetstate: integer = 0
...
CREATE CLASS robot_arm SUBTYPE OF actuator
    STATEVARS: extension: integer = 0
    OBJECTVARS: armmagnet: magnet
    DBVARS: armloaded: integer = 0
...
CREATE CLASS robot SUBTYPE OF rotator
    OBJECTVARS: arm1: robot_arm,
                arm2: robot_arm
...
```

The magnet class has one state variable, which is also a virtual sensor. It simply assumes that the magnet is on when the command for it has been sent.

Modelling the robot arms was more difficult. The control level provides a sensor for each robot arm, which measures the extension of the arm as a real value. But as the robot arms have different tasks, they also have different extensions to go to. Looking deeper into the task of the robot arms, I found, that both arms have only three positions to go to: one position, where they are most retracted to avoid collision with other machinery (the "safe position"), one position, where they are most extended (the "front position") and one position in between (the "back position"). This task dependent fact introduces the possibility to create a single robot arm class with a DFA having only three nodes. The control level abstracts from the measured sensor values and translates them to the three states for each robot arm (see section 8.2.3).

Finally, I modelled the robot base, which inherits the rotation state variable from the rotator class. It has six different positions, which are reached in a sequential order: in the initial position the robot base is not rotated, its rotation is zero degree. The description of the task of the robot (see section 2.1.1) gives the other possible positions for the robot base and the order, in which the positions are reached:

1. the robot rotates to the er-table to get a blank
2. the robot rotates counterclockwise, until arm2 reaches the press to get the last forged blank
3. the robot rotates again counterclockwise, until arm2 reaches the deposit belt to put the forged blank on the belt
4. the robot rotates further on, until arm1 reaches the press to load it with a blank
5. the robot rotates clockwise, to avoid a collision of arm1 and the press

Together with the initial state are these the mentioned six possible states. Figure 8.5 shows the DFA of the robot base. As the whole behaviour of the robot base is sequential with no alternative paths, it is a chain of nodes.

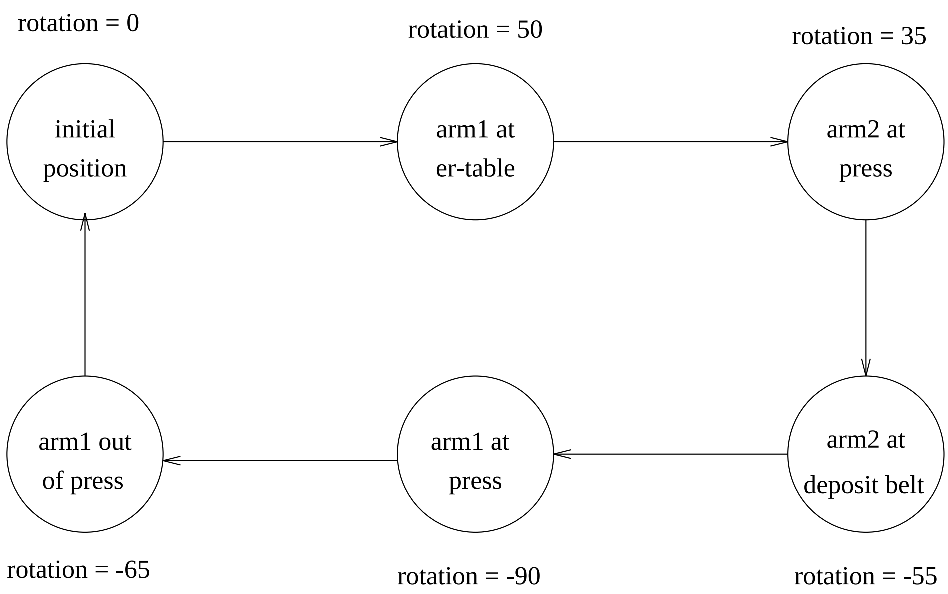

Figure 8.5: The DFA of the robot base in the production cell

At last, the stability problem with container classes as mentioned in section 7.1.3 had to be solved. I decided not to overload the stable function to make it possible to move the arms and the robot base simultaneously. This is not the safest solution, but faster in any way. Operations and their wait conditions implement the required safety instead.

8.2 Designing the control code for the simulation

The modelling of the classes defines the state variables and creates in this way the interface to the task level programmer, the WM. The DFAs model the behaviour of the world objects, the transitions in a DFA invoke a change in the real world by sending the name of a control algorithm to the real-time kernel.

For this production cell, the control algorithms are included in the simulation, so this programming is very special for this production cell; for other tasks there must be another implementation of the real-time kernel. That is why there is nothing said about programming the control level in the previous chapter about modelling techniques.

The first section describes how programming the real-time kernel in the simulation works in common. Then the modelling of the different control algorithms is separated according to the underlying sensors: control algorithms for switch-sensors, for real-value sensors and for virtual sensors.

For a complete description of the simulation see Appendix A in [Lew94]. The control code for this production cell is available at URL:
`http://www.ida.liu.se/labs/rtslab/projects/camos/simctrl.txt`.

8.2.1 Programming guards for the simulation

The CAMOS system does not provide any guarantees for real-time behaviour, therefore the built-in mechanism of the simulation models the real-time behaviour.

All control algorithm in the real-time kernel have the same structure: They consist of two parts: One part, which is activated by the triggering transition and a second, which acknowledges the state transition to the world model. Both are implemented in TCL procedures. The first procedure uses the built-in mechanism, called guard, to activate the second, when the transition is done.

An example illustrates this:

```
proc craneToDepbelt { obj } {
global crane
    set crane $obj
    new_guard 10 == 1 craneDepack
    cmd crane_to_belt2
}

proc craneDepack {} {
global crane
    cmd crane_stop_h
    tack [format "(TACK %s (ACTUALH_POSITION 2))" $crane]
}
```

These are the two parts of the **craneToDepbelt** command from figure 8.2, the travelling crane. The first procedure is directly called from the triggering transition. Its parameter contains the name of the crane to make the procedures reusable for different objects of the same class. The `new_guard` command is the built-in mechanism in order to model the real-time behaviour. Its general syntax is:

```
new_guard sensor_number operator destination_value
          acknowledge_command
```

In this case the `new_guard` command watches sensor number 10 (the sensor, which measures, if the crane is above the deposit belt) and starts the acknowledge procedure `craneDepack` when the sensor is equal than 1. After the activation of the guard, the crane starts to move to the deposit belt.

When the guard activates the second procedure, it stops the crane immediatly and sends an acknowledge message to the CAMOS interpreter that the state transition of the crane is done. This message includes the new value of the changed state variable, here the actual variable.

8.2.2 Control code for switch-sensors

Switch-sensors are the simplest possible sensors. They measure only two values: on or off. The example of the above section already provides a control algorithm for a switch-sensor. To show the use of the objectname as parameter, which is sent additionally to the control algorithm's name (see section 6.1.2), here another example from the belt class is presented:

```
proc beltMoveToEnd {obj} {
    global belt
    set belt $obj
    if { $obj == "FEEDBELT" } {
        new_guard 13 == 1 feedbeltendack
        cmd belt1_start
    }
    if { $obj == "DEPBELT" } {
        new_guard 14 == 1 depbeltendack
        cmd belt2_start
    }
}

# two procs, because ifs take more time to react,
# the stop must be immediately
proc feedbeltendack {} {
    global belt
    cmd belt1_stop
    tack [format "(TACK %s (ACTUALLOADEDBEG 0)
          (ACTUALLOADEDEND 1))" $belt]
}

proc depbeltendack {} {
    global belt
    cmd belt2_stop
    tack [format "(TACK %s (ACTUALLOADEDBEG 0)
          (ACTUALLOADEDEND 1))" $belt]
}
```

The `beltMoveToEnd` control algorithm is activated by the DFAs of both belts, when a blank should be transported from the beginning position to the photocell. In opposite to the example from the crane it has two subparts to deal with both belts. The acknowledge procedures are separated not to stop the belts too late. Having both included in one procedure in the same manner as the first procedure, using an if to differ between both belts, might take too much time, so that the belt stops too late.

As moving a blank from the beginning of a belt to the end affects two state variables, the control level has to acknowledge also the new values of both state variables. This is, how the control level reacts to multiple primitive operations (see section 6.2.3).

8.2.3 Control code for real-value sensors

Mapping real-value sensors directly to the world model does not make sense because of the indefinite state space of real values. Thus, all real value sensors, provided by the production cell, are mapped to discrete values to reduce the state space to the minimal information, needed for operating the production cell.

Again the crane from the production cell gives an example for a mapped real-value sensor:

```
proc craneDown { obj } {
global crane
    set crane $obj
    new_guard 12 >= 0.9450 craneDownack
    cmd crane_lower
}
proc craneDownack {} {
global crane
    cmd crane_stop_v
    tack [format "(TACK %s (ACTUALELEVATION 0))" $crane]
}
proc craneUp { obj } {
global crane
    set crane $obj
    new_guard 12 <= 0.6593 craneUpack
    cmd crane_lift
}
proc craneUpack {} {
global crane
    cmd crane_stop_v
    tack [format "(TACK %s (ACTUALELEVATION 1))" $crane]
}
```

These two control algorithms map the indefinite amount of possible elevation states to the two only interesting state: Being at the height of the feed belt (0.6593) or at the height of the deposit belt (0.9450).

8.2.4 Control code for virtual sensors on the task level

Having virtual sensors on the task level requires a special form of control algorithm in the real-time kernel, as there is no real existent equipment to measure the value. The only possibility for the real-time kernel to react is to assume that the value of the sensor is reached in zero time. Thus, the only thing to do is to send an acknowledge about the new value. For the virtual sensor of the feed belt, measuring, if the pile is empty, it looks like this:

```
proc switchMode { obj } {
    tack [format "(TACK %s (ACTUALPILEEMPTY 1))" $obj ]
}
```

8.3 Designing the operations

The design of the operations was done in the same order as described in chapter 7: first the access operations, then the simple operations and finally the complex operations.

8.3.1 Access operations

Writing the access operations does not take much time, as they have an easy structure (see section 7.3.1), but one third of all operations are access operations, so they enlarge the code. The following example for an access operation shows their structure:

```
CREATE OPERATION saveArm (robot_arm .ra)
AS WHEN
  true
DO
IF extension (.ra) != 0 THEN
    1: PUT .ra: extension = 0;
ENDIF
```

This example was taken from the robot arm class. It puts the robot arm, given in the formal parameter list, into its safe position, so that the robot base can rotate without causing a collision.

8.3.2 Simple operations

Simple operations are the first operations modelling active behaviour in the production cell. The `moveToEnd` operation from the basic class in section 8.1.1 is one example for a simple operation, the following describes the behaviour of the er-table:

```
CREATE OPERATION returnTable (er_table .e)
AS WHEN
  ertloaded (.e) = 0 and rotation (.e) != 0
    AND elevation (.e) != 0
DO
IF rotation (.e) != 0 AND elevation (.e) != 0 THEN
    1: PUT .e: rotation = 0,
           elevation = 0;
ENDIF
```

This simple operation is activated as a top level worker and starts working every time, when the er-table is empty and not in the position to be loaded by the feed belt. That means, every time the robot unloads the er-table, it returns automatically to the feed belt to be loaded again.

8.3.3 Complex operations

The most important operations for the behaviour of the production process are the complex operations. They synchronize the different machinery in the production cell. Nearly all of them are interesting to see, because they describe the whole task of the production cell in very little pieces, being local restricted to the place, where the synchronized machinery meets. As the space in this report is limited, only the robot example from above will be continued to show, how the smaller safety of a container classes compared to a composite class can be compensated by programming operations.

At first the iterative and only top level worker for operating the robot:

```
CREATE OPERATION operateRobot (robot .r, er_table .e,
                               press .p, depbelt .db)
AS WHEN
  ertloaded (.e) = 1 AND rotation (.e) = 50
   AND elevation (.e) = 1 AND rotation (.r) = 50
DO
IF true THEN
    1: loadArm1 (.r, .e);
    2: rotateArm2ToPress (.r);
    DEPENDENCIES: (1,2);
ENDIF
IF loaded (.p) = 1 THEN
    3: loadArm2 (.r, .p);
    DEPENDENCIES: (2,3), (3,4);
ENDIF
IF loaded (.p) = 0 THEN
    DEPENDENCIES: (2,4);
ENDIF
IF true THEN
    4: rotateToBelt (.r);
    5: unloadArm2 (.r, .db);
    6: saveArm ( ((arm2 (.r))) );
    7: rotateArm1ToPress (.r);
    8: unloadArm1 (.r, .p);
    9: rotateArm1OutPress (.r);
    10: retractArm ( ((arm1 (.r))) );
    11: rotateToErt (.r);
    DEPENDENCIES: (4,5), (5,6), (5,7), (7,8),
                  (8,9), (8,10), (10,11);
ENDIF
```

According to the wait condition, the robot should start working, when there is a blank on the er-table and both the robot and the er-table are at the right positions. The following mostly serialized sequence of suboperations describes the work of the robot. This serialization provides the safety that could have been also reached through modelling the robot as a composite class. Describing the work of the robot in words makes this easier to understand:

At first robot arm1 retracts into the "back position" and takes the blank from the er-table, done by the operation `loadArm1`, which has to be executed before all the other suboperations. After successfully ending `loadArm1` arm1 is surely loaded with a blank. So the robot can rotate counterclockwise, until arm2 points to the press (done by `rotateArm2ToPress`). Again this operation must be ready with executing, before any other operation may execute. Then the robot may load arm2 with a blank from the press or not, depending on if there is a blank in the press. This is only used for the first blank, processed in the production cell, when there has not been any blank forged in the press yet.

The third `IF` is only necessary to continue the serialization, if the second `IF` is not executed. This guarantees that suboperation four is not executed, before one, two and possibly three are ready. Then the robot rotates to the deposit belt and afterwards unloads a possible taken blank on the deposit belt. If there was no blank taken, the `unloadArm2` operation simply returns immediately without changing anything in the simulation (see appendix C for the complete source).

Now follow two parallel suboperations: Retracting arm2 into the safe position and rotating arm1 to the press. This can be done in parallel, because arm2 cannot collide with any machinery, when the robot rotates from -55 degrees to -90 degrees. Then arm1 is unloaded in the press. Again this suboperation executes separately to avoid, that the robot rotates, before the blank is safely positioned in the press. Then follow again two parallel suboperations, retracting arm1 to the position to take again a blank from the er-table, and rotating the robot out of the press. This contains no risk of machinery collision. Finally the robot rotates back to the er-table and the operation restarts again when there is another blank on the er-table and the table is in its unload position.

Thus, the serialization ensures, that no parallel suboperations can lead to an error. Concurrency problems because of other workers, operating the robot, cannot appear, since the `operateRobot` operation in the only top level worker.

The stable function of the robot is not overloaded, because this would make it impossible to retract the robot arms and rotate the base of the robot simultaneously.

8.4 Comparison to the requirements of the case study

In this section the requirements on the system, given through the task (see section 2.2), are summarized. There are no formal proofs, as this was not part of this thesis.

8.4.1 Safety

As described in section 2.2 the safety requirements are divided in four different classes of problems. Problems with machines, moving further than their specification, cannot appear for two reasons:

1. The real-time kernel within the simulation guarantees that the movement stops at the required place (see section 8.2.1).

2. Activating control algorithms twice is prevented by using active rules for the transitions of DFAs. When a control algorithm has executed and the DFA has changed its state, the same change request again will not result in the same transition triggering (see section 4.3 for how transitions work).

The following paragraphs discuss the other three mentioned safety classes.

Collisions between machines

As the behaviour between separate classes cannot be modelled in DFAs, the operations have to take care that collision between machines cannot appear. The following list gives an overview on the possible collisions and the reasons, why they cannot appear. It was taken from Appendix A, section 6 of [Lew94].

1. collision between er-table and feed belt: Avoided, because the real-time kernel guarantees that the er-table stops rotating. This belongs to the problem of the er-table moving out of its specified range.

2. collision between er-table and the loaded robot arm1, when the er-table is up and robot arm1 above the er-table: Avoided through the `turnTable` operation: the er-table lifts itself only up, when arm1 is unloaded.

3. collisions between both robot arms and the press: Avoided through the `pressBlank` and `pressReturn` operation: the press in only moved, if the robot is in a safe rotation position.

4. collision between the crane and the belts: Avoided through the DFA of the crane. See section 8.1.2 for details.

Dropping blanks outside certain areas

Only four of the machines in the production cell can drop blanks: both belts, the robot and the crane.

The feed belt guards dropping blanks through the `moveToErt` operation, which waits until the er-table is ready to take the blank. The robot releases blanks, taken from arm1 only, when in the press, and blanks, taken from arm2 only, when above the deposit belt. The DFA of the deposit belt prevents that the belt starts, when there is a blank at the unload position. Finally, the crane switches off the magnet only, when in the unload position above the feed belt.

Keeping blanks sufficiently separate

On both belts the DFA allows to load another blank on the belt only, if the last blank has arrived at the photocell. At all the other places, this safety problem is reduced to loading a machine with more than one blank, since all machines except form the belts can only carry one blank. Therefore, the operations that load the next machine with a blank, guard always if there is already a blank in that machine.

8.4.2 Liveness

No proved statements can be made about this, but watching the simulation gave the impression that the implementation provides liveness.

8.4.3 Efficiency and flexibility

For this production cell, there are only a few statements on flexibility or efficiency to make. As this is the only implementation of the production cell in CAMOS, there is no possibility to compare it to another one. So the speed of the implementation cannot be judged. The throughput of the production cell seems to be ok, as all the blanks, provided by the simulation, are in use during operating the production cell. For efficiency, one thing might be interesting: When the classes and their DFA were completely modelled, I programmed at first a completely serialized operation to test the functionality of the classes. After I removed some errors in the classes, I reused them completely for the complex implementation, described in this chapter. So separating the internal class behaviour and the behaviour between objects makes it also easier to change the system's behaviour.

8.5 Summarizing the implementation

The aim of this implementation of the production cell was not to have the most efficient and the safest solution. Trying the different possibilities of modelling with the CAMOS(L) language was the main target. Implementing the classes with a simple, serialized operation, having only one blank in the production cell, took about 15 hours. Creating the complex operations to work with the maximum amount of blanks in the production cell needed another 10 hours. So the whole implementation was done in about five days.

In general, I tried to model the production cell according to the rules from the modelling techniques chapter. The design of some DFAs (e.g. from the robot arm) was arbitrary, there are also other possibilities of designing them, perhaps with greater security, but the implemented DFAs work properly. Modelling the robot as container class with separate DFAs for the magnets, the robot arms and the robot base made the modelling of the appropriate classes easier.

Using small, but more and possibly parallel working operations is the most flexible design, although the problem of concurrency became more important compared to using less and serialized operations.

Chapter 9

Summary of the thesis

The thesis has studied how the MCS CAMOS can be used for modelling a production cell of moderate size. It has also showed methods to satisfy the requirements on flexibility and safety.

The task level language CAMOS(L) provides compact code and is easy and fast to program, because it has only few syntactical constructs and presents all data via a single access mechanism, the database query language. Modelling the machines in the production cell with the object-oriented techniques presents an abstract, but similarly structured view of the real world. The hierarchy of classes and operations helps the task level programmer to structure the modelling of classes and operations.

Separating the class modelling from the operation modelling increases the flexibility of CAMOS, since the task of the machines in the production cell can be enhanced or changed without affecting the class modelling of the machines. If the task level programmer keeps the operations local, a small change in the task of few machines results also in a small effort to change only the affected operations.

CAMOS(L) supports two level of safety: DFAs and synchronization techniques. The design of DFAs for each machine makes the basic behaviour of machines with DFAs independent from programming the operations. Programming operations cannot bypass safety restrictions in DFAs. The synchronization techniques restrict the execution of operations to a definite and sequential order.

Problems remain with the concurrency between operations. This thesis has presented rules to avoid these problems for production cells of moderate size. For larger projects, it will be difficult to handle these problems manually.

Chapter 10

Future work

As the first part of this thesis was writing a compiler for CAMOS(L), the first suggestions for future work relate to the compiler. As shown in section 6.1 about the syntax of CAMOS(L), the syntax for the activation of workers is not complete, because it makes use of internal LISP functions to access the objects. An own construct in CAMOS(L) could replace this way of activating workers. The basic types, used for state variables, could be enhanced with booleans and real values. Furthermore, the overloading of functions should be implemented to complete the system. Finally, the compiler could be included in CAMOS to have one single system.

The second part is about problems with the semantics of CAMOS(L), which are not solved: the concurrency problem between operations. I suggest to introduce a further locking mechanism, perhaps in the formal parameter list, to lock objects from the formal parameter list, when this object should be changed. The formal parameter are then separated in read-only and writable parameter. This should avoid then that another operation executes in parallel, which also wants to have writing access.

Then there remain also problems with the implementation of CAMOS, for example with the stability locking mechanism. When database queries are used as values for assignments in primitive suboperations, the stability of objects from the formal parameter list is not checked. Also, the scheduler in the operation manager could be improved for better throughput of the system. Maintenance algorithm for single nodes are also not possible, because active rules are not activated, when the start node and the end node of an transition are the same.

Another remaining problem is, how to model sensors in the real world that do not guide control algorithms but provide important information such as an emergency switch.

As there is no implementation of the control level in CAMOS, an implementation of it could make the system usable for other purposes than this special production cell. A formal development, as demanded in the case study, could make the CAMOS system better comparable to the other systems in the case study and could prove the satisfaction of the safety requirements.

Finally, properties such as support of dynamical changes within the machinery (see section 3.2.5) could be added to CAMOS. This includes constructs in order to activate or deactivate operations while the other operations are still running.

Appendix A

Abbreviations

ADBMS: Active Database Management System

AMOS: Active Mediator Object System

AMOSQL: AMOS extension to OSQL

ARAMIS: A Robot And Manufacturing Instruction System

CAMOS: Control Application Mediator Object System

CAMOS(L): Control Application Mediator Object System Language

DFA: Deterministic Finite State Automaton

er-table: elevating rotary table

FZI: Forschungzentrum Informatik in Karlsruhe

MCS: Manufacturing Control System

OSQL: Object-oriented Structured Query Language

SQL: SeQueL: Query language

TCL: Tool Command Language

WM: World Model

WO: World Object

VM: Virtual Machine

Appendix B

EBNF of the CAMOS(L) syntax

In this appendix, the EBNF for the CAMOS(L) syntax is described. Keywords are written in *italic* style.

B.1 General constructs

Here are the syntactical elements used by classes, object and operations.

```
IDENTIFIER ::=          [_a-zA-Z][_a-zA-Z0-9]*

CAMOS-IDENTIFIER ::=    '.'IDENTIFIER

CONSTANT ::=            NUMBER | STRING

NUMBER ::=              {DIGIT}+

LABEL ::=               NUMBER':'

DIGIT ::=               0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

STRING ::=              '"' [a-zA-Z0-9_ ] '"'
```

B.2 The class definition

Here are the syntactical elements for defining classes.

```
CLASS-DEFINITION ::=      CREATE CAMOSCLASS CLASSNAME
                          SUBTYPE OF
                          SUPERCLASS-COMMA-LIST
                          (STATEVARS':'
                          MEMBER-DECLARATION-COMMA-LIST)opt
                          (OBJECTVARS':'
                          MEMBER-DECLARATION-COMMA-LIST)opt
                          (DBVARS':'
                          MEMBER-DECLARATION-COMMA-LIST)opt
                          (NODES':'
                          NODE-DECLARATION-COMMA-LIST)opt
                          (TRANSITIONS':'
                          TRANS-DECLARATION-COMMA-LIST)opt
                          ';'
CLASSNAME ::=             IDENTIFIER
SUPERCLASS ::=            IDENTIFIER
MEMBER-DECLARATION ::=    MEMBERFUNCTIONNAME ':' TYPE (
                          '=' DEFAULTVALUE)opt
MEMBERFUNCTIONNAME ::=    IDENTIFIER
DEFAULTVALUE ::=          CONSTANT
NODE-DECLARATION ::=      NUMBER NODENAME ':' NODECONSTRAINT
NODENAME ::=              STRING
NODECONSTRAINT ::=        AMOSQL expression with CAMOS
                          identifier, returning a boolean
                          result
TRANS-DECLARATION ::=     NUMBER '->' NUMBER ':'
                          CONTROLALGORITHM
                          OBJECT-REFERENCE
OBJECT-REFERENCE ::=      AMOSQL expression with CAMOS
                          identifier, returning a CAMOS
                          object
CONTROLALGORITHM ::=      STRING
```

B.3 The object definition

This section contains the syntactical constructs for the creation of an object with CAMOS(L).

```
OBJECT-DEFINITION ::=  CREATE OBJECT CLASSNAME '('
                       MEMBERFUNCTIONNAME-COMMA-LIST
                       ')' OBJECTNAME '('
                       INITVALUE-COMMA-LIST ')' ';'

OBJECTNAME ::=         IDENTIFIER

INITVALUE ::=          OBJECTNAME | CONSTANT
```

B.4 The operation definition

Here are the syntactical constructs for defining an operation with CAMOS(L).

`OPERATION-DEF ::=`	*CREATE OPERATION* `OPNAME '(' PARAMETER-COMMA-LIST ')'` *AS WHEN* `WAITCONDITION` *DO* `IF-STATEMENT-LIST`
`OPNAME ::=`	`IDENTIFIER`
`PARAMETER ::=`	`CAMOS-CLASS CAMOS-IDENTIFIER`
`CAMOS-CLASS ::=`	`IDENTIFIER`
`WAITCONDITION ::=`	`AMOSQL expression with CAMOS identifier, returning a boolean value`
`IF-STATEMENT ::=`	*IF* `IFCLAUSE` *THEN* `SUBOPERATION-LIST` *ENDIF*
`IFCLAUSE ::=`	`AMOSQL expression with CAMOS identifier, returning a boolean value`
`SUBOPERATION ::=`	`{LABEL {MULTIPUTSUBOP | ACTIONCALL | DBSUBBOP } ';'} |` *DEPENDENCY* `':' DEPENDENCY-COMMA-LIST ';'`
`MULTIPUTSUBOP ::=`	*PUT* `CAMOS-IDENTIFIER ':' PUTOP-COMMA-LIST`
`PUTOP ::=`	`MEMBERFUNCTION '=' CAMOS-VALUE`
`MEMBERFUNCTION ::=`	`IDENTIFIER`
`CAMOS-VALUE ::=`	`CONSTANT | '((' AMOSQL expression with CAMOS identifier '))'`
`ACTIONCALL ::=`	`ACTIONNAME (CAMOS-OBJECT-COMMA-LIST)`
`ACTIONNAME ::=`	`IDENTIFIER`
`CAMOS-OBJECT ::=`	`CAMOS-IDENTIFIER | '((' AMOSQL expression with CAMOS identifier, returning a CAMOS object'))'`
`DBSUBOP ::=`	*SET* `CAMOS-IDENTIFIER ':' PUTOP`
`DEPENDENCY ::=`	`(NUMBER, NUMBER)`

Appendix C

The CAMOS(L) source for the production cell

```
/* some common base classes */

CREATE CLASS actuator SUBTYPE OF camosobject;

CREATE CLASS rotator SUBTYPE OF actuator
    STATEVARS: rotation: integer = 0;

CREATE CLASS elevator SUBTYPE OF actuator
    STATEVARS: elevation: integer = 0;

/* ***** the base class for both belts **************** */

CREATE CLASS belt SUBTYPE OF actuator
    STATEVARS: loadedend: integer = 0,
               loadedbeg: integer = 0; /* virtual sensor */

CREATE OPERATION moveToEnd (belt .b)
AS WHEN
  loadedbeg (.b) = 1 AND loadedend (.b) = 0
DO
IF true THEN
    1: PUT .b: loadedbeg = 0,
               loadedend = 1;
ENDIF
```

```
/****************** the feeding conveyor belt *****************/

CREATE CLASS feedbelt SUBTYPE OF belt
    STATEVARS: pileEmpty: integer = 0  /* virtual sensor */
    DBVARS: blanksfrompile: integer = 0
    NODES:
        1 "empty, feed from pile"
        ((loadedend (.obj) = 0 AND loadedbeg (.obj) = 0
            AND pileEmpty (.obj) = 0)),
        2 "loaded at the beginning"
        ((loadedend (.obj) = 0 AND loadedbeg (.obj) = 1)),
        3 "loaded at the end, feed from pile"
        ((loadedend (.obj) = 1 AND loadedbeg (.obj) = 0
            AND pileEmpty (.obj) = 0)),
        4 "loaded at both"
        ((loadedend (.obj) = 1 AND loadedbeg (.obj) = 1)),
        5 "empty, feed from crane"
        ((loadedend (.obj) = 0 AND loadedbeg (.obj) = 0
            AND pileEmpty (.obj) = 1)),
        6 "loaded at the end, feed from crane"
        ((loadedend (.obj) = 1 AND loadedbeg (.obj) = 0
            AND pileEmpty (.obj) = 1))
    TRANSITIONS:
            1 -> 2 "addBlankFromPile" ((.obj)),
            2 -> 3 "beltMoveToEnd" ((.obj)),
            3 -> 4 "addBlankFromPile" ((.obj)),
            4 -> 2 "beltMoveToUnload" ((.obj)),
            3 -> 1 "beltMoveToUnload" ((.obj)),

            5 -> 2 "addBlankFromCrane" ((.obj)),
            2 -> 6 "beltMoveToEnd" ((.obj)),
            6 -> 4 "addBlankFromCrane" ((.obj)),
            6 -> 5 "beltMoveToUnload" ((.obj)),

            1 -> 5 "switchMode" ((.obj)),
            3 -> 6 "switchMode" ((.obj));

CREATE OBJECT feedbelt(currentstate, loadedbeg,
                       loadedend, pileEmpty)
```

```
            feedbelt (1, 0, 0, 0);

CREATE OPERATION addBlank (feedbelt .b)
AS WHEN
  loadedbeg (.b) = 0 AND pileEmpty (.b) = 0
    AND blanksfrompile (.b) < 5
DO
IF true THEN
    1: PUT .b: loadedbeg = 1;
    2: SET .b: blanksfrompile = (( blanksfrompile (.b) + 1 ));
    DEPENDENCIES: (1,2);
ENDIF

CREATE OPERATION guardPile (feedbelt .b)
AS WHEN
    loadedbeg (.b) = 0 AND pileEmpty (.b) = 0
      AND blanksfrompile (.b) = 5
DO
IF blanksfrompile (.b) = 5 AND pileEmpty (.b) = 0 THEN
    1: PUT .b: pileEmpty = 1;
ENDIF

/* ************** the elevating rotary table ************** */

CREATE CLASS er_table SUBTYPE OF rotator, elevator
    DBVARS: ertloaded: integer = 0
    NODES:
        1 "At the feedbelt"
        ((rotation (.obj) = 0 AND elevation (.obj) = 0)),
        2 "At the robot"
        ((rotation (.obj) = 50 AND elevation (.obj) = 1))
    TRANSITIONS:
            1 -> 2 "ertGoRobot" ((.obj)),
            2 -> 1 "ertGoBelt" ((.obj));

CREATE OBJECT er_table(currentstate, rotation, elevation)
            ert (1, 0, 0);

CREATE OPERATION returnTable (er_table .e)
AS WHEN
```

```
  ertloaded (.e) = 0 AND rotation (.e) != 0
    AND elevation (.e) != 0
DO
IF rotation (.e) != 0 AND elevation (.e) != 0 THEN
    1: PUT .e: rotation = 0,
               elevation = 0;
ENDIF

/********* operations between ert and feed belt ***************/

CREATE OPERATION moveToErt (feedbelt .b, er_table .e)
AS WHEN
  rotation (.e) = 0 AND elevation (.e) = 0
    AND loadedend (.b) = 1 AND ertloaded (.e) = 0
DO
IF true THEN
    1: PUT .b: loadedend = 0;
    2: SET .e: ertloaded = 1;
    DEPENDENCIES: (1,2);
ENDIF

/* ************************* the magnets ********************** */

CREATE CLASS magnet SUBTYPE OF actuator
    STATEVARS: magnetstate: integer = 0
    NODES:
        1 "magnet off" ((magnetstate (.obj) = 0)),
        2 "magnet on" ((magnetstate (.obj) = 1))
    TRANSITIONS:
            1 -> 2 "magnetOn" ((.obj)),
            2 -> 1 "magnetOff" ((.obj));

CREATE OBJECT magnet(currentstate) arm1magnet (1);
CREATE OBJECT magnet(currentstate) arm2magnet (1);

/* access operations for the magnet class */
CREATE OPERATION magnetOn (magnet .m)
AS WHEN
  true
DO
```

```
IF magnetstate(.m) != 1 THEN
    1: PUT .m: magnetstate = 1;
ENDIF

CREATE OPERATION magnetOff (magnet .m)
AS WHEN
  true
DO
IF magnetstate(.m) != 0 THEN
    1: PUT .m: magnetstate = 0;
ENDIF

/* ******** the robotarms ***************************** */

CREATE CLASS robot_arm SUBTYPE OF actuator
    STATEVARS: extension: integer = 0
    OBJECTVARS: armmagnet: magnet
    DBVARS: armloaded: integer = 0
    NODES:
        1 "safe position" ((extension (.obj) = 0)),
        2 "back position" ((extension (.obj) = 1)),
        3 "front position" ((extension (.obj) = 2))
    TRANSITIONS:
            1 -> 2 "armext 1" ((.obj)),
            2 -> 1 "armext 0" ((.obj)),
            1 -> 3 "armext 2" ((.obj)),
            3 -> 1 "armext 0" ((.obj)),
            2 -> 3 "armext 2" ((.obj)),
            3 -> 2 "armext 1" ((.obj));

CREATE OBJECT robot_arm (extension, currentstate, armmagnet)
    armErt (0, 1, arm1magnet);
CREATE OBJECT robot_arm (extension, currentstate, armmagnet)
    armBelt (1, 2, arm2magnet);

/* access operations for the robot arm class */
CREATE OPERATION extendArm (robot_arm .ra)
AS WHEN
  true
DO
```

```
IF extension (.ra) != 2 THEN
    1: PUT .ra: extension = 2;
ENDIF

CREATE OPERATION retractArm (robot_arm .ra)
AS WHEN
  true
DO
IF extension (.ra) != 1 THEN
    1: PUT .ra: extension = 1;
ENDIF

CREATE OPERATION saveArm (robot_arm .ra)
AS WHEN
  true
DO
IF extension (.ra) != 0 THEN
    1: PUT .ra: extension = 0;
ENDIF

/* access operations for the database variable armloaded */
CREATE OPERATION setLoaded (robot_arm .ra)
AS WHEN
  true
DO
IF armloaded (.ra) = 0 THEN
    1: SET .ra: armloaded = 1;
ENDIF

CREATE OPERATION setUnloaded (robot_arm .ra)
AS WHEN
  true
DO
IF armloaded (.ra) = 1 THEN
    1: SET .ra: armloaded = 0;
ENDIF

/* ****************** the robot ********************* */

CREATE CLASS robot SUBTYPE OF rotator
```

```
    OBJECTVARS: arm1: robot_arm,
                arm2: robot_arm
    NODES:
        1 "initital position" ((rotation (.obj) = 0)),
        2 "arm1 at er-table" ((rotation (.obj) = 50)),
        3 "arm1 at press" ((rotation (.obj) = -90)),
        4 "arm1 out of press" ((rotation (.obj) = -65)),
        5 "arm2 at press" ((rotation (.obj) = 35)),
        6 "arm2 at deposite belt" ((rotation (.obj) = -55))
    TRANSITIONS:
            1 -> 2 "robrot 50" ((.obj)),
            2 -> 5 "robrot 35" ((.obj)),
            5 -> 6 "robrot -55" ((.obj)),
            6 -> 3 "robrot -90" ((.obj)),
            3 -> 4 "robrot -65" ((.obj)),
            4 -> 2 "robrot 50" ((.obj))
;

CREATE OBJECT robot (currentstate, rotation, arm1, arm2)
    robot (1, 0, armErt, armBelt);

CREATE OPERATION initRobot (robot .r)
AS WHEN
  true
DO
IF rotation (.r) != 50 THEN
    1: saveArm ( ((arm1 (.r))) );
    2: saveArm ( ((arm2 (.r))) );
    3: PUT .r: rotation = 50;
    DEPENDENCIES: (1,3), (2,3);
ENDIF

/* access operations for the robot base, partly enhanced by
   additional wait conditions to ensure safety requirements */

CREATE OPERATION rotateToErt (robot .r)
AS WHEN
  extension (arm2 (.r)) = 0 /* arm2 and press could collide */
DO
IF rotation (.r) != 50 THEN
```

```
    1: PUT .r: rotation = 50;
ENDIF

CREATE OPERATION rotateArm1ToPress (robot .r)
AS WHEN
  extension (arm1 (.r)) != 2 /* arm1 and press could collide */
DO
IF rotation (.r) != -90 THEN
    1: PUT .r: rotation = -90;
ENDIF

CREATE OPERATION rotateArm1OutPress (robot .r)
AS WHEN
  true
DO
IF rotation (.r) != -65 THEN
    1: PUT .r: rotation = -65;
ENDIF

CREATE OPERATION rotateArm2ToPress (robot .r)
AS WHEN
  extension (arm2 (.r)) = 0  /* arm2 and press could collide */
DO
IF rotation (.r) != 35 THEN
    1: PUT .r: rotation = 35;
ENDIF

CREATE OPERATION rotateToBelt (robot .r)
AS WHEN
  true
DO
IF rotation (.r) != -55 THEN
    1: PUT .r: rotation = -55;
ENDIF

CREATE OPERATION rotateToInit (robot .r)
AS WHEN
  true
DO
IF rotation (.r) != 0 THEN
```

```
    1: PUT .r: rotation = 0;
ENDIF

/* ************* operations between robot and er_table ****** */

CREATE OPERATION turnTable (er_table .e, robot .r)
AS WHEN
  ertloaded (.e) = 1 AND  rotation(.e) != 50 AND elevation(.e) != 1
    AND armloaded (arm1 (.r)) = 0 /* avoid collision of loaded arm1
                                     with the loaded er_table */
DO
IF rotation(.e) != 50 AND elevation(.e) != 1 THEN
    1: PUT .e: rotation = 50,
           elevation = 1;
ENDIF

CREATE OPERATION loadArm1 (robot .r, er_table .e)
/* not taking the robot arm as paramter to be sure, that the
   robot is not moving, when entering this operation */
AS WHEN
   ertloaded (.e) = 1 AND rotation (.e) = 50 AND elevation (.e) = 1
   AND armloaded (arm1 (.r)) = 0 AND rotation (.r) = 50
DO
IF true THEN
    1: retractArm ( ((arm1 (.r))) );
    2: magnetOn ( ((armmagnet (arm1 (.r)) )) );
    3: SET .e: ertloaded = 0;
    4: setLoaded ( ((arm1 (.r))) );
    DEPENDENCIES: (1, 2), (2, 4), (2, 3);
ENDIF

/* ******** the press ******************* */

CREATE CLASS press SUBTYPE OF actuator
    STATEVARS: position: integer = 1
    DBVARS: loaded: integer = 0
    NODES:
        1 "down position" ((position (.obj) = 0)),
        2 "middle position" ((position (.obj) = 1)),
        3 "upper position" ((position (.obj) = 2))
```

```
    TRANSITIONS:
            1 -> 2 "pressMiddle" ((.obj)),
            2 -> 3 "pressUp" ((.obj)),
            3 -> 1 "pressDown" ((.obj));

CREATE OBJECT press (currentstate, position) press (2, 1);

/* access operations for the press */
CREATE OPERATION pressDown (press .p)
AS WHEN
  true
DO
IF position(.p) != 0 THEN
    1: PUT .p: position = 0;
ENDIF

CREATE OPERATION pressMiddle (press .p)
AS WHEN
  true
DO
IF position(.p) != 1 THEN
    1: PUT .p: position = 1;
ENDIF

CREATE OPERATION pressUp (press .p)
AS WHEN
  true
DO
IF position(.p) != 2 THEN
    1: PUT .p: position = 2;
ENDIF

/* ***** operations between robot and press ************* */

CREATE OPERATION loadArm2 (robot .r, press .p)
AS WHEN
    rotation (.r) = 35 AND position (.p) = 0
DO
IF armloaded (arm2 (.r)) = 0 THEN
    1: extendarm ( ((arm2 (.r))) );
```

```
    2: magnetOn ( ((armMagnet (arm2 (.r)))) );
    3: setLoaded ( ((arm2 (.r))) );
    4: SET .p: loaded = 0;
    DEPENDENCIES: (1,2), (2,3), (2,4);
ENDIF

CREATE OPERATION unloadArm1 (robot .r, press .p)
AS WHEN
    rotation (.r) = -90 AND position (.p) = 1 AND loaded (.p) = 0
DO
IF armloaded (arm1 (.r)) = 1 THEN
    1: extendarm ( ((arm1 (.r))) );
    2: magnetOff ( ((armMagnet (arm1 (.r)))) );
    3: setUnloaded ( ((arm1 (.r))) );
    4: SET .p: loaded = 1;
    DEPENDENCIES: (1,2), (2,3), (2,4);
ENDIF

CREATE OPERATION pressBlank (press .p, robot .r)
AS WHEN
    rotation (.r) = -65 AND loaded (.p) = 1 AND position (.p) = 1
    /* the robot and the press could collide */
DO
IF true THEN
    1: pressUp (.p);
    2: pressDown (.p);
    DEPENDENCIES: (1,2);
ENDIF

CREATE OPERATION pressReturn (press .p, robot .r)
AS WHEN
    loaded (.p) = 0 AND position (.p) = 0 AND rotation (.r) = -55
    /* the robot and the press could collide */
DO
IF true THEN
    1: PUT .p: position = 1;
ENDIF

/* ****************** the deposit conveyor belt ************* */
```

```
CREATE CLASS depbelt SUBTYPE OF belt
    STATEVARS: atunload : integer = 0 /* virtual sensor */
    NODES:
        1 "empty"
        ((loadedbeg (.obj) = 0 AND loadedend (.obj) = 0
            AND atunload (.obj) = 0)),
        2 "loaded at the beginning"
        ((loadedbeg (.obj) = 1 AND loadedend (.obj) = 0
            AND atunload (.obj) = 0)),
        3 "loaded at the end"
        ((loadedbeg (.obj) = 0 AND loadedend (.obj) = 1
            AND atunload (.obj) = 0)),
        4 "at unloadposition"
        ((loadedbeg (.obj) = 0 AND loadedend (.obj) = 0
            AND atunload (.obj) = 1)),
        5 "loaded twice"
        ((loadedbeg (.obj) = 1 AND loadedend (.obj) = 1
            AND atunload (.obj) = 0)),
        6 "loaded twice, at unload"
        ((loadedbeg (.obj) = 1 AND loadedend (.obj) = 0
            AND atunload (.obj) = 1))
    TRANSITIONS:
            1 -> 2 "addBlankFromCrane" ((.obj)),
            2 -> 3 "beltMoveToEnd" ((.obj)),
            3 -> 4 "beltMoveToUnload" ((.obj)),
            3 -> 5 "addBlankFromCrane" ((.obj)),
            4 -> 1 "unloadBlank" ((.obj)),
            4 -> 6 "addBlankFromCrane" ((.obj)),
            5 -> 6 "beltMoveToUnload" ((.obj)),
            6 -> 2 "unloadBlank" ((.obj));

CREATE OBJECT depbelt (currentstate, loadedbeg, loadedend)
    depbelt (1, 0, 0);

CREATE OPERATION depMoveToEnd (depbelt .db)
AS WHEN
    atunload (.db) = 0 AND loadedbeg (.db) = 1
DO
IF true THEN
    1: PUT .db: loadedend = 1,
```

```
                loadedbeg = 0;
ENDIF

CREATE OPERATION moveToCrane (depbelt .db)
AS WHEN
    atunload (.db) = 0 AND loadedend (.db) = 1
DO
IF true THEN
    1: PUT .db: loadedend = 0,
                atunload = 1;
ENDIF

CREATE OPERATION unloadBlank (depbelt .db)
AS WHEN
  atunload (.db) = 1
DO
IF atunload (.db) = 1 THEN
    1: PUT .db: atunload = 0;
ENDIF

/* ****** operations between robot and deposit belt ******** */

CREATE OPERATION unloadArm2 (robot .r, depbelt .db)
AS WHEN
    rotation (.r) = -55 AND loadedbeg (.db) = 0
DO
IF armloaded (arm2 (.r)) = 1 THEN
    1: retractArm ( ((arm2 (.r))) );
    2: magnetOff ( ((armMagnet (arm2 (.r)))) );
    3: setUnloaded ( ((arm2 (.r))) );
    4: PUT .db: loadedbeg  = 1;
    DEPENDENCIES: (1,2),(2,3),(2,4);
ENDIF

/* * the whole robot (with er_table, press and deposit belt) * */

CREATE OPERATION operateRobot (robot .r, er_table .e,
                               press .p, depbelt .db)
AS WHEN
  ertloaded (.e) = 1 AND rotation (.e) = 50 AND elevation (.e) = 1
```

```
   AND rotation (.r) = 50
DO
IF true THEN
    1: loadArm1 (.r, .e);
    2: rotateArm2ToPress (.r);
    DEPENDENCIES: (1,2);
ENDIF
IF loaded (.p) = 1 THEN
    3: loadArm2 (.r, .p);
    DEPENDENCIES: (2,3), (3,4);
ENDIF
IF loaded (.p) = 0 THEN
    DEPENDENCIES: (2,4);
ENDIF
IF true THEN
    4: rotateToBelt (.r);
    5: unloadArm2 (.r, .db);
    6: saveArm ( ((arm2 (.r))) );
    7: rotateArm1ToPress (.r);
    8: unloadArm1 (.r, .p);
    9: rotateArm1OutPress (.r);
    10: retractArm ( ((arm1 (.r))) );
    11: rotateToErt (.r);
    DEPENDENCIES: (4,5), (5,6), (5,7), (7,8),
                  (8,9), (8,10), (10,11);
ENDIF


/* ****************** the crane ********************** */

CREATE CLASS crane SUBTYPE OF elevator
    STATEVARS:     h_position: integer = 0
    OBJECTVARS:    craneMagnet: magnet
    DBVARS:        craneloaded: integer = 0
    NODES:
        1 "Init"
        ((h_position (.obj) = 0)),
        2 "At depBelt and down"
        ((h_position (.obj) = 2 AND elevation (.obj) = 0)),
        3 "At depBelt and up"
```

```
        ((h_position (.obj) = 2 AND elevation (.obj) = 1)),
        4 "At feedbelt and up"
        ((h_position (.obj) = 1 AND elevation (.obj) = 1))
    TRANSITIONS:
            1 -> 3 "craneToDepbelt" ((.obj)),
            2 -> 3 "craneUp" ((.obj)),
            3 -> 4 "craneToFeedbelt" ((.obj)),
            4 -> 3 "craneToDepbelt" ((.obj)),
            3 -> 2 "craneDown" ((.obj));

CREATE OBJECT magnet (currentstate) cranemag (1);
CREATE OBJECT crane (currentstate, craneMagnet, elevation)
    crane (1, cranemag, 1);

CREATE OPERATION initCrane (crane .c)
/* this operation is needed for moving the crane
   to its initial position above the deposit belt;
   its position, when starting the TCL simulation
   is somewhere between both belts... */
AS WHEN
    h_position (.c) = 0
DO
IF h_position (.c) != 2 THEN
    1: PUT .c: h_position = 2;
ENDIF
IF elevation (.c) != 0 THEN
    2: PUT .c: elevation = 0;
ENDIF
IF h_position (.c) != 2 AND elevation(.c) != 0 THEN
    DEPENDENCIES: (1,2);
ENDIF

/* *** operations between the crane and the deposit belt ******* */

CREATE OPERATION getBlankfromDep (crane .c, depbelt .db)
AS WHEN
    atunload (.db) = 1 AND craneloaded (.c) = 0
    AND h_position (.c) = 2 AND elevation (.c) = 0
DO
IF true THEN
```

```
    1: magnetOn ( ((craneMagnet (.c))) );
    2: SET .c: craneloaded = 1;
    3: PUT .c: elevation = 1;
    4: PUT .db: atunload = 0;
    5: PUT .c: h_position = 1;
    DEPENDENCIES: (1,2), (1,3), (3,4), (3,5);
ENDIF

/* *** operations between the crane and the feed belt ******** */

CREATE OPERATION putBlankOnFeedAndReturn (crane .c, feedbelt .f)
AS WHEN
    loadedbeg (.f) = 0 AND craneloaded (.c) = 1
    AND h_position (.c) = 1 AND elevation (.c) = 1
    AND pileEmpty (.f) = 1
DO
IF true THEN
    2: magnetOff ( ((craneMagnet (.c))) );
    3: SET .c: craneloaded = 0;
    4: PUT .f: loadedbeg = 1;
    5: PUT .c: h_position = 2;
    6: PUT .c: elevation = 0;
    DEPENDENCIES: (2,3), (2,4), (4,5), (5,6);
ENDIF


:AMOSQL
activateworker (addblank(), [getfunction (".feedbelt")], "cyclic");
activateworker (guardPile(), [getfunction (".feedbelt")], "");
activateworker (moveToEnd(), [getfunction (".feedbelt")],"cyclic");
activateworker (moveToErt(), [getfunction (".feedbelt"),
                              getfunction (".ert")], "cyclic");
activateworker (turntable(), [getfunction (".ert"),
                              getfunction (".robot")], "cyclic");
activateworker (returntable(), [getfunction (".ert")], "cyclic");

activateworker (initRobot(), [getfunction (".robot")], "");
activateworker (operateRobot(), [getfunction (".robot"),
                                 getfunction (".ert"),
                                 getfunction (".press"),
```

```
                                getfunction (".depbelt")],
                                "cyclic");

activateworker (pressBlank(), [getfunction (".press"),
                              getfunction (".robot")],"cyclic");
activateworker (pressReturn(), [getfunction (".press"),
                               getfunction (".robot")], "cyclic");

activateworker (depMoveToEnd (), [getfunction (".depbelt")],
                                "cyclic");
activateworker (moveToCrane (), [getfunction (".depbelt")],
                               "cyclic");

activateworker (initCrane (), [getfunction (".crane")], "");
activateworker (getBlankFromDep (), [getfunction (".crane"),
                                    getfunction (".depbelt")],
                                    "cyclic");
activateworker (putBlankOnFeedAndReturn (),
               [getfunction (".crane"), getfunction (".feedbelt")],
               "cyclic");

lisp;
(setq *trace0* nil)
(setq *trace* nil)
(setq _debugging_ 1)
(if *trace*
  (progn
;;    (trace runPrimitive)
    (setq _debugging_ t))
)
:osql
:CAMOS
```

References

[Cat91] Cattell, Roderic G. G. (1991), *Object data management: object-oriented and extended relational database systems.* Addison-Wesley, Reading, Massachusetts, U.S.A.

[Elm89] Elmasri, R., Navathem, S. B. (1989), *Fundamentals of Database Systems.* Addison-Wesley, Reading, Massachusetts, U.S.A.

[Fal96] Falkenroth, Esa (1996), *Data Management in Control Applications – A Proposal Based on Active Database Systems.* Linköping University, Linköping, Sweden.

[Kar94] Karlsson, Jonas S., Larsson, Staffan, Risch, Tore, Sköld, Martin, Werner, Magnus (1994), *AMOS User's Guide.* CAELAB Memo 94-01, Linköping University, Linköping, Sweden.
See also URL: http://www.ida.liu.se/labs/edslab/

[Lew94] Lewerentz, Claus, Lindner, Thomas (Editors) (1994), *Case Study "Production Cell" – A Comparative Study in Formal Software Development.* FZI-Publication 1/94, Forschungszentrum Informatik an der Universität Karlsruhe, Karlsruhe, Germany. See also URL:
http://www.fzi.de/prost/projects/production_cell/ProductionCell.html

[Lob94] Loborg, Peter (1994), *Error Recovery Support in Manufacturing Control Systems.* Linköping University, Linköping, Sweden.

[LobHol92] Loborg, Peter, Holmbom, Per, Sköld, Martin, Törne, Anders (1992), *Programming Flexible Manufacturing System.* CAELAB Memo 92-02, Linköping University, Linköping, Sweden.

[LobSkö92] Loborg, Peter, Sköld, Martin, Törne, Anders (1992), *A hierarchical software architecture for control of industrial robots and manufactur-*

ing equipment. CAELAB Memo 92-01, Linköping University, Linköping, Sweden.

[LobTör94] Loborg, Peter, Törne, Anders (1994), *Manufacturing Control System Principles supporting Error Recovery.* Linköping University, Linköping, Sweden.

[WidCer96] Widom, Jennifer, Ceri, Stefano (1996), *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann Publishers, San Francisco, U.S.A.