A Quantity Scheduling Language for Manufacturing Systems

by

Joseph Edward Nemec

B.A. in Mathematics, Manhattanville College, 1992

Submitted to the Operations Research Center
in partial fulfillment of the requirements for the degree of

Master of Science in Operations Research

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1996

Author . . . .
Operations Research Center
May 6, 1996

Certified by . . . .
Stanley B. Gershwin
Senior Research Scientist, Department of Mechanical Engineering
Thesis Supervisor

Certified by . . . .
Donald E. Troxel
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Thomas L. Magnanti
Professor of Management Science

A Quantity Scheduling Language for Manufacturing Systems

by

Joseph Edward Nemec

Submitted to the Operations Research Center
on May 6, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Operations Research

Abstract

We propose the syntax and supporting structure of a language that allows a programmer to devise and implement scheduling policies for manufacturing systems that make large numbers of identical parts. In the course of the description of the language, we define various methods of representing a factory, including the equipment and personel in the factory, the process flows of parts and constraints on the production of parts. We take object-oriented approach to the definitions of the various components of the factory model. With the factory well-defined, we then describe the structure and syntax of a language that allows for the implementation of scheduling policies, such as Kanban and CONWIP [**?**]. A running example of a five-machine transfer line is used throughout the thesis to illustrate the constructs defined. Last, we describe the structure of a scheduler based on the constructs developed that takes a real-time control perspective.

Thesis Supervisor: Stanley B. Gershwin
Title: Senior Research Scientist, Department of Mechanical Engineering

Thesis Supervisor: Donald E. Troxel
Title: Professor of Electrical Engineering and Computer Science

A Quantity Scheduling Language for Manufacturing Systems

by

Joseph Edward Nemec

Submitted to the Operations Research Center
on May 6, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Operations Research

Abstract

We propose the syntax and supporting structure of a language that allows a programmer to devise and implement scheduling policies for manufacturing systems that make large numbers of identical parts. In the course of the description of the language, we define various methods of representing a factory, including the equipment and personel in the factory, the process flows of parts and constraints on the production of parts. We take object-oriented approach to the definitions of the various components of the factory model. With the factory well-defined, we then describe the structure and syntax of a language that allows for the implementation of scheduling policies, such as Kanban and CONWIP [**?**]. A running example of a five-machine transfer line is used throughout the thesis to illustrate the constructs defined. Last, we describe the structure of a scheduler based on the constructs developed that takes a real-time control perspective.

Thesis Supervisor: Stanley B. Gershwin
Title: Senior Research Scientist, Department of Mechanical Engineering

Thesis Supervisor: Donald E. Troxel
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Science is not a solitary endeavor, and I received an enormous amount of support from people both at MIT and outside of MIT. All contributed to this thesis, whether they worked on it directly or not, and all deserve credit for its completion.

I wish to thank Stan Gershwin and Don Troxel for the kind opportunity to do this work that they have given me. Through their patient tutelage I have gained knowledge, experience, and friendship.

From Professor Troxel's research group I have made many friends who have broaded my level of understanding of computer science. Kind thanks go to Yonald Chery for his close friendship, patientence in revealing the secrets of IC fabrication, and running a tight ship at the Thirsty Ear. Without Myron "Fletch" Freeman and Will Moyne this thesis would never have been completed due wholly to my ability to crash the only computer with a complete copy. James Kao and Thomas Lohman provided kind support, and all conspired to make Thursday nights at the Pour House an enjoyable distraction from the rigors of research.

From Dr. Gershwin's research group, I thank Asbjoern Bonvik first and foremost. The time he took to answer my questions must surely have interfered with his work, yet he did so with patience and kindness. The excellent C code and Perl scripts he provided me with saved months of tedious coding. Mitchell Burman and Jim Schor deserve thanks for having provided me valuable factory experience by taking me along on some of the consulting jobs done by Analytics, Inc.

At the Operations Research Center, David Markowitz in particular helped me with his kind patience, excellent tutoring, and good friendship. I would not have made it through my classes in Queueing Theory and Advanced Stochastic Processes without him.

My friends outside of MIT have provided me with support and caring, and particular thanks go to Charles Terhune, Michael Puskar, Archie Smart, and Lucy von Brachel for putting up with me when the work at this place drove me to frustration.

Lastly, I wish to thank my family: my parents, my younger brothers John and

Michael, my two grandmothers, and my Aunt Marie and Uncle Ronnie. Without their support, love, and care I never would have made it to MIT at all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

This thesis to proposes the syntax and supporting framework for a language that allows for the simple implementation of various scheduling policies by an experienced computer programmer. The systems in question are repetitive manufacturing systems, that is, those that make large numbers of similar parts or products, such as microprocessor or automobile manufacturing. We are not concerned with custom or one-piece work, such as the production of luxury cruise liners. The language we propose is meant for use by programmers with experience in such scheduling policies as kanban and CONWIP [2]. In addition to allowing for the expression of various scheduling policies, we allow for the expression of various queueing disciplines, such as first-in first-out and priority scheduling. Our goal is to make the language sufficiently broad that new scheduling policies may be easily constructed and implemented, yet not so broad that a great deal of "busy work" is required to implement the scheduling policies developed. In particular, we have designed our language so that it can encompass some of the new hybrid control policies developed in [1].

1.2 Literature review

1.3 Terminology

The following is a glossary of the terms used in this thesis.

- *Action List:* The collection of all possible actions at generated by the scheduler for the purpose of selection.

- *Action Object:* A class definition representing changes in the physical state.

- *Attribute:* A field of a class definition, the attribute can be one of several types (integer, floating point number, another object).

- *Class definition:* An collection of attributes, each of which has a certain properties defined. Synonomous with *object*, the two terms are used interchangeably.

- *DR object*: A class definition in the demand representation.

- *Demand representation*: The collection of class definitions that describe the characteristics of demand for goods produced by the factory.

- *FR object*: A class definition in the factory representation.

- *Factory representation*: The collection of class definitions that describe the physical resources of the factory, including machines, buffers, and technicians, as well as conceptual groupings, such as manufacturing cells.

- *Initialization*: The initial instantiation of the objects in the various schema.

- *Instantiation:* A textual representation of a particular object or schema. Instantiation is the process of creating copies of objects into memory. Until an object is instantiated, it is simply an abstract data structure that has no meaning. Instantiation assigns values to each of the object attributes.

- *MR object*: A class definition in the materials representation.

- *Materials representation*: The collection of class definitions that describe the data which represents the material in the factory, including parts and raw materials.

- *Object:* An collection of attributes, each of which has a certain properties defined. Synonomous with *class definition*, the two terms are used interchangeably.

- *PR object*: A class definition in the process representation.

- *Physical state:* The factory state and materials state.

- *Possible Action:* An insantiation of an action object.

- *Predicate Definition:* Defined in an action object, these are the prerequisites for the possible action that is the instantiation of said action object to occur.

- *Process representation*: The collection of class definitions that describe the operations performed in the factory, including the prerequisite raw materials and technicians necessary for the performance of the operation.

- *Replication*: A copy of an instantiated object in memory.

- *Schema:* The collection of all objects in a particular representation of a factory. Synonymous with *class definition list*, the two terms are used interchangeably.

- *Syntax:* The grammar required for the expression of objects in the various representations.

1.4 Thesis Outline

In Chapter 2 we introduce the model that is a necessary prerequisite of the scheduling language. In Section 2.1, we introduce the idea of the five representations: the FR, or *factory representation*, the PR, or *process representation*, the DR, or *demand* *representation*, the MR, or *materials representation*, and the SR, or *scheduling representation*. We also introduce the three states: the FS, or *factory state*, and the MS, or *material state*, and the IS, or *information state*.

In Chapter 3 we describe, in detail, the structures and the syntaxes of the various representations. Section 3.1 introduces the physical representations in more detail. To illustrate the concepts of the thesis, the running example of a five-machine transfer line that makes two different part types is used. This is described in 3.1.1. Section 3.2 describes the general nature of the FR, and gives some examples of the possible forms which the objects can take. Section 3.2.1 the describes the syntax for the FR. An example of the FR is given in Section 3.2.4. In Section 3.3, we describe the PR. The syntax is defined in Section 3.3.1, and an example of the PR is given in Section 3.3.2. Section 3.4 describes the structure of the MR, and the syntax for the MR is defined in Section 3.4.1. Examples of the MR are given in Section 3.4.2. Finally, the structure for the DR is described in section 3.5, the syntax is defined in Section 3.5.1, and examples are of the DR are given in 3.5.2.

Chapter 4 describes the action objects, their purpose, and their relation to the representations defined in chapter 3. The syntax of the action objects is defined in section 4.0.3, and examples are given in section 4.0.4.

Chapter 2

The model

2.1 Overview

Our purpose is to define the structure and syntax of a language that allows for the easy coding by a programmer of policies to schedule a repetitive manufacturing system. In order to create such a language it is necessary to define what we mean by a "factory". Specifically, we need to devise a consistent means of describing what objects a "factory" consists of, and how these objects are related to one another.

We will use an object-oriented approach to the definition of the items in the factory similar to that used by [5]. Specifically, we will provide the structure and syntax of a language that allows for the declaration that certain objects exist, that these objects have certain attributes, and that the attributes themselves have certain properties. For example, if we have machines in the factory then we will declare that there exists and object called *machine*, that this object has certain attributes, e.g. *mttf*, the *mean time to failure*, and that this attribute *mttf* is a floating point number.

We propose that the various objects of a factory, and their attributes, be captured in several different *representations*. Each representation is simply a grouping of objects that have something to do with one another. These representations are the FR, or *factory representation*, the PR, or *process representation*, the DR, or *demand representation*, the MR, or *materials representation*, and the SR, or *scheduling representation*. In addition to the representations, we have two separate *states*, the

FS, or *factory state*, and the MS, or *material state*. Lastly, we have the *scheduler*. The relationship between the five representations, three states, and the scheduler are illustrated in Figure 2-1.

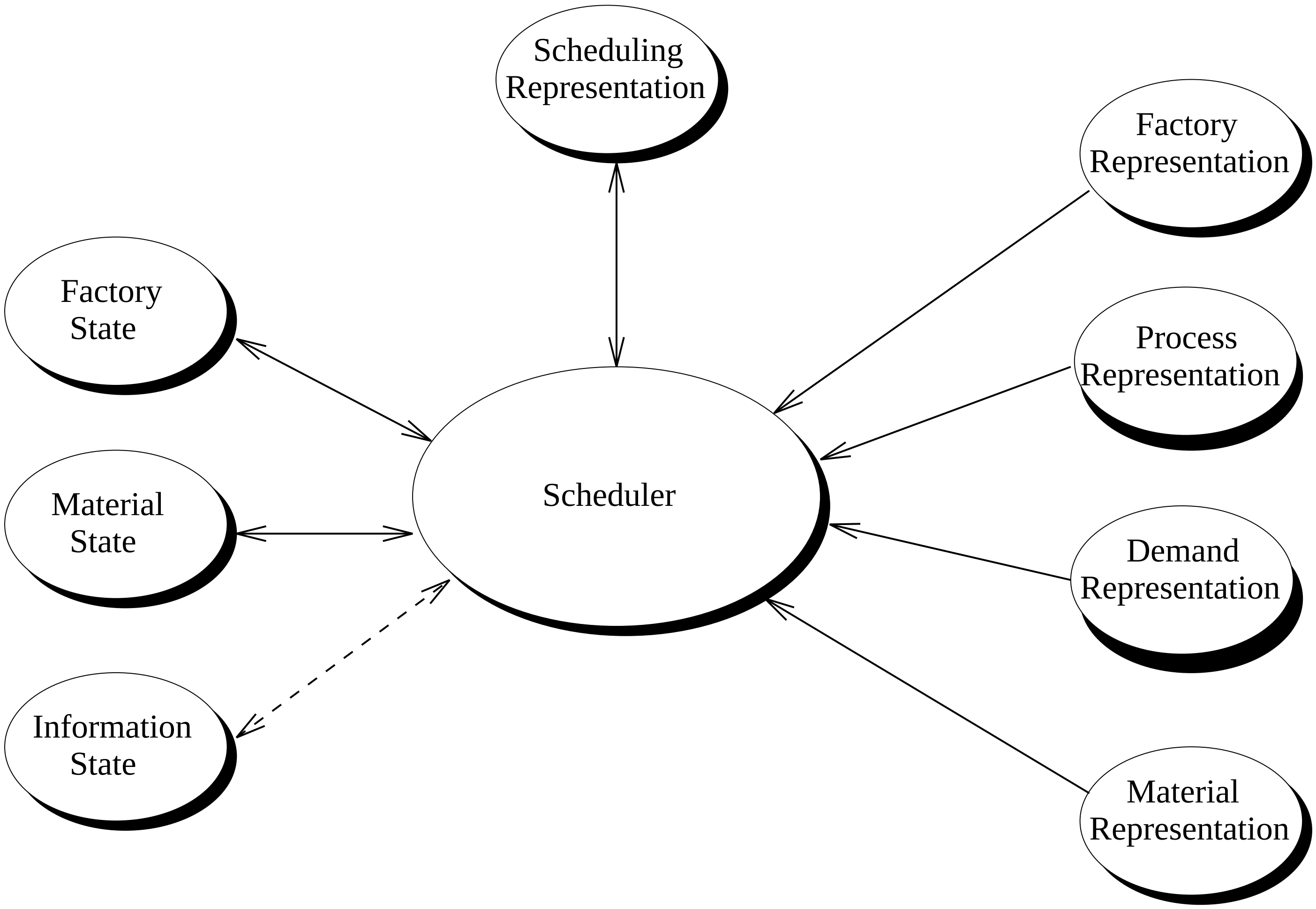

Figure 2-1: The General Model Overview.

The FR, or *factory representation* is where we define the physical objects that reside in the factory. These can include such things as machines, buffers and technicians, as well as such theoretical constructs as manufacturing cells, or groupings of machines.

The PR, or *process representation* is where we describe the process steps need for the creation of a particular part. In the PR, we enumerate all possible paths that a given part may take through the factory to completion.

The DR, or *demand representation* is where we define the nature of demand that arrives to the factory.

The MR, or *material representation* is where we define the items that are to be worked on in the factory, and their attributes. These objects capture the attributes of the parts that will be made in the factory. Such attributes may include such things as a part number, total time in current buffer, and total time in factory. The MR does *not* retain any information about the current position of the part, or its state.

The SR, or *scheduling representation*, is where the scheduling policies are described. These may include such queueing policies as LIFO or FIFO, as well as such inventory control policies as Kanban, CONWIP, or various hybrids.

The FS, or *factory state* is where the information about the current state of the factory is stored. This can include such things as the level of all the buffers, what workers are sick, which machines are down, and what parts are where.

The MS, or *material state* is where the information about the current state of all parts in the system are stored. This can include such things as the current stage of work a part is in, which parts are defective, and how long a certain part has been in the system.

The IS, or *infomation state* is where the information about the current state of the scheduling representation is stored. If a kanban-based scheduling system is being used, for example, the IS is where the current location of the kanbans is stored.

This modularization offers several advantages.

¡list of advantages goes here¿

2.2 The Scheduler

The scheduler is the part of the model that decides which action to carry out when the time for a decision is come. We think of the scheduler as a series of filters which winnow a list of possible actions. Once all "impossible" actions have been removed, a selector chooses the action most desirable according to some value function which the user defines.

The scheduler begins with the state, and determines all possible actions that can be done next. Such actions can be, for example, load a part in a machine, move a part from one place of the factory to another, idle a machine, begin work on a part, or give a worker an order. The list is generated by considering the possible moves that can be made next.

Once the list has been generated, the winnowing begins. The list is passed through a series of filters (see Figure 2-2) until all impossible actions have been weeded out. In Figure 2-2, the oval denotes the state of the system, including the factory state, the materials state, and the state of any tokens that we may be using to do our scheduling. The arrows indicate order of actions, the state being the start of the process. The rectangles represent specific actions carried out by the scheduler, such as updating the state or reducing the list of possible actions. Finally, the diamonds represent the state of the action list.

The first of the filters are the physical constraints. The nature of these constraints are two fold. First, these are the constraints that describe parts of the state space that are forbidden for physical reasons. Perhaps it is dangerous to process two different part types in the same machine at the same time. Perhaps it is simply unwise. In either case, such actions that would be dangerous, foolish or wasteful are removed from the list by the physical constraints.

Second, these are the constraints that express simply physical properties. For example, suppose that the action that we are contemplating is to take a part from an input buffer, load it into a machine and begin the operation on the part. In order for us to load a part into a machine we need several things. We need that the machine be idle (i.e. not currently working on a part), we need that there be some mechanism to physically move the part from the input buffer to the machine (i.e. that a technician is free to do it, or that the robotic arm that moves the part is functional), and we need to ensure that the part can be operated on once it is in the machine. If any one of these prerequisites fails, then basic common sense and physics decrees that we cannot do such an action.

The second filter is the set of scheduling constraints. These constraints disallow possible actions that may be physically possible, and even desirable, but are unable to be performed because of scheduling rules enacted upon the system. For example, the manufacturing system may be controlled by kanbans, where a card, or token, is a prerequisite for an operation to commence. The absence of such a card or token would prevent the action from being acted upon. It is also possible, however, to have scheduling constraints that do not involve tokens at all. For example, we may require that the only part to be worked on is the one most behind its hedging point. It may be that we have a part in an input buffer, a machine able to take that part and space in an output buffer for the worked part, but that the part in question is not the one that our schedule requires us to work on.

Once the scheduling constraints have further reduced the list, we are still left with possible more than one different actions which we may undertake. It is then the purpose of the selector make the decision which action shall be acted upon next, and which actions shall be discarded. Once the selector has made the selection the state is updated, and the process repeats.

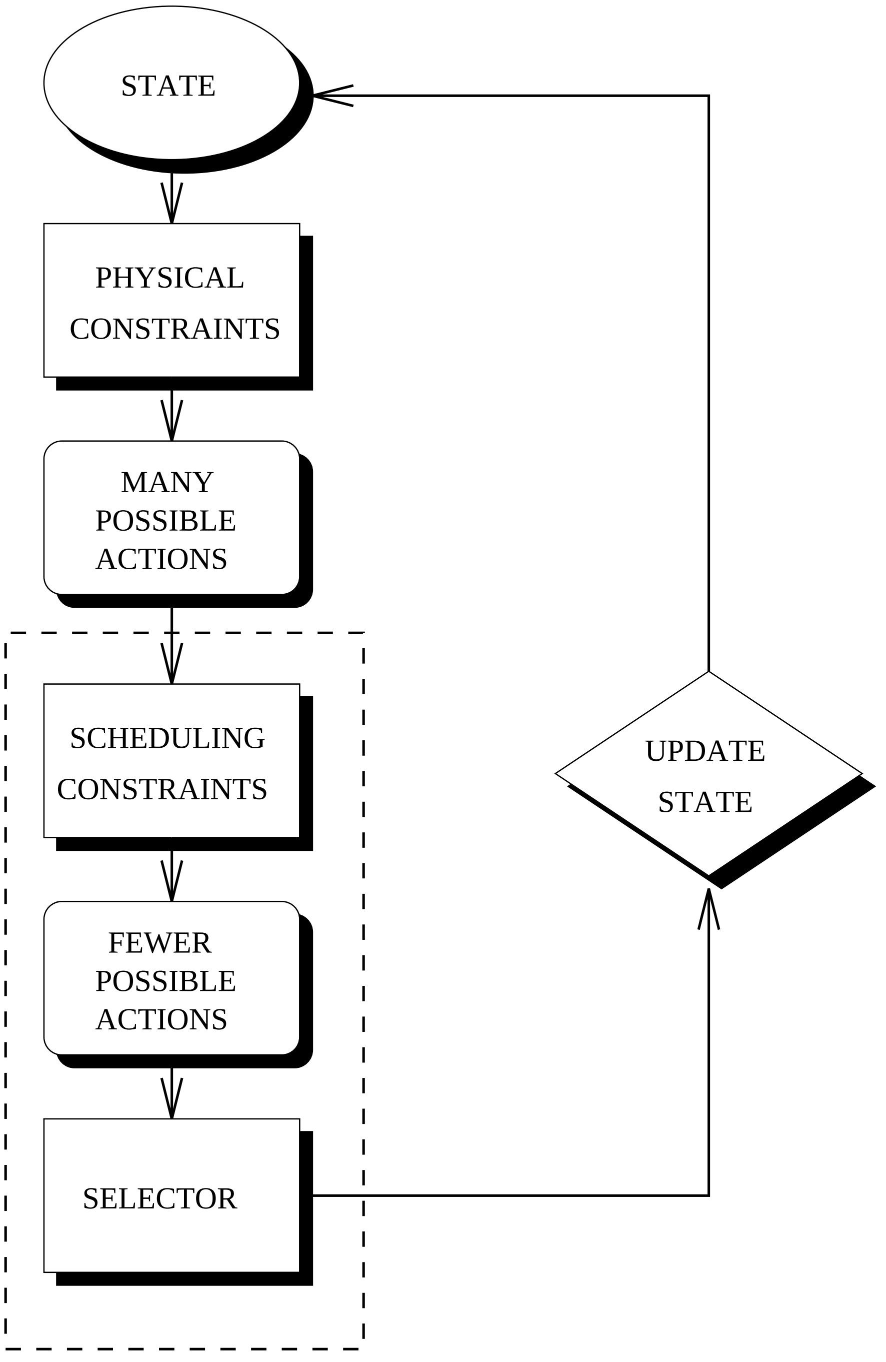

Figure 2-2: The Scheduler.

Recall that

Chapter 3

The Representations

All elements of the factory are described using different collections of objects, or class definitions. These collections are known as class definition lists. There are two different sorts of class definition lists; the physical representations, and the information representation.

3.1 The physical representations

The representations are a class definition list, or collection of class definitions, that describe the characteristics of the physical resources and organization of the factory, as well as the flow of material through the factory and the characteristics of the material itself.

The physical representations are

- the structure of the static physical data.
- the static physical data itself.
- the structure of the dymanic physical data.

The physical representation are *not*

- the dynamic physical data.

- the structure of the information data.
- the information data itself.

The dynamic data is contained in the physical state, and the

There are four different representations. They are the FR, or *factory representation*, the PR, or *process representation*, the DR, or *demand representation*, and the MR, or *materials representation.*

3.1.1 A five-machine transfer line

As an illustration of the ideas presented in this thesis, a running example of a five-machine transfer line will be used. The transfer line is pictured in Fig 3-1. The circles represent machines, the inverted triangles represent buffers, and the solid arrows are the direction of material flow. Thus, a part would enter the line at `Machine_1`, and upon completion of processing will enter `Buffer_1`. `Machine_2` will load a part from `Buffer_1`, and upon completion of processing place the part in `Buffer_2`, which will be loaded by `Machine_3`, and so on. Parts will be processed last at `Machine_5`, and upon completion of processing will be placed in `Buffer_5`. Arriving demand will take parts from the finished goods inventory, that is, the last buffer, at which time the parts will exit the system. If demand arrives for a part of type A, and there are none in the finished goods inventory buffer, then the demand is lost. If parts arrive for a demand of part type B, and there are no parts in the finished goods inventory buffer, that demand may be backlogged.

More specifically, we will consider a transfer line consisting of some flexible machines that will be capable of producing two different part types, denoted part type A and part type B. Part type A will require operation at each station in the transfer line where the operations must be done in increasing order (i.e. operation at station 1 is a pre-requisite for operation at station 2, and so on.), Part type B will require operation at stations 1, 3, and 5, although the order that these operations must be done in will not be rigid. Operation 1 must be performed first, but the part may then go on to station 5 or station 3, after which it must visit the remaining station to complete work.

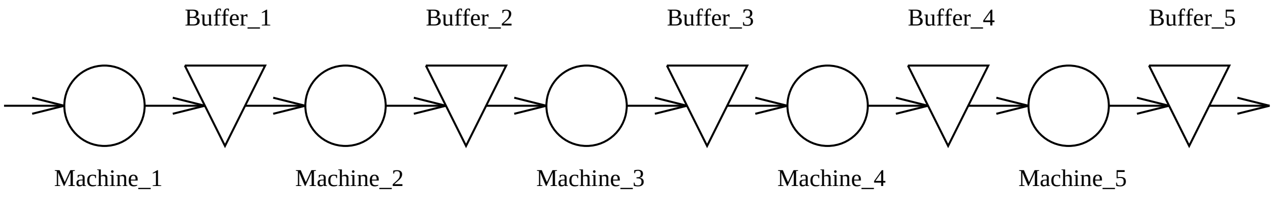

Figure 3-1: A five-machine transfer line.

A description of how this five-machine transfer line is scheduled is the purpose of this thesis. Some obvious questions present themselves. Do a particular machine always load a part from its upstream buffer if it can do so? When does the first machine begin processing a part? How large should the internal buffers be? How does information flow through the system? Before these questions can even be addressed, it is necessary to define the various components of the five-machine transfer line in detail.

3.2 The factory representation

That factory representation, or FR, is the collection of class definitions that describe physical the resources of the factory that do not change on a regular basis. It is in the FR that the existence of machines, buffer, technicians, and manufacturing cells, among others, are declared. The factory representation is object-oriented, and analogous to the Microsystems Factory Representation developed in [5].

In the FR, we describe the basic building blocks of the factory - things that will not change very frequently in the long term. It is here that we declare that there is an object called *machine* or *buffer*, for example, and what that such an object may have characteristics named *mean time to fail* or *mean time to repair*. Characteristics are any attributes that a machine may have. These can include any level of detail that the programmer may wish to include.

It is also in the FR that we construct conceptual groupings of objects in the factory.

For example, it is also here that we declare that there is an object called *manufacturing cell*, and its characteristics. It is also here that we describe the personnel that work in the factory, such as technicians or machine operators.

Specifically, the FR consists of

- the structure of the factory resource static data.
- the factory resource static data
- the structure of the dynamic factory resource data.

The FR does *not* include the dynamic factory resource data. That is contained in the *factory state*.

To illustrate the factory representation, consider the example of the five-machine transfer line. Physically, there are two sorts of objects that make up the line: machines and buffers. As well as the machines and buffers, there are also technicians that work on the line. The existence of these objects must be declared before they can be instantiated. They are declared in the FR.

3.2.1 The factory representation syntax

The symbols used in this section are defined in Table 3.1. The general syntax used is that found in [5].

The factory representation is a list of class definitions that has a syntax that is block structured, similar to C. Each class definition has a keyword, taken to be the name of the object, and a list of attributes that characterize the object. These attributes themselves have a keyword, or name, and several pieces of information.

class-definition-list : /* empty */
| **FR-object-definition class-definition-list**
| **end-of-file**
;

Table 3.1: Definition of syntactical symbols

Symbol	Meaning
:	can have the form
|	or
'...'	terminal
...	nonterminal
/*...*/	comment
CAPS	token
;	end of production

The block of code must end with the keyword 'end'.

end-of-file: 'end';

Each element of the **class-definition-list** is a **FR-object-definition** with the following structure.

FR-object-definition : NAME { **type-definition-list**
| **attribute-definition-list** }
;

3.2.2 Type definitions

We allow for the declaration of new types, either within the object being defined, or during instantiation. These type definitions are simlar to the 'typdef' command in C (see [**?**]).

As a general rule, if a type being defined will not change, it is appropriate to represent it in the object itself. If the type being defined is one that will be changed a great deal, then it should be defined in the textual representation of the object.

The **type-definition-list** is the collection of various type definitions used in the object. The **type-definition-list** has the following form.

type-definition-list : /* empty */
| **type-definition class-definition-list**

;

type-definition : **type** EQUAL **value**

The lexical analyzer returns EQUAL when it reads '='. The use of '=' is optional. It is used primarily when the new type being defined is a specific array of items, for example the failure modes specific to a given class of machine.

3.2.3 Attribute definitions

The attribute definitions are what defines the object. When defining an object, an attribute definition should be made for any relevant piece of information. The ability to capture almost any sort of characteristic about a machine, buffer, or technician by adding new attribute definitions greatly adds to the power of the language.

The **attribute-definition-list** is the collection of **attribute-definition**s for the object in question. Each **attribute-definition** must have the following form.

attribute-definition : NAME COLON **type** COMMA **single** COMMA **static**
COMMA **child** COMMA **default** COMMA
| NUMBER TERMINAL
;

The lexical analyzer will return COLON when it sees a ':', COMMA when it sees a ',', and TERMINAL when it sees a ';'.

The various possible entries for each of the above fields are defined in Table 3.2, and are defined as follows:

type: This field defines the type of the attribute. The type may be an integer, a floating point variable, a boolean variable, an string of characters, or an object. The lexical analyzer will return INTEGERTYPE when it sees `integer`, FLOATTYPE when it sees `float`, BOOLEANTYPE when it sees `boolean`, STRINGTYPE when it sees `string`, FROBJECT when it sees `frobj`, PROBJECT when it sees `probj`, MROBJECT when it sees `mrobj`, and DROBJECT when it reads `drobj`. It is also

Table 3.2: Possible entries for each field of **attribute-definition** of an FR object.

---

type : INTEGERTYPE | FLOATTYPE |
BOOLEANTYPE | STRINGTYPE | NAME ;
single : ONE | MANY;
static : STATIC | DYNAMIC;
parent : PARENT | CHILD;
dfl : STRING | INTEGER | FLOAT | CNULL | /* empty */ | ARRAY;

---

possible to define new types. For example, suppose that the attribute we are defining is the state of a particular machine. In this case, we allow the machine to take on three different possible states, defined by the array `status` that contains the entries `up`, `down`, and `in_maintenance`. We would then denote that the attribute `state` has type `status`, where `status` is the array defined above.

one vs. many: This field defines the number of copies of the attribute. The lexical analyzer returns ONE when it reads `one`, and MANY when it reads `many`. The entry `one` is used when the attribute in question can only take one argument, and `many` if it takes an array of arguments. The default value for this field is `one`.

static vs. dynamic: This field defines whether or not the data for the attribute is dynamic or static. The lexical analyzer returns STATIC if it reads `stat`, and DYNAMIC if it reads `dynm`. If the data is static, then the data of the attribute does not change once the object has been instantiated and the data initialized. If the data is dynamic, then the data of the attribute is stored in the factory state. The default value for this field is `stat`.

parent vs. child: This field defines whether or not the attribute is a parent or a child. The lexical analyzer returns PARENT when it reads `par`, and CHILD when it reads `chld`. If the attribute is a parent, then each replication of that object will also replicate all of the object's children. The default value for this field is NULL.

default: This field defines the default value for the attribute. When the object is instantiated, the attribute will take on this value unless one is specifically assigned to it during initialization. If the attribute is an array, the default data is entered in brackets, with each entry seperated by a comma. That is, default data has the form `[<entry>,<entry>,...,<entry>]`.

3.2.4 An example of an FR representation

We define the objects necessary to represent the five-machine transfer line in the example. To do so, we need four different kinds of class definitions. The first represents the machines in the system, and any corresponding attributes that are necessary and/or sufficient for the description of a machine in the factory in question. The second represents the organisation of the factory into manufacturing cells. The third represents the buffers in the system, and any corresponding attributes of interest. The fourth represents the technicians that work in the factory.

At this moment, we are not actually describing the specific characteristics of the five machines in the transfer line, but only the structure of their data, both static and dynamic. The actual creation of `Machine_1`, `Machine_2`, `Buffer_1`, etc., is done when the textual representation is written. This is described in Chapter 5. Thus, the object `FR_OBJECT_machine` is not actually one of the machines in the transfer line, but rather the class definition that describes the structure of the data of the machines in the transfer line.

```
FR_OBJECT_machine{
string operation-status=[started,completed,held,interrupted,
         restarted,failed];
string machine-status=[working,idle,jammed,down,in-maintenance,set-up,loading,
```

```
                        blocked,starved];

name:                     string;
cell:                     frobj,one,stat,,FR_OBJECT_cell;
technicians:              frobj,many,stat,,FR_OBJECT_technician;
operations:               probj,many,stat,,PR_OBJECT_operation;
input_buffer:             frobj,one,stat,,FR_OBJECT_buffer;
output_buffer:            frobj,one,stat,,FR_OBJECT_buffer;
currentWIP:               mrobj,one,stat,,MR_OBJECT_widget;
mttf:                     float,one,stat,,1.0;
mttr:                     float,one,stat,,1.0;
machine-status:           machine-status,many,dynm,,'idle';
setup:                    setup,one,dynm,,'part-type-A';
operation-status:         operation-status,one,dynm,,'not-started';
number;
};
```

The first class definition made is for an object called `machine`, and each machine in the transfer line will be a copy of this object. That is, each machine in the line will be an instantiation of this object.

The object has the following attributes:

- *name*: this is the name of the object. It has **type** `string`. Since no other fields are assigned values, they are automatically given the values `one`, `stat`, and NULL, respectively.

- *cell*: this attribute has as its type another FR object, named `FR_OBJECT_cell`, which is defined below. The attribute takes a single argument in the instantiation, is static data, and has no default value.

- *technicians*: this attribute has as its type the FR object `FR_OBJECT_technicians`, defined below. The attribute may have multiple entries, and is static. This attribute is meant to be a list of all technicians who are capable of operating the machine that this object represents.

- *operations*: this attribute has the same structure as the attribute `technicians`, and is the list of all possible operations that the machine represented by this object can carrout. Note, however, that the type of the field is not an FR object, but rather a PR object.

- *input-buffer*: this attribute represents the machine's input buffer. As this example is a simple transfer line, each machine only has one input and one output buffer. This need not be the case in general, and simple changing the field `one` to `many` would allow for multiple input buffers to be assigned to this machine during instantiation.

- *output-buffer*: this attribute, identical in structure to the one above, represents the machine's output buffer.

- *currentWIP*: this attribute, of type `mrobj`, points to the part the machine is currently working on.

- *mttf*: this attribute is the mean time to fail of the machine in question. It is a floating point number, static, and has default value 1.0.

- *mttr*: this attribute is the mean time to repair of the machine in question. It is a floating point number, static, and has default value 1.0.

- *machine-status*: this attribute has a previously-defined type called `machine-status`, which is an array of entries of type `string`. In this case, the array in question is defined in the FR object. Recall that this need not be the case; such arrays can also be defined in the instantiation of an object. The attribute may have multiple entries. For example, a machine may be both simultaneously `blocked` and `starved`. This attribute is also *dynamic*, denoted by `dynm`. Upon instantiation of this object, an entry for it is created in the factory state, and all the dynamic data is initialized there.

- *setup*: identical in structure to the attribute above, this attribute takes as its entry the part type that the machine is set up to work on. The default value is part type A.
- *operation status*: identical in structure to the entry above, this attribute takes as its entry the status of the operation on the current part in the machine.
- *number:* this attribute is a special dummy attribute that allows for the creation of multiple identical copies of a specific instantiation of a machine. For example, if this number were two in the textual representation, then two copies of the machine represented would be instantiated into memory at initialization.

```
FR_OBJECT_cell{
name:                    string
supercell:               frobj,one,static,par,'factory';
subcells:                frobj,many,static,chld,FR_OBJECT_cell;
machines:                frobj,many,static,chld,FR_OBJECT_machine;
buffers:                 frobj,many,static,chld,FR_OBJECT_buffer;
technicians:             frobj,many,static,,FR_OBJECT_technician;
};
```

The second object declared is an abstract structural object that allows for hierarchical ordering of groups of machines and buffers. The object does not represent any physical object in the factory, but rather a grouping of objects in the factory. In this case, the object represents a manufacturing cell. Each manufacturing cell belongs to a supercell, and may have subcells.

This example illustrates the use of the default entry. Here, we are setting the default value of the attribute `super-cell` to `'factory'`. The attribute `super-cell` has type `frobj`. Therefore, during instantiation, any replication of this FR object will, by default, be given as its supercell the replication of the FR object named 'factory'. If this is not created, an error will be returned. However, there is no restriction on what sort of FR object the attribute `supercell` may point to. It could be a machine, buffer, or another cell, as long as there is replication that has the name 'factory'. In contrast, the next attribute, `subcells`, also of type `frobj`, takes as its default value the *FR object* with the name `FR_OBJECT_cell`. In this case, during instantiation, the replication of this object's attribute `subcells` *must* take as an argument a replication of the object `FR_OBJECT_cell`.

Each cell also has machines, buffers, and technicians in it. All of the data is static, but this need not necessarily be the case. For example, if management wanted to allow for technicians to be flexible, and move between manufacturing cells, then the attribute `technicians` would be given the value `dynm`, rather than `stat`.

This example also illustrates the importance of the parent-child constructions. In our example, we are only considering one five-machine transfer line. Suppose we wished to investigate the benefits of adding a second line identical to the first. One way to do so would be to create a manufacturing cell with the transfer line contained in it, and then create two identical copies of the cell during instantiation. Thus, during instantiation, the computer would create two copies of the transfer line. That is, it would create two copies of each machine and two copies of each buffer. However, it makes no sense to create two copies of each technician, because technicians are not like machines and buffers in that identical units can be simply purchased. Therefore, we need a way to tell the compiler that certain objects are to be replicated and others are not.

```
FR_OBJECT_buffer{
name:                   string;
in-cell:                frobj,one,static,par,FR_OBJECT_cell;
capacity:               float,many,static,,[+inf,+inf];
holding-cost:           float,many,static,,[0,0];
buffer-level:           float,many,dynm,,[0,0];
number;
};
```

This block of code makes the class definitions necessary to represent a buffer in the transfer line. It also serves to further illustrate the use of arrays in these object definitions. In this case, each buffer will be able to contain two different part types - part type A or part type B. Thus, when defining the capacity, holding cost, and current buffer level attributes necessary to represent this information, we create space for two entries in each attribute.

```
FR_OBJECT_technician{
name:                     string;
qualified-on-machines:    frobj,many,static,, FR_OBJECT_machine;
number;
};
end
```

In this last block of code the attributes for the objects that represent technicians are defined.

As this is the last block of code in the factory representation schema, we must terminate. Hence the final `end`.

3.3 The process representation

The process representation codifies the successive steps necessary for the production of the goods that the factory produces. These process flows can be linear in that they begin with a specific step and end with a specific step, or they can encompass multiple passes through various machines. It is also possible that the process need not follow a particular order. That is, there may be some flexibility in when a certain operation is performed.

The process representation should be as independent of a given instantiation of the factory representation as possible. That is, it should not depend by name on specific machines. This independence allows for greater flexibility in factory design. For example, suppose we have a process that simply requires that operations 1, 2, and 3 must be carried out for a given part to be completed. If we require the process representation to state, by name, the machines on which each operation must be carried out it would have to be updated if the configuration of the factory is changed. If, however, a process representation is simply dependent on the operations involved, then this rewriting of the PR is not necessary should a change in the instantiation of the FR be incorporated. Likewise, should a new part type be added to the system, it is undersirable to have to go back and make large changes in the PR to incorporate this change. The only time the PR will be required to be changed is when there is the introduction of a new kind of operation that can take place.

Therefore, the process representation will simply be a series of class definitions that detail the various operations that can be performed in the factory. These will incorporate whatever information is needed to carry out the operation, i.e. what materials are needed, what specifications define a successful operation, and so on. Thus, the process representation is

- The structure of the operations that will make up a process for the production of a finished good.
- The materials needed for the operation.
- The resulting product for the operation.

The process representation does *not* include

- The recipe to produce a specific part. This is defined in the textual representation.

The reason that the flow of the sequence of steps for the process to produce a given product is not found in the process representation is because this sequence is necessarily dependent on

1. the products produced in the factory.
2. the equipment in the factory.

both of which are dependent on the textual representation of the factory.

3.3.1 The process representation syntax

The process representation syntax is similar to that of the factory representation, but the fields of the various attributes are different.

The symbols used are defined in Table 3.1. The general syntax used is that found in [5].

class-definition-list : /* empty */
| **PR-object-definition class-definition-list**
| **end-of-file**
;

Each element of the **class-definition-list** is a **PR-object-definition** with the following structure.

PR-object-definition : NAME { **type-definition-list**
| **attribute-definition-list** }
;

The **type-definition-list** is the collection of various type definitions used in the object. The **attribute-definition-list** is the collection of **attribute-definition**s for the object in question. Each **attribute-definition** must have the following form.

attribute-definition : NAME COLON **type** COMMA **single** COMMA **assembly** COMMA **invertible** COMMA **default** COMMA
| NUMBER TERMINAL
;

The various possible entries for each of the above fields are defined in Table 3.3, and are defined as follows:

Table 3.3: Possible entries for each field of **attribute-definition** of a PR object.

---

type : INTEGERTYPE | FLOATTYPE |
BOOLEANTYPE | STRINGTYPE | NAME ;
single : ONE | MANY;
static : ASSEMBY | DISASSEMBLY;
parent : INVERTIBLE | NONINVERTIBLE;
dfl : STRING | INTEGER | FLOAT | CNULL | /* empty */ | ARRAY;

---

type: This field defines the type of the attribute. The type may be an integer, a floating point variable, a boolean variable, an string of characters, or an object. The lexical analyzer will return INTEGERTYPE when it sees `integer`, FLOATTYPE when it sees `float`, BOOLEANTYPE when it sees `boolean`, STRINGTYPE when it sees `string` FROBJECT when it sees `frobj`, PROBJECT when it sees `probj`, MROBJECT when it sees `mrobj`, and DROBJECT when it reads `drobj`. As in the FR, it is possible to define new types either in the definition of the object, or during istantiation of the object.

one vs. many: This field defines the number of copies of the attribute. The lexical analyzer returns ONE when it reads `one`, and MANY when it reads `many`. The entry `one` is used when the attribute in question can only take one argument, and `many` if it takes an array of arguments. The default value for this field is `one`.

assembly vs. disassembly : this field denotes the nature of the operation that will be performed on the attrinute. The lexical analyzer returns ASSEMBLY if it reads `ass`,DISASSEMBLY if it reads `dis`, and NULL if it reads anything else. If the operation is an assembly operation, the necessary inputs will be assembled to create the output. The default is NULL.

invertible vs. noninvertible : if an operation is invertible, then it may be undone, returning the raw material to their original condition. For example, the operation of assembling two pieces of metal with a nut and a bolt may be inverted, while an ultrasonic weld of two pieces of plastic cannot. The lexical analyzer returns INVERTIBLE if it reads `inv` and NONINVERTIBLE if it reads `noninv`.

default : this indicates the default value of the type of the attribute, should none be determined at instantiation.

If, when defining the various fields of the attribute, only the type is entered, followed by a ';', then the remaining fields will automatically be given the following attributes: `one`,CNULL,`noninv`,CNULL.

There are also some attribute names that are reserved words that the compiler will look for. These constitute the canonical attributes that would make up any representation of an operation, such as what is needed to begin the operation, and what is the result of the operation at the end.

INPUT : this attribute captures the input(s) necessary to do the operation in question. The advantage to using this reserved attribute is that the physical predicates will automatically look for the items in the input buffers of the machine in question.

OUTPUT : this attribute captures the output(s) that result from the operation being performed on the inputs.

3.3.2 An example of a PR representation

```
PR_OBJECT_operation{
name:              string;
INPUTS:            mrobj,one,,noninv,MR_OBJECT_widget;
OUTPUTS            mrobj,one,,noninv,MR_OBJECT_widget;
technician:        frobj,many,,,FR_OBJECT_technician;
<restrictions,
```

```
etc>
};
end
```

This block of code is the class definition that captures the necessary requrements for an operation to be performed at any station in the five-machine transfer line. In the case of our example, each station performs the same sort of operation. It is neither an assembly nor a disassembly, but a straight-forward operation performed on a single part. For example, it could be drilling a hole or etching a wafer, but not bolting two parts together. The following attributes are defined in this block of code:

- *name*: this is the name of the object (in this case, the operation being defined). As in the FR, it is a `string` type.
- *INPUTS*: this attribute captures the inputs necessary for the perforance of the operation in question. It has as its type and `MR_OBJECT` called `widget`, which will be defined below. It is a singluar attribute, neither and assembly nor disassembly is performed on this object, and the operation performed is irreversible.
- *OUTPUTS*: this attribute captures the outputs produced when the operation represented by this object is implemented. The item produced is of type `MR_OBJECT_widget`, which will be defined below.
- *technician*: this attribute capture the technicians that are capable of carrying out this operation. Unlike the attributes `INPUTS` and `OUTPUTS`, the necessity of having one of the listed technicians to perform the operation must be specifically stated in the physical predicates.

As this is the only block of code in the process represention schema for the example, we must terminate after this block. Hence the `end`.

3.4 The materials representation

The materials representation, or MR, is a collection of class definitions that capture the nature of the materials that exist in the system. It is here that the structure and attributes of the various parts are described. Any information that is indigenous to a part or piece in the factory is recorded here.

The question presents itself: why have a seperate representation for pieces in the system? Why not simply represent any piece in the system by an object in the factory representation? The answer is that while the information contained in each MR object is static, the replications of those objects are not. That is, individual parts can be created and destroyed in the system. They can be scrapped, enter, or leave the system. Each part in the system at a given time is stored entirely in the factory state.

3.4.1 The materials representation syntax

The symbols used in the syntactical definitions of the materials representation are defined in Table 3.1. The general syntax used is that found in [5].

class-definition-list : /* empty */
| **MR-object-definition class-definition-list**
| **end-of-file**
;

Each element of the **class-definition-list** is a **MR-object-definition** with the following structure.

MR-object-definition : NAME { **type-definition-list**
| **attribute-definition-list** }
;

The **type-definition-list** is the collection of various type definitions used in the object. The **attribute-definition-list** is the collection of **attribute-definition**s for

Table 3.4: Possible entries for each field of **attribute-definition** of an MR object.

---

type : INTEGERTYPE | FLOATTYPE |
BOOLEANTYPE | STRINGTYPE | NAME ;
single : ONE | MANY;
static : INVERTIBLE | UNINVERTIBLE;
dfl : STRING | INTEGER | FLOAT | CNULL | /* empty */ | ARRAY;

---

the object in question. Each **attribute-definition** must have the following form.

attribute-definition : NAME COLON **type** COMMA **single** COMMA **invertible**
COMMA **default** COMMA
| NUMBER TERMINAL
;

The various possible entries for each of the above fields are defined in Table 3.4, and are defined as follows:

type: This field defines the type of the attribute. The type may be an integer, a floating point variable, a boolean variable, an string of characters, or an object. The lexical analyzer will return INTEGERTYPE when it sees `integer`, FLOATTYPE when it sees `float`, BOOLEANTYPE when it sees `boolean`, STRINGTYPE when it sees `string` FROBJECT when it sees `frobj`, PROBJECT when it sees `probj`, MROBJECT when it sees `mrobj`, and DROBJECT when it reads `drobj`. As before, it is also possible to define new types.

one vs. many: This field defines the number of copies of the attribute. The lexical analyzer returns ONE when it reads `one`, and MANY when it reads `many`. The entry `one` is used when the attribute in question can only take one argument, and `many` if it takes an array of arguments. The default value for this field is `one`.

invertible vs. uninvertible : *this field is different from the notion of invertibility found in the PR!* It does not record whether or not the part can be reverted to a previous state. If an attribute in the MR is invertible, it means that a search can be done on all attributes of this sort that have a particular value. Thus, in our example, a search could be done for all `product-codes` that have the value 135, giving us the ability to locate where, and in what stage of completion, all products with code 135 are in the factory.

default: This field defines the default value for the attribute. When the object is instantiated, the attribute will take on this value unless one is specifically assigned to it during initialization. If the attribute is an array, the default data is entered in brackets, with each entry seperated by a comma. That is, default data has the form `[<entry>,<entry>,...,<entry>]`.

If, when defining the various fields of the attribute, only the type is entered, followed by a ';', then the remaining fields will automatically be given the following attributes: `one`,`noninv`,CNULL.

3.4.2 An example of an MR representation

Although we are producing two different part types on the five-machine transfer line, their class definitions are identical. There are five distinct stages of production for part type A, and three for part type B. Therefore, we need a total of 8 different kinds of parts to be created, representing the various stages of completion of each part type. Therefore, during instantiation, eight different copies of the above object will be created in the textual representation of the factory, one for each stage of completion of the product, where this is done for each product. This is analogous to what we do for the transfer line; although there are five machines and five buffers in the transfer line, we only make one class definition for all the machines, and one class definition for all the buffers. The distinctness between them is created at instantiation.

When a part first enters the system, it will be represented by an replication of `MR_OBJECT_widget` that represents any part of type A that is set to receive service at station 1. When the part in question finished service, the copy of the part that previously existed will be deleted, and a new copy representing the part in question will be created with the apropriate updated data. In this manner, we have an accurate record of all parts in the system. We note that things need not be done in this manner, but we choose to do so to emphasize the fact that copies of this object reside in the factory state, not in the schema. The instantiations of parts in the system will be the following:

`part-type-A-station-1` is the instantiation of `MR_OBJECT_widget` represents a part for product type A at the first work station. When an order for product type A comes in, and the demand is serviced, this is the initial part that is created. `part-type-A-station-2` This instantiation represents a part for product type A at the second stage of completion. `part-type-A-station-3`,`part-type-A-station-4`, and `part-type-A-station-5` represent the product in the third, fourth, and fifth stages of completion, respectively. In the case of product A, the stages of completion are the same as the workstations in the transfer line. For product B, there are two possible routes to completion: $1 \to 3 \to 5$, or $1 \to 5 \to 3$. `part-type-B-stage-1` represents a type B product in its initial stage of work, i.e. when it first enters the system. `part-type-B-stage-2` represents a type B product in its second stage of work (i.e. at *either* station 3 *or* station 5), and `part-type-B-stage-3` represents a type B product in its final stage of completion (station 5 is the product in question did its second stage at station 3, and station 3 otherwise).

```
MR_OBJECT_widget{
name:              string;
product-type:      int;
subassemblies:     mrobj,many,MR_OBJECT_widget;
nextpart:          mrobj,one,MR_OBJECT_widget;
inbuffer:          frobj,one,FR_OBJECT_buffer;
};
end
```

- *name*: this is the name of the part.
- *product-type*: this field records the information of which product this part is to be a part of. A '0' is entered for product type A, and a '1' is entered for product type B.
- *subassemblies*: this attribute records any sub- assemblies that make up the part that this object represents. In the case of the five-machine transfer line, this attribute will simply list the previous steps performed on the part in question. For example, if this object is representing a part of type A that is at station 4, then we know that it has already had work done at stations 1, 2, and 3. It is here that the previous states of the current part are recorded.
- *nextpart*: this attribute is a pointer to the next stage in the products production.
- *timestamp*: this attribute is of type `float`. It is simply the recording of the time that the part in question was created. This is, in our example, simply the time that the part represented by this object entered the system, if the part is at station 1, or the time that it completed service at the most recent station.
- *product number*: this attribute records the number of the product that will eventually be produced when this part completes service at all of the stations it must visit.

3.5 The demand representation

The demand representation captures the structure and properties of arriving demand to the system. Each individual demand is a copy of an object defined in the DR. As with the PR, copies of objects in the DR reside in the factory state. They are created and destroyed as needed.

3.5.1 The demand representation syntax

The symbols used in the syntactical definitions of the demand representation are defined in Table 3.1. The general syntax used is that found in [5].

class-definition-list : /* empty */
| `DR_OBJECT`-**definition class-definition-list**
| **end-of-file**
;

Each element of the **class-definition-list** is a `DR_OBJECT`-**definition** with the following structure.

`DR_OJECT` : NAME { **type-definition-list**
| **attribute-definition-list** }
;

The **type-definition-list** is the collection of various type definitions used in the object. The **attribute-definition-list** is the collection of **attribute-definition**s for the object in question. Each **attribute-definition** must have the following form.

attribute-definition : NAME COLON **type** COMMA **single** COMMA **backlog**
COMMA **duedate** COMMA **default** COMMA
| NUMBER TERMINAL
;

The various possible entries for each of the above fields are defined in Table 3.5, and are defined as follows:

type: This field defines the type of the attribute. The type may be an integer, a floating point variable, a boolean variable, an string of characters, or an object. The lexical analyzer will return INTEGERTYPE when it sees `integer`, FLOATTYPE

Table 3.5: Possible entries for each field of **attribute-definition** of a DR object.

---

type : INTEGERTYPE | FLOATTYPE |
BOOLEANTYPE | STRINGTYPE | NAME ;
single : ONE | MANY;
static : BACKLOG | NOBACKLOG;
static : DUEDATE | NODUEDATE; **dfl** : STRING | INTEGER | FLOAT | CNULL | /* empty */ | ARRAY;

---

when it sees `float`, BOOLEANTYPE when it sees `boolean`, STRINGTYPE when it sees `string` FROBJECT when it sees `frobj`, PROBJECT when it sees `probj`, MROBJECT when it sees `mrobj`, and DROBJECT when it reads `drobj`. As before, it is also possible to define new types.

one vs. many: This field defines the number of copies of the attribute. The lexical analyzer returns ONE when it reads `one`, and MANY when it reads `many`. The entry `one` is used when the attribute in question can only take one argument, and `many` if it takes an array of arguments. The default value for this field is `one`.

backlog vs. nobacklog This field defines whether or not the object ordered can be backlogged. The lexical analyzer returns BACKLOG when it reads `bklg`, and NOBACKLOG when it reads `nobklg`. The default value is `nobklg`.

default: This field defines the default value for the attribute. When the object is instantiated, the attribute will take on this value unless one is specifically assigned to it during initialization. If the attribute is an array, the default data is entered in brackets, with each entry seperated by a comma. That is, default data has the form `[<entry>,<entry>,...,<entry>]`.

If, when defining the various fields of the attribute, only the type is entered, followed by a ';', then the remaining fields will automatically be given the following attributes: `one`,`noninv`,CNULL.

3.5.2 An example of a DR representation

We define two seperate demand objects, one to represent product orders that may be backlogged, and one to represent item orders that may not be backlogged.

Recall that orders for part type A may not be backlogged, while orders for a part of type B may be backlogged.

```
DR_OBJECT_nobacklog_demand{
itemdemanded      MR_OBJECT_widget,one,nobklg,;
};
```

This block of code defines the object that represents orders for a part of type A.

```
DR_OBJECT_backlog_demand{
itemdemanded      MR_OBJECT_widget,one,bklg,;
};
end
```

This block of code defines the object that represents orders for a part of type B.

As this is the last block of code in the demand representation schema, we terminate with `end`.

Chapter 4

The Action Objects

The action objects are the class definitions that desribe the way that the physical states, both factory and material, change. Such changes occur when, for example, a part is moved from one place to another, a machine goes down, or a technician goes off shift.

The actions objects differ from the representations described in Chapter 3 in two ways. First, an instantiation of an action object is a function of elements in the physical state. For example, if the action being represented is that of loading a part from a buffer to a machine, and beginning operation on the part, then the representing action object will be a function of the objects that represent the machine, buffer, and part. Second, when an action object is instantiated, it's replication lies entirely in the physical state.

Why are the actions objects not part of the physical representation. The class definitions in the four physical representations are almost entirely independent of one another. While an object in the FR can take as an attribute an object in the MR, for example, the only consistency requirement is that the object in the MR exist. The actual structure of the MR object is irrelevant as far as the FR object is concerned. For example, an FR object `machine` could have an attribute that restricts it from loading a certain MR object called `wing-nut`, but the attributes of `wing-nut` can be changed freely without affecting the FR class definition.

The action objects are closely linked to the objects in the process representation.

In a sense, the form two sides of the same coin. The attributes of the objects in the process representation represent, in detail, the physical prerequisites for a given operation. However, the do not represent how the physical state will change is the operation represented in enacted. This catagorization of the physical state change falls to the action objects. Therefore, it is required that *for each object in the process representation, there must be a corresponding action object.*

While the actions objects have close ties to the objects in the process representation, they are by no means limited to them. Action objects are meant to capture any sort of change in the physical state.

4.0.3 The syntax of the action objects

The syntax of the action objects is block structured, like C. The symbols used are defined in Table 3.1. The descriptive syntax used is that found in [5].

The collection of action objects forms a **class-definition-list** that has the following structure.

class-definition-list : /* empty */
| **action-object-definition bf class-definition-list**
| **end-of-file**
;

The list must end with the keyword 'end'.

end-of-file: 'end';

Each element of the **class-definition-list** is a `ACTION_OBJECT`**-definition** with the following structure.

action-object : NAME **argument-definition-list** {
type-definition-list type-definition
| **attribute-assignment-list attribute-assignment**
| **function-definition-list function-definition** }

;

The **argument-definition-list** is the collection of definitions that are the arguments of the action object. These arguments represent the objects in the factory state with dynamic attributes that will take on new values upon enactment of the action that the action object represents. The **type-definition-list** is the collection of locally defined types, analogous to the types defined in the representations in chapter 3. The **function-definition-list** is the collection of functions.

The form of the argument definition list is as follows.

argument-definition-list : (/* empty */
| **argument-definition-list argument-definition**)
argument-definition : NAME COLON **type** ;

Thus, the **argument-definition-list** begins with an open paranthesis, has a collection of **argument-definitions** or comments, and ends with a closed parenthesis.

Here, **type** may be one of INTEGERTYPE, FLOATTYPE, BOOLEANTYPE, STRINGTYPE, or NAME.

The form of the **type-definition-list** is identical to that for the factory representation, described in section 3.2.1.

The **attribute-assignment-list** is a collection of **attribute-assignments**.

attribute-assignment-list : /* empty */
| **attribute-assignment attribute-assignment-list**)
;
attribute-assignment : **type operation value** ;

Here, **type** may be one of INTEGERTYPE, FLOATTYPE, BOOLEANTYPE, STRINGTYPE, or NAME.

Remember: action objects are class definitions that describe how the *physical state* changes when a specific action is taken. Therefore, the only attributes that may be changed by an bf attribute-assignment are *dynamic* attributes.

Postfix expressions group left to right. That is, the proper form making the attribute assignment for an attribute of an object is

postfix-expressions ->identifier

For **operation**, the lexical analyzer will return ASSIGN when it reads ':=', DECREMENT when it reads '-=', INCREMENT when it reads '+=', MULTIPLY when it reads '*=', DIVIDE when it reads '/=', APPEND when it reads '.=', and TERMINAL when it reads ';'. The meanings of these are summarized in Table 4.1. If an operation is enacted on a **type**, there are restrictions on the **value** that may be assigned. These are also detailed in Table 4.1. A syntax error will be returned if, say, an `FR_OBJECT_machine` is assigned the **value** '6', as this is assigning an object an integer value.

Table 4.1: Attribute value operations

Operation	Meaning	**value** Restrictions
`:=`	attribute is assigned value **value**	none
`-=`	value of attribute is decremented by **value**	float, integer
`+=`	value of attribute is incremented by **value**	float, integer
`*=`	value of attribute is multiplied by **value**	float, integer
`/=`	value of attribute is divided by **value**	float, integer, non-zero
`.=`	value of attribute is appended by **value**	none

The **function-definition-list** is a collection of **function-definition**s. Each **function-definition** has the following form.

function-definition : **function** (**expression-list** {
`ACTION_OBJECT`**-definition** `ACTION_OBJECT`**-definition-list**
| **attribute-assignment attribute-assignment-list**
| **function-definition-list function-definition** }
;

A function evaluates the elements of the **expression-list** and, depending on the function in question, executes the items found within the brackets. These items can be other action objects, other functions, or attribute assignments. Each function definition must end with a ';'.

function may be one of the following.

if : This function evaluates the **expression-list** and, if it evaluates to TRUE, exectutes the items in the brackets exactly once.

while : This function evaluates the **expression-list**, and if it evaluates to TRUE, executes the items in the brackets until the **expression-list** evaluates to FALSE.

foreach: This function executes the items in the brackets exactly once for each expression in the **expression-list**.

create : This function will instantiate any dynamic object, such as PR object, and place it in memory with the attribute definitions made in the braces.

destroy : This function will destroy the replication of the dynamic object passed to it.

Like the **argument-list** of the action object definition, the **expression-list** of the **function-definition** is made up of **expression-list**s or **expression**s. It must end with a right parenthesis. The expression list is of the form below.

expression-list /* empty */
| **expression-list expression**
| &
| ˆ
)

Each **expression** or **expression-list** is connected with the others by the logical connectives `&` and `^`. These take the meanings 'and' and 'or', respectively. As is standard in any zeroth-order logical system, `a ^ b` (read 'a or b') given the value TRUE if either `a` or `b` takes the value TRUE, and FALSE if both `a` and `b` take the value FALSE. (See [4] or [3] for details.)

Each **expression** is a boolean statement that is to be evaluated as either TRUE or FALSE by the program. Each has the following structure, and must end with a semi-colon, which the lexical analyzer regards as TERMINAL.

expression : **type operator type** ;

Here, **operator** stands for a relational operator. Thus, we are comparing the element on the left with the element on the type. For this reason, the **type**s *must* be the same. If not, an syntax error is returned. This relational operator groups left to right, and compares the values of the two different **types**. If the relation holds, the value returned is TRUE, otherwise FALSE is returned. The allowable operators are defined in Table 4.2.

note: in the case of the function being **foreach**, *no* operator is entered, and only one **type** entry is entered.

Table 4.2:

Operator	Lexical Analyzer Return	Meaning	Restrictions
`>`	GREATERTHAN	greater than	any numerical
`<`	LESSTHAN	less than	any numerical
`=`	EQUAL	equal	any numerical
`>`	GREATERTHAN	greater than	any numerical
`>=`	GREATERTHANEQUAL	greater than or equal	any numerical
`<=`	LESSTHANEQUAL	less than or equal	any numerical
`!<`	RIGHTELEMENT	right is element of left	any nonnumerical
`>!`	LEFTTELEMENT	left is element of right	any nonnumerical

4.0.4 An example of the action objects

In Chapter 3, we made the necessary class definitions to describe the physical components of the five-machine transfer line (the objects created in the factory representation), the parts that move through the transfer line (the objects created in the materials representation), the structure of demand to the tansfer line (the objects created in demand representation), and the various operations that can be performed by the stations in the transfer line (the objects in the process representation).

In this section, we make the necessary class definitions to describe the actions that take place on (or in) the transfer line. In order to keep the example to a reasonable size, and still demonstrate the power of the language, we will only describe three actions. These will be load and operate on a part, unload a completed part into a buffer, and satisfy a demand to the finished goods inventory buffer.

Recall that action objects may only

```
ACTION_OBJECT_load_part(FR_OBJECT_machine,
                        PR_OBJECT_operation,
                        MR_OBJECT_widget)
{
FR_OBJECT_machine->status='working';
FR_OBJECT_machine->output-buffer->
   buffer-level[MR_OBJECT_widget->product-type] -= 1;
FR_OBJECT_machine->currentWIP = MR_OBJECT_widget;
};
```

This piece of code encodes the changes in the state that will be made if a part is loaded into a machine from an input buffer and work is begun on it. If this action is taken, the physical state will change in a number of ways. The buffer that the part to be worked on is in will lose that part, and its level will be decremented by one. The machine that the part is loaded into will no longer be idle, but will rather be working on the part.

The first line states that this is an action object called `load_part`, and that this action object has as arguments the FR object `FR_OBJECT_machine`, the PR object `PR_OBJECT_operation`, and the FR object `MR_OBJECT_widget`. Recall that when this action object is instantiated into a possible action, the arguments will not be general pointers to objects, but rather specific replications of `FR_OBJECT_machine` that is to do the loading, `PR_OBJECT_operation` that is the operation that will be done, and `MR_OBJECT_widget`, which is the part to be loaded. However, it is the *replications* which are the arguments for this action object, not the names of the objects themselves. This will be clarified in Chapter 5.

We make no type definitions in this example. The first line of code after the open braces is an attribute assignment that sets the value of the attribute `machine-status` of the replication of the object `FR_OBJECT_machine` to the string 'working'. This represents the change in state of the machine from whatever it was before, in this case, one element of the type definition `operation-status`, made in the class definition of the object `FR_OBJECT_machine`.

The second line of code decrements the level of the machine's input buffer. Recall that this was defined as an attribute in the class defenition of the FR object `machine`.

The third line of code makes the attribute assignment that places the part to be worked on in the machine.

```
ACTION_OBJECT_unload_part(FR_OBJECT_machine)
{
MR_OBJECT_widget   oldpart = FR_OBJECT_machine->currentWIP;
MR_OBJECT_widget   newpart;

FR_OBJECT_machine->output-buffer->
   buffer-level[part->product-type] += batchsize;
FR_OBJECT_machine->status = 'idle';
FR_OBJECT_machine->currentWIP = NULL;
create(newpart);
newpart = oldpart->nextpart;
```

```
newpart->subassemblies .= oldpart->subassemblies;
newpart->inbuffer = FR_OBJECT_machine->outputbuffer;
destroy(oldpart);
}
```

This object catpures the changes made in the physical state that occur when a machine unloads a part into its output buffer.

The first line of code is a type definition that is local to this action object. We declare that there is something called `oldpart` that has type `mrobj`, and must be an instance of `MR_OBJECT_widget`. It's value is set to be `FR_OBJECT_machine->currentWIP`. The second line of code defines `newpart` in exactly the same manner as `oldpart` was defined, but its value is not set. The next line of code increments the level of the output buffer of the machine that is unloading the part. The fourth line of code sets the status of the machine to 'idle', and the fifth sets the internal part to NULL, removing the part just worked on from the machine's bay.

The state of the machine has been entirely changed. It is now idle and has no part in its work bay. The level of the machine's output buffer has been increased to reflect the fact that it now has one more part in it. The last thing needed to do is actually place the new part in the buffer.

Recall that we have chosen to work with eight different instantiated part types, one for each stage of completion of each product being produced. Again, this is not a necessary construction, but rather was chosen to illustrate some of the features of the language. Because of this construction, we must destroy the replication of the old part, and create a new replication. The remaining lines of code do this. `create(newpart)` creates a copy of the new part in memory, `newpart=oldpart->nextpart` sets this new part in memory to be the next stage of production of the old part, `newpart->subassemblies=oldpart->subassemblies` sets the information of all previous stages of production to the new part created so that it inherits the history of production to date, and `newpart->subassemblies.=newpart` appends the information to reflect the current stage of production that has just occurred. The next line of code places the new part in the buffer, and the last line of code destroys the replication of the old part.

```
ACTION_OBJECT_fill_demand(DR_OBJECT_demand)
{
MR_OBJECT_widget part;
part = DR_OBJECT_demand->itemdemanded;
Buffer_5->buffer-level[part->product-type]-=1;
destroy(part);
destroy(DR_OBJECT_demand);
}
```

The last action object fills demand. This can only happen at the last buffer, so there is no need to tell the action object where the action is being taken. All that is passed to the object is the type of demand that has arrived. The demand is filled, the replication of the demand object that represented the arriving demand is deleted.

The first line of code makes a type definition. The next line of code assigns this new type to be the item demanded by demand. This third line of code of this action object decrements the level of the finished goods inventory. Notice that this declaration is distinct from the ones made in previous action object definitions in that it specifically refers to a *replication* of an object, and not to the class definition of the object itself. The ability to refer to instantiations of objects as well as objects is what sets the action object class definitions apart from the physical representations. The remaining lines of code delete the replications of the part and the demand, respectively.

Chapter 5

The textual representations - instantiation of the physical schema

The textual representation is where the factory is instantiated. It is here that the abstract objects are given concrete form. All the necessary replications needed to describe the factory are created, and their attributes are assigned the correct data.

5.1 Textual representation syntax

The grammar for the textual representation is similar to that for the action objects. The symbols used are defined in Table 3.1. The general syntax used is that found in [5].

The textual representation for a given factory is a collection of the instantiations of the various objects defined in each of the physical schema. The textual representation has the following form.

schema-instantiation : /* empty */
| **object-instantiation-block object-instantiation**
| **end-of-file**
;

The textual representation must end with the keyword `end`. Hence **end-of-file** = `end`;

Each group of objects of the same class are specified in an **object-instantiation-block**, which has the following structure.

object-instantiation-block : /* empty */
| **object-instantiation-block object-instantiation**
| **function-list**
;

The function list

Each element of the **object-instantiation-block** is an **object-list** with the following structure.

object-list : **object** {
| **object** COMMA **object-list**
;

The structure of the **object** is as follows.

object : **object-name** = NAME {
type-list
| **function-list**
| **attribute-list** }
specific-name-list ;

That is, each object is instantiated stating the name of the object being instantiated, and setting it equal to the name of the replication. After the name of the specific replication follow a left braces { , the assignment of the attributes, and then a right braces } . The specific name list follows.

The type list makes the type definitions for replications of the object. They type list has the following structure.

type-list : **type** {
    | **type** COMMA **type-list**
    ;
type : **type** ASSIGNMENT **value**;

In this case, the lexical analyzer returns EQUAL when it reads a '=', and APPEND when it reads a '.='. If the assignment operator EQUAL is used, then any type definition made in the class definition of the same name as the type definition made in the textual representation will be replaced by the type definition made in the textual representation. If the assignment APPEND is used, any entries in the type definition (if any) in the textual representation will be appended to the entries of the a type definition of the same name made in the class definition. If the the two type definitions are of different type (i.e. if an array of integers is appended with a string of characters) and error will result.

The attribute list has the following structure.

attribute-list : /* empty */
    | **attribute attribute-list**)
    ;
attribute : **type** EQUAL **value-list** TERMINAL;

Here, **type** may be one of INTEGERTYPE, FLOATTYPE, BOOLEANTYPE, STRINGTYPE, or NAME. The lexical analyzer returns EQUAL when it reads a '='.

Assignment of multiple values to a single attribute is possible to attributes with the characteristic `many`. In this case, the multiple values assigned to the attribute are seperated by commas, with a semi-colon at the end of the list.

value-list : **value**

| **value** COMMA **value-list**)

;

attribute : **type** EQUAL **value-list** TERMINAL;

The specific name list is a short cut that allows for the creating of multiple replications of objects. It is a particularly pleasant construct in [5], and we incorporate it here. If the special attribute `number` is used in making the class definition of an object, then for each name entered in the specific name list, a replication of the object is created, with that name placed in the NAME slot of the **object** structure defined above. The NAME slot can either be empty, or have entries already in it. If it does have entries in it, then any replications of those names in the specified name list will be ignored. If a number, say 6, is specified for the field `number`, then only the first six names in the specific name list will be used. If there are fewer names in the specific name list, then the last machine named will be the name of each additional machine, indexed by a number. For example, if there are only four names in the specific name list, and the attribute `number` is specified to be six, and the fourth name in the specific name list is `drillpress`, the remaining two replications will have the names `drillpress-1` and `drillpress-2`, respectively. If there are more names in the specific name list than 6, only the first 6 will be used, and only six replications will be instantiated. In this case, the entries in NAME will *not* be used, except as a way of referencing this group of machines.

The form of the specific name list is as follows.

specific-name-list : **specific-name**

| **specific-name** COMMA **specific-name-list**

;

specific-name : **object**;

Observe that each entry in the spcific name list has the form **object**. In this way, attributes of each replication can be changed. Any attributes not assigned are identical to the object that the specific name is a part of.

5.2 An example of the textual representation

Based on the schema defined in the examples in Chapter 3, the actual transfer line is described. That is, copies of the objects defined above are created in memory to represent the actual objects in the factory.

5.2.1 Instantiation of the FR schema

First, the objects from the Factory Representation are instantiated.

```
FR_OBJECT_cell = Factory{
name = 'factory';
machines = Machine_1, Middle_Machines, Machine_5;
buffers = Buffers;
technicians = Apprentices, Machinists;
}
```

The first bloack of object instantiation code creates the factory itself. That is, it creates an instantiation of the class definition `FR_OBJECT_cell`, gives it the name 'factory', and assigns values to the attributes. Specifically, it states that the attribute `name` takes the value `'factory'`, the attribute `machines` takes the values `Machine_1`, `Middle_Machines`, and `Machine_5`, the attribute `buffers` takes the value `Buffers`, and the attribute `technicians` takes the values `Apprentica` and `Machinist`. All of these values that are instantiations of class definitions (e.g. `Apprentice`, which is an instantiation of the class definition `FR_OBJECT_technician`) *must* be defined, or an error will result.

```
FR_OBJECT_machine = Machine_1{
cell = Factory;
technicians = Machinists;
output_buffer = buffer_1;
mttf = 90;
mttr = 6;
```

```
number = 2;
} Machine_5{output_buffer = Buffer_5};
```

This block of object instantiation code uses the specific name option. It instantiates two identical replications of the class definition `FR_OBJECT_machine`, naming them `Machine_1` and `Machine_5`, respectively. Notice, however, that while the replications are given these names, there is no entry in the attribute `name` that was defined in the class definition. The value for the entry `number` is `2`. Has it been `3`, a thrid replication would have been created. Since no name would have been specified for it, it would have automatically been given the name `Machine_5-1`.

```
FR_OBJECT_machine = Middle_Machines{
cell = Factory;
technicians = Apprentice, Machinist;
mttf = 120;
mttr = 6;
number = 3;
} Machine_2{output-buffer = Buffer_2},
  Machine_3{output-buffer = Buffer_3},
  Machine_4{output-buffer = Buffer_4};
```

This block of object instantiation code instantiates three identical replications of the class definition `FR_OBJECT_machine`, named `Machine_2`, `Machine_3`, and `Machine_4`, and places them in the cell `factory`. Again, we are using the specified name option. Here, however, there 3 entries in the specific name list, and the attribute `number` is assigned the value 3. Thus, each replication will have one of the names in the specific name list, and the entry in the field NAME, in this case `Middle_Machines`, is the name of the collective block of machines created. Were there to be two names in the NAME field, two seperate banks of three identical machines each would be created.

```
FR_OBJECT_buffer = Buffers{
in_cell = Factory;
```

```
capacity = [15,12];
holding_cost = [10,14];
number = 4;
} Buffer_1, Buffer_2, Buffer_3, Buffer_4, Buffer_5;
```

This block of object instantiation code creates a block of buffers with the names as denoted in the specific name list.

```
FR_OBJECT_technician = Apprentices{
qualified_on_machines = Machine_1, Machine_5;
number = 3;
} Apprentice_1, Apprentice_2, Apprentice_3;
```

This block of object instantiation code creates three apprentice technicians, each of whom is qualified on stations 1 and 5. The collective name for these three technicians is `Apprentice`.

```
FR_OBJECT_technician = Machinists{
qualified_on_machines = Machine_1,
                        Middle_Machines,
                        Machine_5;
number = 2;
} Machinist_1, Machinist_2;
```

This block of object instantiation code creates three apprentice technicians, each of whom is qualified on stations 2,3, and 4. The collective name for these three technicians is `Machinist`.

This ability to refer to groups of replications by a collective name allows for easy transformation of the textual representation. Suppose, for example, the manager of the factory were to higher 4 more Machinists. The only piece of code that would have to be changed is the block above. The change could be made in several ways. First, the entry in the field NAME could have two components added to it. That is, the instantiation could begin `FR_OBJECT_technician= Machinists, Machinists2, Machinists3`, which would create three blocks of two identical technicians in each. Second, the value assigned to the attribute `number` could be set to 6, in which case 4 addition replications would be created in the group `Machinists`, with names automatically assigned to them.

5.2.2 Instantiation of the MR schema

Instantiation of the objects in the MR schema procedes in exactly the same fashion as the instantiation of the objects in the FR schema.

```
MR_OBJECT_widget = product-A{
product-type = 1;
} part-type-A-station-1{nextpart=part-type-A-station-2},
     part-type-A-station-2{nextpart=part-type-A-station-3},
     part-type-A-station-3{nextpart=part-type-A-station-4},
     part-type-A-station-4{nextpart=part-type-A-station-5},
     part-type-A-station-5;
```

Here, using the specified name option, the five different part types representing the various stages of production of product A are created. As the flow is deterministic (that is, product A has a process flow that takes it from station to station in the following order $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$), it can be completely determined here.

```
MR_OBJECT_widget = product-B{
product-type = 1;
} part-type-B-stage-1,
     part-type-B-stage-2,
     part-type-B-stage-3;
```

5.2.3 Instantiation of the PR schema

5.2.4 Instantiation of the DR schema

5.3 The physical predicates

The physical predicates define the necessary prerequisites for an action to be taken. These form the first of the two filters.

5.3.1 The syntax of the physical predicates

5.3.2 An example of the physical predicates

Chapter 6

Representing scheduling policies

6.1 The scheduling representation

6.1.1 The scheduling representation syntax

6.1.2 An example of the scheduling representation

6.2 The textual representation of the information objects

6.3 The scheduling predicates

Chapter 7

Scheduling policies

Chapter 8

Conclusions

8.1 Conclusions

8.2 Further Research

Bibliography

[1] Asbjoern M. Bonvik. *Performance Analysis of Manufacturing Systems Under Hybrid Control Policies.* PhD thesis, Massachusetts Institute of Technology, may 1996.

[2] John A. Buzacott and J. George Shanthikumar. *Stochastic Models of Manufacturing Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[3] A. G. Hamilton. *Logic for Mathematicians.* Cambridge University Press, Port Chester, NY, 1988.

[4] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[5] Miltiadis Aris Stamatopoulos. A factory representation as a design tool in a computer integrated manufacturing environment. Master's thesis, Massachusetts Institute of Technology, 1994.