A Factory Representation as a Design Tool in a Computer Integrated Manufacturing Environment

by

Miltiadis Aris Stamatopoulos

B.S. in Electrical Engineering,
Tufts University (1992)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Operations Research

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author .... [illegible] ....
Operations Research Center
May 6, 1994

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

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

Accepted by ....
Thomas L. Magnanti, Codirector Operations Research Center
Professor of Management Science

A Factory Representation as a Design Tool in a Computer Integrated Manufacturing Environment

by

Miltiadis Aris Stamatopoulos

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science in Operations Research

Abstract

The Microsystems Factory Representation is a language/representation with which the resources pertaining to production systems can be described. It consists of the organization of static data, the static data itself, and the organization of dynamic data.

The MFR, in conjunction with data describing the processes required to manufacture the products of the factory and data on the demand for those products, provides all the necessary information to perform tasks such as scheduling in real time, simulating the operation of a factory and making graphical representations of it. With the aid of MFR, the same factory representation and computer programs can be used for both simulations and real time scheduling. In this way, simulations will be more realistic and schedulers will have the opportunity to be extensively tested before they are put in use. Another way with which MFR can be used as a tool in the design process of factories is for it to provide data to programs that check the feasibility of meeting production requirements.

MFR was developed such that the set of information it consists of, and the way this information is organized, is chosen by the user. A textual representation with which the data of a factory can be specified has been developed, and a program that creates interpreters for a particular schema (organization) has been written.

MFR was designed to be used within the CAFE CIM system, developed by the MIT CIDM project, but it can be used as part of other environments as well.

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

First, I would like to thank my advisors, Dr. Stanley Gershwin and Prof. Donald Troxel for making the past two years a great learning experience. Dr. Gershwin was a continuous source of inspiration and taught me a lot through his work and experience. Prof. Troxel provided the essential background for this work, as well as creative comments throughout its duration.

Next, I would like to thank everybody in my research group: Asbjoern, Augusto, Mitchell, Jim, Stephane, Srivatsan, Sungsu, Tomoyuki and Tulio since in some way or another they have all contributed to my thesis. Special thanks goes to Asbjoern and Sungsu for answering all the computer related questions I ever had and for always being there for me. I would like to extend my gratitude to all my friends at the Operations Research Center for making my stay at MIT an enjoyable one. In particular, I want to thank Arman, Cristina, Dave, Ragu, Rodrigo, Rich, Stefanos, Sougata, Val, and Thalia. During the difficult last months at MIT, Gina and Yannis were the best companions I could ask for, and in this way helped me tremendously in writing this thesis.

Finally, I want to thank my parents for supporting me in many different ways during all my years of higher education and my brother Constantine for inspiring me through his own accomplishments.

This work was funded by the Advanced Research Projects Agency under Contract N00174-93-C-0035.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis tackles the problem of factory representation within a Computer Integrated Manufacturing (CIM) environment. The CIM system in question is the Computer Aided Fabrication Environment (CAFE) developed as part of the Computer Integrated Design and Manufacturing project at the MIT Microsystems Technology Laboratory (MTL). A factory representation provides a way to organize and store data pertaining to resources of production systems. Several application programs, such as factory controllers (schedulers) and simulators can benefit from such a representation.

1.1 The CIDM Project and the CAFE Data base

The CIDM research project deals with several aspects of semiconductor manufacturing such as process optimization, efficient capital resource utilization, process flow and factory representation as well as new technology CAD tools. Semiconductor manufacturing is very complex in nature since it involves many steps, long cycle times and high precision operations. Capital costs are very high and efficient use of resources is vital. Handling all the data required for production is a difficult task that must be tackled effectively.

The CAFE CIM system and Gestalt database [8] was developed, as part of the CIDM project, to support semiconductor manufacturing. It has been in use at the

MIT Integrated Circuits Laboratory since 1988. It uses an object-oriented data model and an explicit schema to organize the data storage. In 1993 CAFE was installed at Lincoln Laboratory of MIT where it is used to run their new fab [5].

One of the important contributions of the CIDM project and the CAFE data base is the Process Flow Representation, known as PFR [6]. PFR is a language with which the steps required in the production of specific products are described. It includes instructions to operators and other knowledge essential to the fabrication process. Some PFR data is useful in capacity estimation, scheduling and simulation. The information specified in PFR can be installed in the data base via a PFR interpreter. The MFR was designed to complement the PFR. The PFR, MFR, and demand data together compose all static data required for capacity estimation, scheduling and simulation.

1.2 Definition of the Microsystems Factory Representation

MFR is a powerful, common language/representation with which semiconductor factories can be described unambiguously.

The data necessary to describe a factory can be segregated into static and dynamic data:

- *Static data:* By static data we mean information about a factory that does not change frequently. An example of static data is information that tells us what machines exist in the factory, their characteristics, their location and so forth. Another example would be information about the technicians currently employed and their duties.

- *Dynamic data:* Dynamic data, on the other hand, does change frequently. An example of dynamic data is information on the current status of a machine, that is, whether it is operational or whether it is busy, what set up it has, etc. Another example would be information about the work in process of the factory.

MFR consists of three items:

1. The organization of static data pertaining to resources of production systems.

2. The static data itself.

3. The organization of dynamic data pertaining to resources of production systems.

By data organization, we mean the definition of the pieces of information that are relevant and the way they are related. So, for example, we might decide that it is important to know the mean time to failure of machines, and that this is part of the static data of our factory. We might also decide that it is important to know the lot(s) that a machine is currently operating on, and make this part of our dynamic data.

As already stated, MFR includes not only the organization of static data, but the static data as well. Thus, MFR will specify that it is important to know the mean time to failure of machines, and it will also provide this information. On the other hand, MFR does not include the dynamic data of a factory. This is part of the factory state.

1.3 Related Work

Work related to this thesis has been done by the SEMATECH's SWIM (Semiconductor Workbench for Integrated Manufacturing) project. A factory data model, implemented in Smalltalk, has been developed as part of it, and is used within the SEMATECH CIM framework. The introduction of new factory data is done via a CRM Loader. ManSim, a factory simulator, uses data provided via the SWIM data model. A Bridge that enables this data to be used by ManSim is necessary. Also, some work has been done on the Future Factory Design (FFD) project that uses data provided via the SWIM data model, with the aid of another Bridge.

As pointed out in [9], one of the major hurdles in the use of the SWIM data model is data entry. MFR provides a language with which all the data about a factory can be described, along with an interpreter that facilitates the use of that data. The interpreter can be used both for storing this data in a data base, and for creating C data structures that include all the information specified. Furthermore, MFR has the advantage of allowing the user to decide what information is important (i.e. the attributes of a factory), and how this information should be organized.

To the best of our knowledge, no other work closely related to this thesis has been done.

1.4 Motivation for the Microsystems Factory Representation

The need for a language that describes the resources of production systems arose when several applications of the CAFE data base needed this information. One of the most important applications is the scheduler/simulator which is discussed in Section 1.4.4. Another application is a factory design tool that is discussed extensively in Chapter 3. The MFR provides a common language for specifying factory data that can be used by all applications that need it. The advantage is that all applications can refer to a common data base and consistency of the data can be secured.

1.4.1 The need for an MFR Textual Representation and interpreter

An MFR Textual Representation is necessary in order to specify the static data pertaining to resources of production systems. Such a representation has been developed and its syntax is defined in Chapter 2. In order to use the data specified in this way, an interpreter was written that parses the textual representation and creates a representation of the factory in computer memory, and if required, in the CAFE data base. The representation in computer memory mirrors the representation in the data base. This way the data is in a format that is easily accessible by any application that needs to refer to it, either from computer memory or from the data base itself.

By making the data accessible from computer memory we allow for experiments with hypothetical factories without affecting the data in the the data base.

1.4.2 The need for a Schema Representation and interpreter

In order to make a good representation of a factory, one needs to record all the relevant information about it. This can be a difficult task since it is not easy to predict ahead of time what data will prove to be important in future applications. The above problem became apparent in the design process of such a representation where a decision on the set of data that is relevant had to be reached. Even though a lot of time and thought was spent, and several people were consulted before we were done, our representation had to be modified several times. This led to the conclusion that it is necessary to be flexible about the MFR schema. The MFR schema specifies the data included in the MFR representation of factories and it defines the way this data is organized.

An MFR textual representation interpreter can work for a particular schema. Therefore, in order to be flexible about the MFR schema an automatic way of creating MFR interpreters from a schema definition is necessary. A program that does exactly that is discussed in Chapter 2.

1.4.3 Various uses for MFR

Some of the ways MFR has been and will be useful include:

- *Help in writing simulations.* This subject is discussed in Section 1.4.4.
- *Provide data for factory controllers (schedulers).* This subject is discussed later in this section (1.4.4) as well.
- *Help in communication.* MFR provides a standard for describing factories.
- *Provide data for analytical models.* Various analytical models that can give insight to the operation of production systems need information that will be

included in MFR.

- *Aid in the design of factories.* One way in which MFR can be used as a design tool is illustrated in Chapter 3.

- *Provide the organization of dynamic information (state).* MFR includes the organization of the dynamic data that will be stored. The state of the factory, which is used by several applications, can then be realized.

- *Provide data for graphical representation of factories.* Data from the MFR can be used to make graphical representation of factories. Also, by using state information, these representations can show the status of the factory in question. This way simulations can be animated and provide further intuition on production control issues.

1.4.4 Relationship between a scheduler, MFR and other factory data

The MFR can be used in conjunction with a set of representations of other semiconductor factory data. They include:

- the *state* of the factory, a list of all relevant *dynamic* information. The evolution of the state provides a history of the factory's activities.

- the *Process Flow Representation* (PFR), a representation of the process that is used to produce a given product. PFR is discussed briefly in Section 1.1.

- the *demand*, all the information available about future requirements including known specific orders, long range forecasts, and any statistical measures of uncertainty.

- the *scheduler*, the set of rules by which the factory is run. It takes information from the current state, the MFR, the PFR, and the demand, and it makes a decision about which operation to do next, where to move lots, etc. This decision, when implemented, determines the next value of the state.

The MFR, the state (and its evolution), the PFR, and the demand, will provide all the necessary data for simulations and for the implementation of factory controllers. When a system is created from these items, the state evolves, and reports are generated that summarize the behavior of the state over time. These reports may include such performance measures as average in-process inventory, mean cycle time, machine utilization and others.

The relationship among the above items and a scheduler is illustrated in Figure 1-1. The scheduler reads the factory representation (MFR) that will tell it *where* to produce, it also reads the process representations (PFR) that will tell it *how* to produce and finally it reads the demand representation that will tell it *what* and *how much* to produce. Then, the scheduler, reads the current state of the factory, determines the actions that need to be taken, and in this way, it influences the evolution of the state.

Figure 1-1 can represent either a real factory or a simulation of a factory. The only difference lies in the state and its dynamics. In a real factory, events occur either as a result of decisions made by the scheduler (*controllable events* such as loading lots) or at random (*uncontrollable events* such as machine failures). In a simulation, the random events are synthesized by means of a random number generator.

One important advantage gained through the use of MFR in scheduling and simulation is that the representation of the factory and the organization of the state are the same, whether used as part of a real-time factory management system, or a simulation of a factory. PFR and future demand information can also be used in the same way, whereas current orders may have to be synthesized in a simulation.

When a simulation is performed, a simulator manager will create the initial state, request a tentative schedule from the scheduler, and follow the schedule until the first random event occurs. At that point it will request a new tentative schedule from the scheduler and follow that schedule until the next random event occurs. This is repeated until the simulation is completed. In the actual operation of the factory, the random events will not be generated by a random number generator but will actually occur in the factory floor. Also, managers or workers review the proposed schedules,

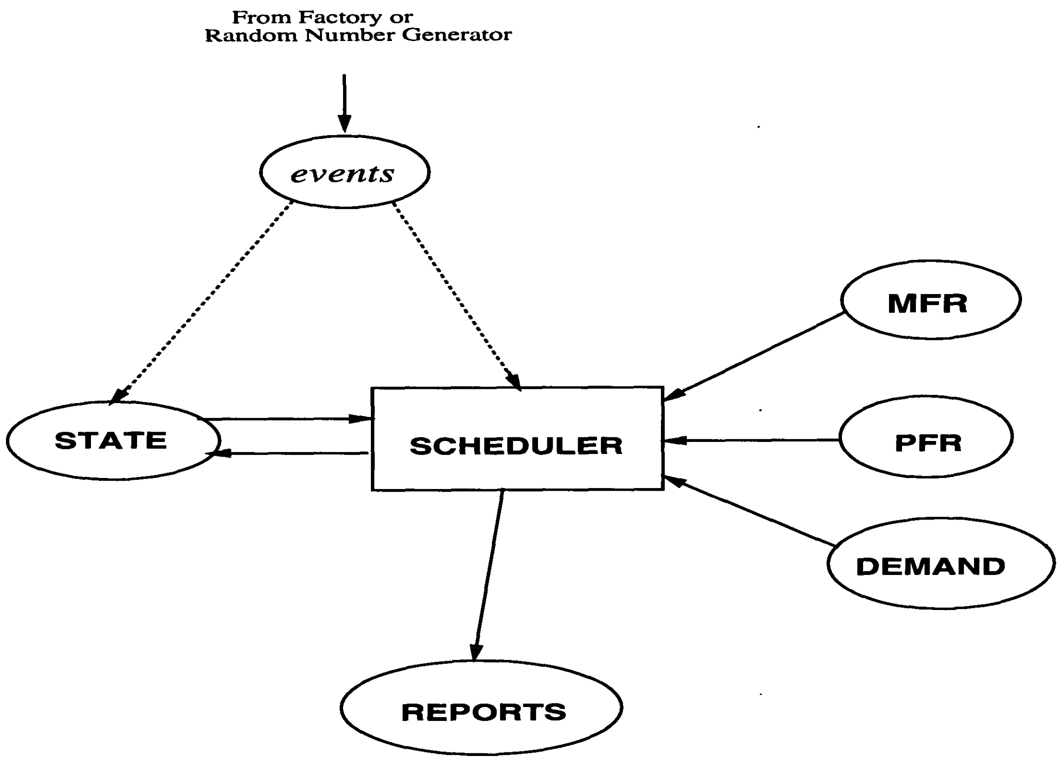

Figure 1-1: Relationship between a Scheduler, MFR, PFR, Demand, State and Reports.

and may override them if they are aware of information not available to the scheduler or to the authors of the scheduler.

The fact that exactly the same factory, process and demand representations can be used both for simulations and real time control has the following consequences:

1. The same computer programs can be used for both simulation and real-time management systems. Computer programs that are written for the simulation, and are debugged, have survived extensive testing. They are therefore more reliable than if they were written from scratch for a management system.

2. The simulator is likely to be a better representation of the actual factory than if it were developed separately from the management system. This makes the simulation more realistic and therefore its outcome more believable.

More information on the scheduler can be found in [4].

1.5 Thesis Outline

Chapter 2 of this thesis is an overview of the Microsystems Factory Representation. There, the MFR textual representation, the MFR schema representation and their relationship are introduced. Chapter 3 illustrates one way in which MFR can be used as a factory design tool and discusses several others. We describe a program that checks for the feasibility of the production requirements of a factory specified via MFR.

In Chapter 4 of this thesis, we offer a formal definition of the MFR textual representation syntax. Also, an interpreter for this representation which makes data easily accessible is described. Then, in Chapter 5, the MFR schema representation is defined, and a schema interpreter is discussed.

Chapter 6 is a programmer's manual for the schema and MFR interpreters. It describes the architecture and data structures of the programs. It also explains how one may use the MFR data directly from the representation created by the MFR interpreter in computer memory. Finally, in Chapter 7 we summarize and offer some conclusions.

Chapter 2

Overview of MFR

This chapter provides an overview of the MFR textual representation, the MFR schema representation and their relationship.

2.1 Introduction

The MFR textual representation is a language with which the resources of factories can be described unambiguously. The schema of this representation is the set of classes that will be used for that purpose. The syntax of the MFR textual representation depends on the schema that is being used, but will always have the same structure. For any given schema, there is one and only one possible syntax.

The schema textual representation is a meta language with which the schema of an MFR textual representation can be specified. A program has been written that creates interpreters for the MFR textual representation given a particular schema.

2.2 A Factory Schema

The schema of the MFR textual representation determines how the information about a factory will be organized. An example of a possible schema is illustrated in Figure 2-1. In this schema, the data is organized hierarchically. The factory is represented as a collection of cells that form a tree-like structure. The cells themselves are collections of machines, buffers, and lower level cells. The factory is the top-most cell and is decomposed into subcells. The leaf-cells are composed of machines and buffers only, and therefore do not include sub-cells. Observe that there can be one or several levels of cells, depending on how the user chooses to organize the factory. Machines and buffers can be part of cells at any level. In practical terms, a cell can be a portion of the factory with some special characteristic. For example, the set of photolithography machines of a semiconductor factory may form a cell of their own.

The MFR allows the user to organize a factory hierarchically, but it does not require such an organization. The user can specify any organization he or she wishes.

2.3 Relationship among MFR and the Data Base Schema

As mentioned in Chapter 1, the CAFE data base uses an object-oriented data model with an explicit schema to organize the data storage. That means that in order to store some information in the data base we must first define how this information is organized. For example, the factory schema described in the previous section would imply a data base schema with the following classes (object types): MACHINE, CELL, BUFFER. It would also imply that some attribute of the class CELL would refer to a list of objects of class MACHINE, some other attribute would refer to a list of objects of class BUFFER, and yet another attribute would refer to a list of objects of class CELL. The data base schema could also specify other attributes of the class CELL that would be used to store various kinds of information, such as a cell name, or possibly a cell description. Classes of type MACHINE and BUFFER would have attributes of their own.

The MFR organization of a factory mirrors the way data is organized in the data base. The information provided by the data base schema is required by MFR since it defines the data structures. Also, the syntax of the MFR textual representation depends on the schema used. Finally, in order to store MFR data in the data base, the schema used by MFR must be installed in the data base.

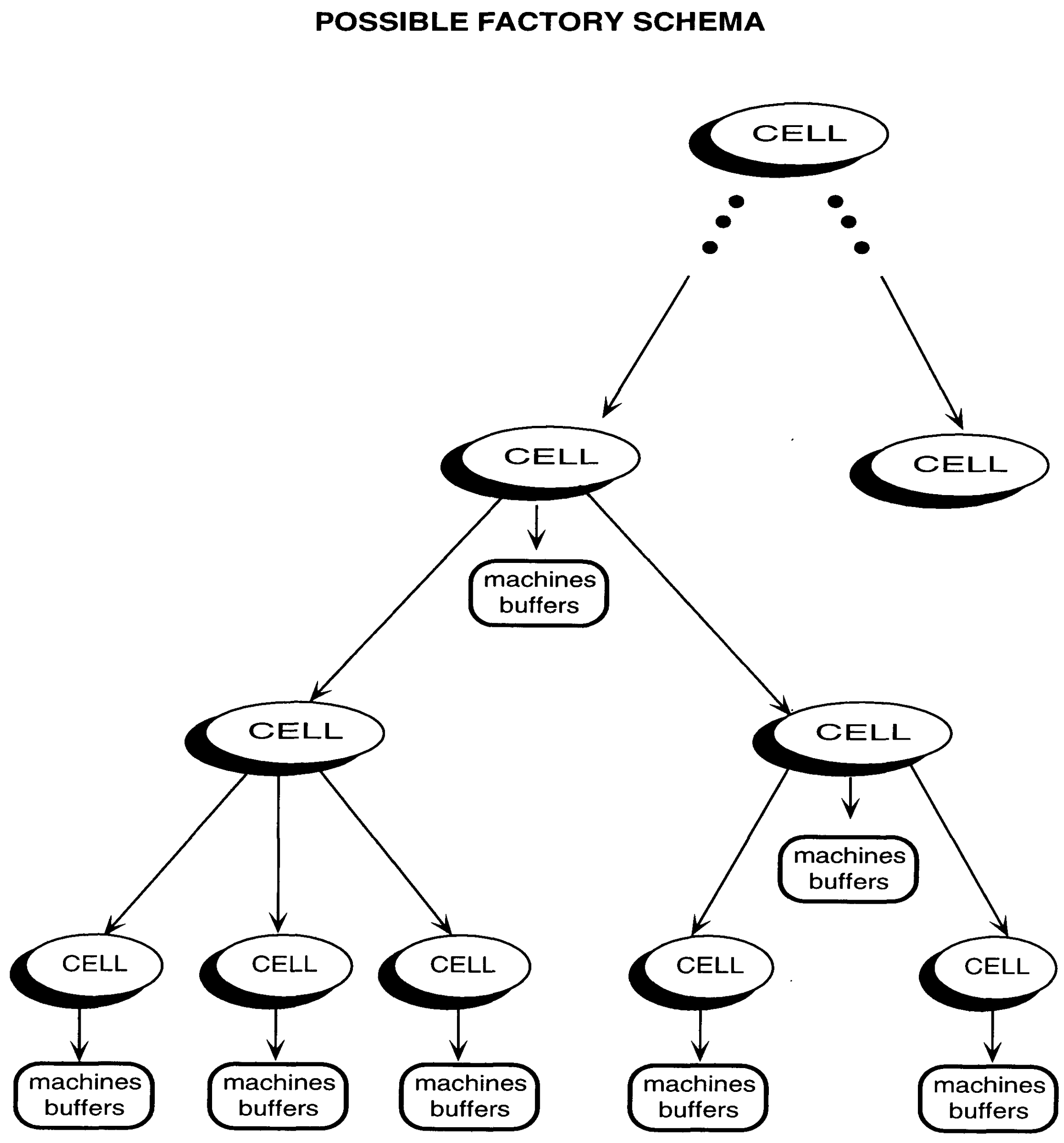

Figure 2-1: A possible factory schema.

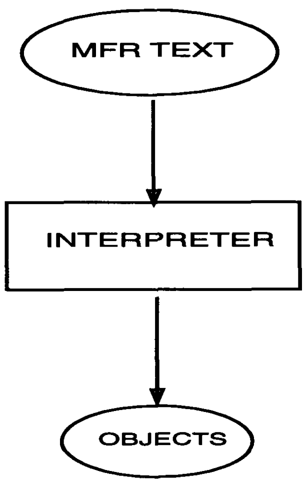

Figure 2-2: Instantiation of MFR in the CAFE data base.

2.4 The MFR Interpreter

The MFR interpreter can parse the textual representation, perform various consistency checks on the data provided and create a representation of the MFR objects in computer memory. The interpreter can also be used to store the information provided by the textual representation in the CAFE data base. In other words, if the schema used has been installed in the data base, the interpreter can instantiate the MFR objects in it. This process is illustrated in Figure 2-2. If the schema has not been installed in the data base, application programs can use MFR data directly from the C data structures the interpreter creates.

An important feature of the MFR interpreter is that it allows the user to define multiple identical objects once and to specify how many exist. This feature applies only to classes that have an attribute with keyword **`number`** that must be of type **`integer`**. These classes must have a singleton attribute with keyword **`name`** of type **`string`** or **`varstring`** as well. The MFR interpreter is then responsible for creating all the identical objects, and if necessary for providing default names for those objects. Identical objects are objects that have the same values for all their attributes, except for the **`name`** attribute.

2.5 The MFR Schema Representation

The MFR schema representation is a language with which the schema of an MFR textual representation can be described unambiguously. A schema interpreter has been developed that parses the schema representation and creates interpreters for the MFR textual representation. More about this program is said in Chapter 4. In order to use the CAFE data base to store the MFR objects one must install the schema that is being used in the data base. The relationship between the MFR schema, the MFR interpreter and the data base is shown in Figure 2-3.

2.6 Using MFR to Describe a Factory

In this section, a simple example of using MFR to describe a factory is presented. For a formal definition of the syntax of the MFR schema representation and the MFR textual representation please refer to Chapters 4 and 5 respectively.

2.6.1 Specifying the schema

A schema consisting of three classes will be described here. The schema has the characteristics described in Section 2.2, and the classes described in Section 2.3.

Attributes of class CELL:
Static

- STRING name;
- LIST machines; (MACHINE)
- LIST buffers; (BUFFER)
- LIST subcells; (CELL)
- CELL supercell;
- STRING description;

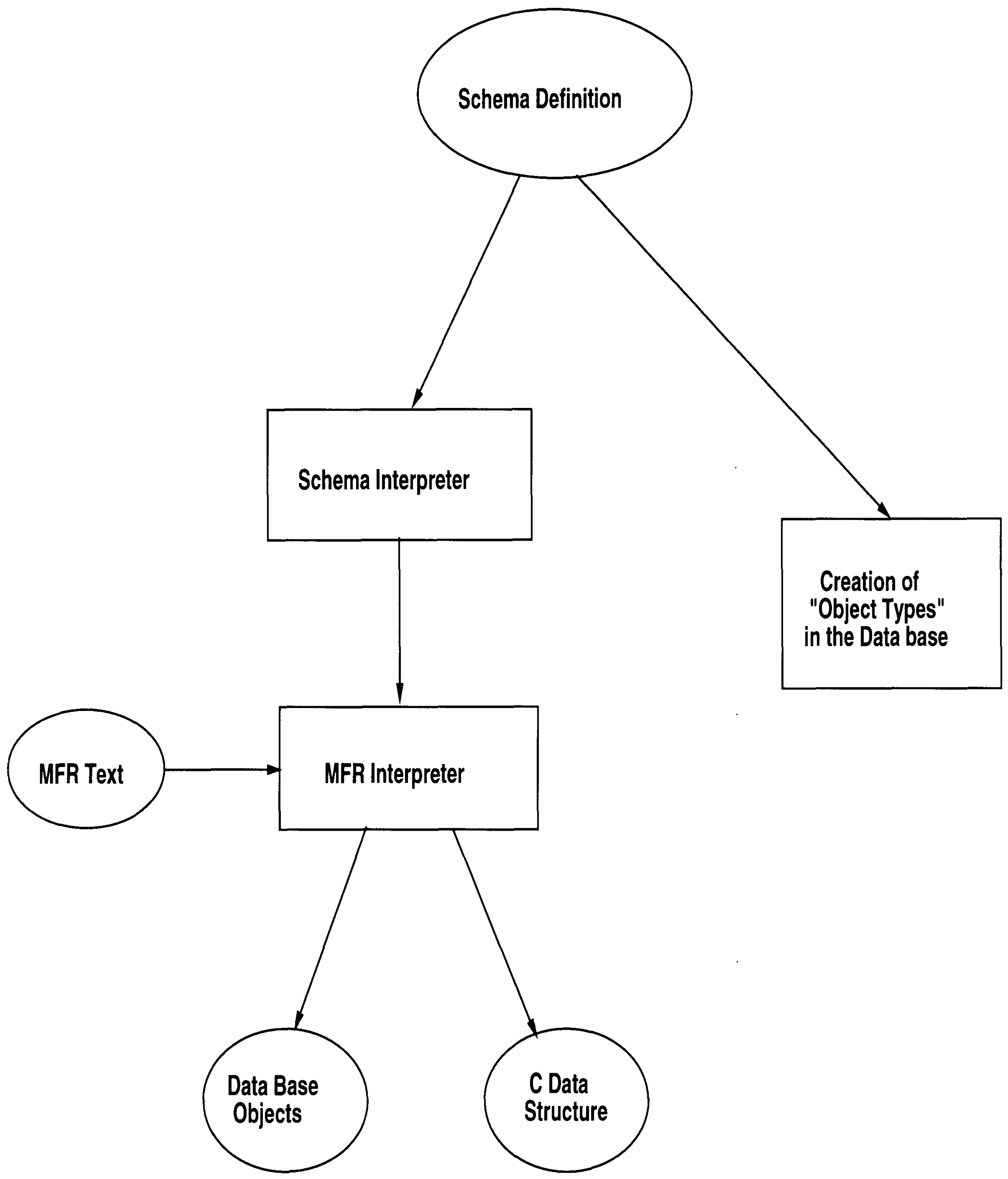

Figure 2-3: Schema representation, the MFR interpreter and the data base.

- INTEGER number;

Attributes of class MACHINE:

Static

- STRING name;
- CELL cell;
- INTEGER batch; (max batch size)
- FLOAT mttf;
- FLOAT mttr;
- INTEGER number;

Dynamic

- STRING status; (busy, idle, down, engineering)
- STRING detailed status; (if busy, operation; if down, failure mode)

Attributes of class BUFFER:

Static

- STRING name;
- CELL cell;
- INTEGER capacity; (max number of lots)
- INTEGER number;

Dynamic

- LIST lots; (LOT)

The type of the attributes of an object is indicated by capital letters. Only the dynamic attributes will be mutable in the data base. In order to change a static attribute one must reinstall the MFR. Observe that a dynamic attribute of the class BUFFER is a LIST of type LOT, even though LOT is not an MFR class.

This schema is a subset of the schema presented in Section 5.2. For the formal schema textual representation defining these classes, please refer to Chapter 4.

2.6.2 Specifying the factory

The syntax of the MFR textual representation is a C-like syntax. Some of its features are demonstrated in this section through an example.

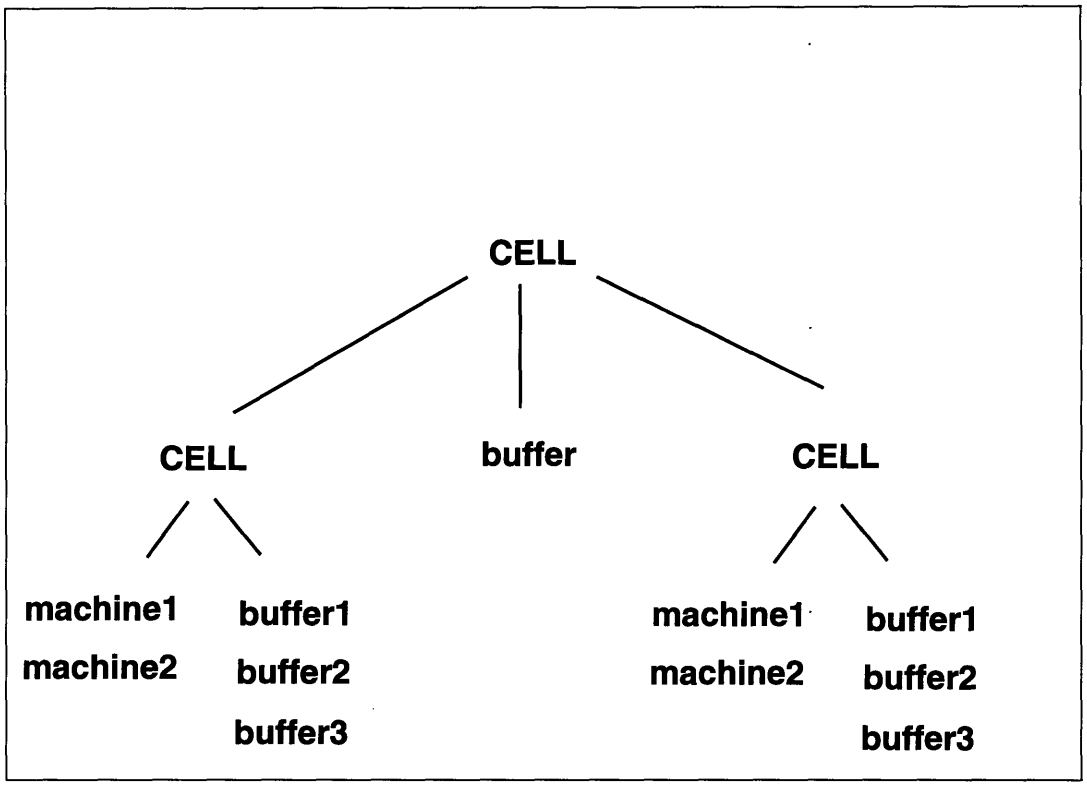

Figure 2-4: Possible factory configuration

Consider the factory configuration of Figure 2-4. The highest level cell is the factory to be represented. Machines **machine1** and **machine2** are identical, and so are the buffers **buffer1**, **buffer2** and **buffer3**. The leaf cells are identical as well.

We want to define this factory with names as shown in Figure 2-5.

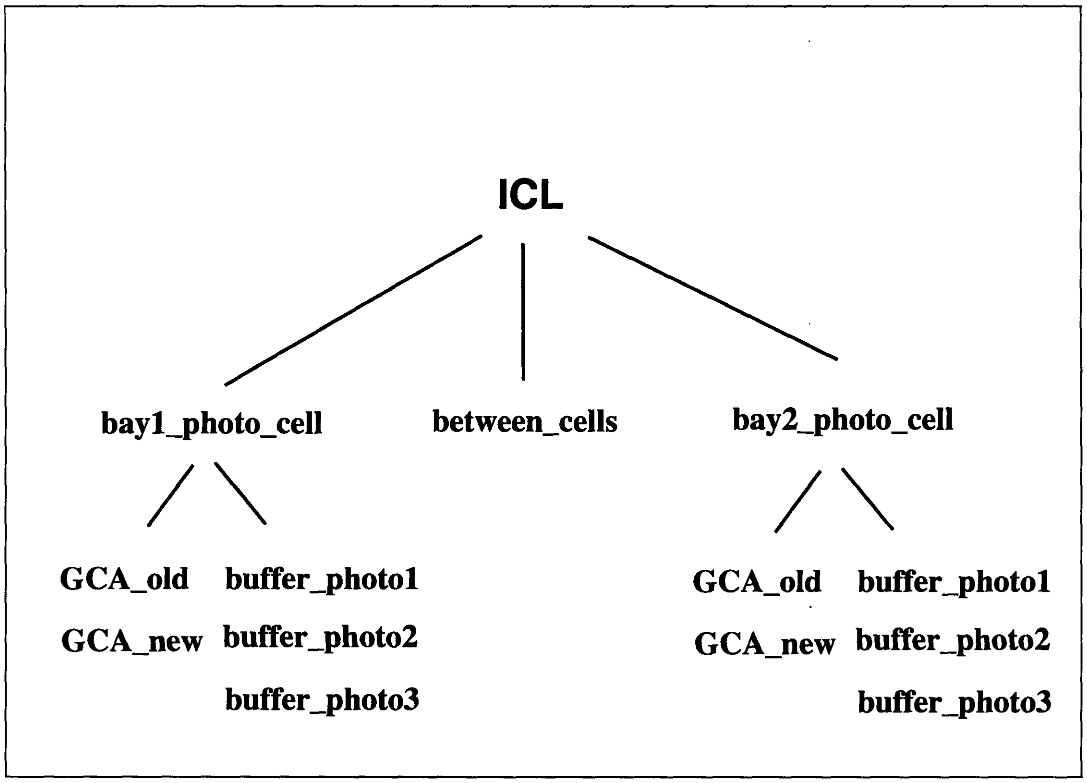

Figure 2-5: Example of a factory

The MFR textual representation of this factory is the following:

```
/* Higher level cell ICL */
CELL ICL {
description ''Integrated Circuits Laboratory at MIT'';
number 1;
buffers ''between_cells'';
};
/* photolithography cells in room 3-110 */

CELL photolithography {
```

```
description ''These cells include the photo equipment'';
supercell ICL;
number 2;
}bay1_photo_cell bay2_photo_cell;

/* buffer between photo cells */

BUFFER between_cells {
capacity 300;
number 1;
};

/* machines in photo */

MACHINE GCA {
cell photolithography;
batch 3;
mttf 14.0;
mttr 2.0;
number 2;
}GCA_old GCA_new;

/* buffers in photo */

BUFFER buffer_photo {
cell photolithography;
capacity 100;
number 3;
};
```

The first block is the definition of cell ICL, which is the highest level CELL. The **description**, **number** and **buffers** attributes are specified. The **machines** and **subcells** are not specified, so the fact that they belong to ICL should be specified when the corresponding objects are defined. Since there is only one CELL ICL, the name of the CELL will be ICL.

The next block is the definition of CELLs **photolithography**. Note that the particular names of each cell are specified to be **bay1_photo_cell** and **bay2_photo_cell**. This is done in a list between the closing bracket of the object and the semicolon terminating the object definition. Had they not been specified, they would be **photolithography1** and **photolithography2**.

Next, the buffer **between_cells** is declared. The CELL it belongs to is not specified. This is not a problem because the fact that this buffer belongs to CELL ICL is specified in the first block. It would therefore be redundant for it to be specified again. Redundant specifications are allowed if they are consistent.

The next two blocks are the definitions of the machines and buffers of the cells photolithography. Since there are two cells photolithography, the machines and buffers defined are 4 and 6 respectively, even though the number attributes are 2 and 3. The particular names of the buffers are not written, so the default names **buffer_photo1**, **buffer_photo2** and **buffer_photo3** are given.

The order in which the attributes of the objects are defined is irrelevant. In fact, it is not necessary to give a value for all the attributes. Default values for them are provided in the MFR schema representation. Another feature of MFR not illustrated in this example is that objects can be specified in a nested fashion. These features and more are illustrated in Section 5.2.

2.7 How to use the Schema and the MFR Interpreters

In order to use the schema interpreter, one must first write a schema representation in the syntax defined in Section 4.2. Then, a directory where the source code for the interpreter will be written must be created. The command for the schema interpreter is:

```
schi [-s] <infile> <subdirectory> [outfile]
```

The `-s` option should be used only if the schema represented has been installed in the data base. If this option is set, the MFR interpreter created will have the ability to instantiate objects in the data base. The file in which the schema representation is stored and the directory where the MFR interpreter will be created must follow. Then the user may specify an output file. The default name for this file is `out`. In this file a representation of the schema, as it was read by the interpreter will be printed. If any syntax or consistency errors are made, error messages will be printed to standard error. The output file for the example of Section 4.3 can be found in Appendix A.

Once the source code has been written, it should be compiled. First, the lex lexical analysis program generator and the yacc parsing program generator must be used to create a lexical analyzer and a parser. This can be done by typing `lex clex` and `yacc cyacc`. Then, the Makefile shown in Appendix B can be used to compile and link the MFR interpreter.

The following command can be used for the MFR interpreter:

```
mfri [-s] <infile> [outfile]
```

The MFR textual representation file must be written in the syntax defined in Section 5.1. If any syntax or consistency errors are made, error messages will be printed in standard error and in the outfile. If the `-s` option is set, and the schema used has been installed in the data base, the objects will be instantiated in it. The user may specify an output file. If not the default name for this file is `out`. Provided that no errors have been made, a representation of the objects will be printed in the outfile. The computer memory address where the objects are stored will be printed next to each one of them. The output file for the example of Section 5.2 can be found in Appendix A.

It is possible to use the MFR objects directly from computer memory, as they are created by the MFR interpreter. For that purpose, some modifications of the main function in *main.c*, and an understanding of the data structures in *interp.h* is necessary. The data structures used to create the objects in computer memory are described in the programmer's manual for the MFR interpreter in Chapter 6.

Chapter 3

Use of MFR as a Design Tool

There are several ways in which MFR can be useful in the design process, in simulation and in the real time control of a factory. Some of these are discussed in Chapter 1. Here we first give a description of the tools that can be used, and then we illustrate one way in which MFR can be useful in the design of a factory. This is done with the aid of a program that checks the feasibility of meeting production requirements in a hypothetical factory described via MFR.

3.1 Tools in the Design and Control of Factories

Some tools useful for the design and control of factories, and the data they require are illustrated in Figure 3-1 [1]. Tool number 1 is a feasibility check against the existence of the machines required to produce the products described by PFR. Tool number 2 is an approximate feasibility check against machine capacity. This tool is discussed in detail in Sections 3.2 and 3.3.

The third tool is a more sophisticated capacity feasibility check, that takes queuing delays and limited buffer space into account. No scheduling policy is required for this tool. The next item, policy development, refers to choosing a method for scheduling the production process. Item 5, system design, is the step where the set of resources for the production system in question is decided. This decision should be based on

[1]This picture was made by Dr. Stanley Gershwin.

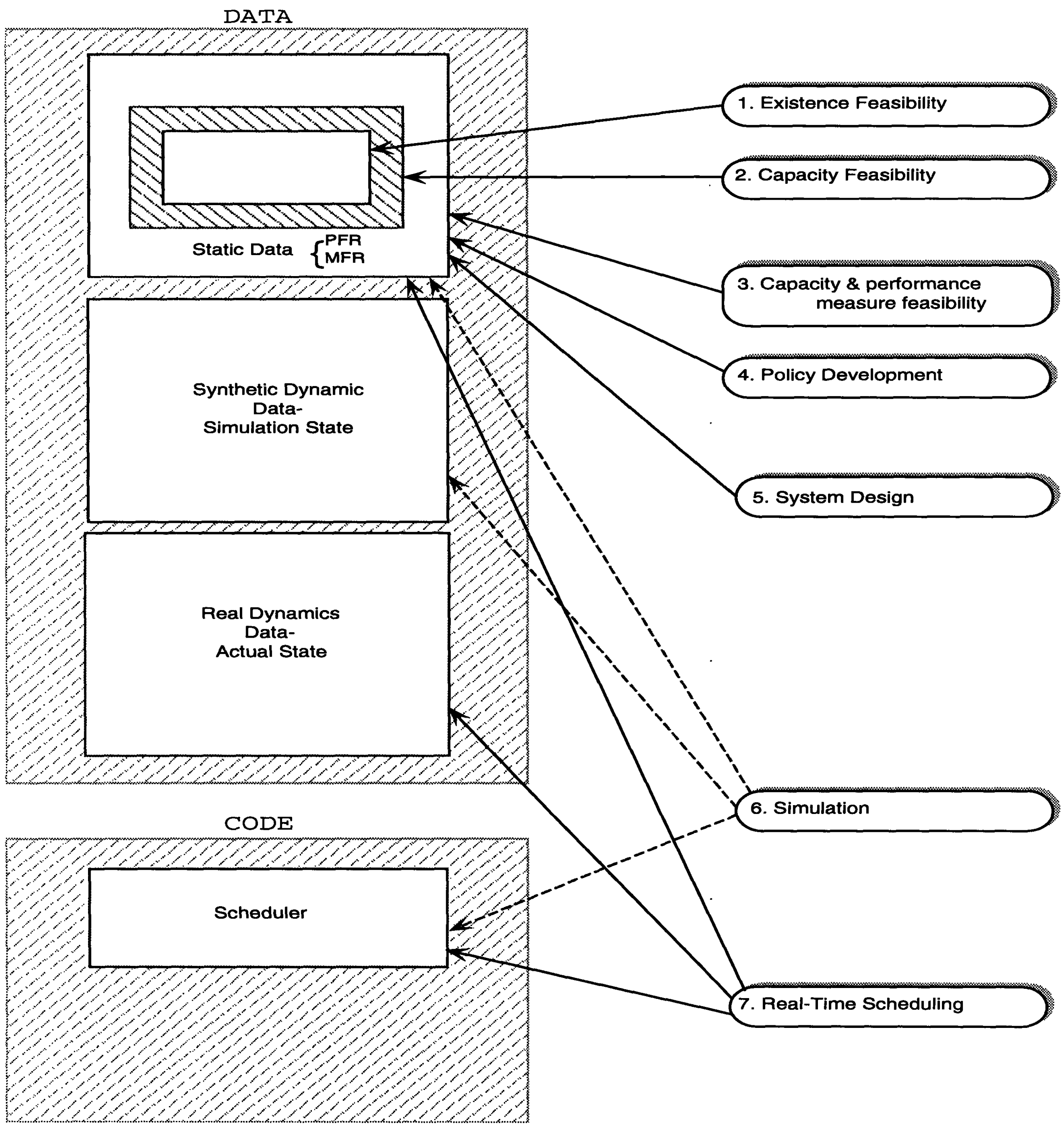

Figure 3-1: Tools in the design and control of factories.

the previous steps, after several designs have been considered. Tools 1 through 5 use only static data, namely the description of the factory (MFR), the description of the processes required to produce the products (PFR), and the demand for those products.

Before finalizing the design of a factory, it is a good idea to make a full scale simulation. The same scheduling algorithm and code as the ones intended to be used for the real time control of the factory should be used in the simulation. Also, the same static data and the same organization of dynamic data must be used.

The real dynamic data of the factory in question is a representation of the current state of that factory. In the case of a simulation, the state of the factory at every tick of the clock will be stored as if it were real dynamic data. The same schema, determined by MFR, will be used both for storing real dynamic data, and dynamic data synthesized by the simulation manager.

Finally, the real time scheduling tool can be an aid to management for the control of the factory. The only differences between real time scheduling and simulation is that the dynamic data in the real time scheduling case will be real as opposed to synthetic, and that in the real control, the management will have the opportunity to modify the schedules proposed by the scheduler.

3.2 Checking the Feasibility of Factory Requirements

In the design of a factory, one of the most important concerns is whether the factory to be built will meet the production requirements. There are several ways to tackle this problem including analytic and simulation techniques. Here, we describe a program that first makes sure that all the machines needed for every product are present, and then calculates the utilization of every machine in the factory. A programmer's manual for this program can be found in Appendix D. By utilization we mean the percentage of time a machine will be operating given it is operational. Failure and repair rates are taken into account when performing capacity calculations.

The capacity check performed by this program is only a first cut approximation to determine whether the production requirements can be met. This is so because no queuing delays or limits are taken into account. Still, the program offers a very quick and useful way of rejecting many infeasible proposed designs. It will flag all the machines that have utilization higher than an upper bound and lower than a lower bound specified by the user. That way it is easy to identify machines whose capacity constraints may cause problems, and machines that will be idle most of the time. Then, with the help of the MFR textual representation, one may change the number of certain machine types and run another feasibility check.

3.2.1 Data required in order to perform the feasibility checks

In order to perform the feasibility checks we need a representation of the factory, a representation of the processes of the products to be manufactured, and the demand for those products. We also need the number of hours that the factory will be operating in a year.

The factory in question is specified via MFR, and the processes required to manufacture the products via PFR. PFR (Process Flow Representation) is a language with which the processes for making semiconductor products can be described. It is discussed in Chapter 1 and in [8]. The demand for the products is expressed as a number of lots per year for each product. Finally, the number of working hours in a year is specified as the number of days that the factory will be operating in a year, and the number of working hours in a day.

For this particular program, the mean time to failure and the mean time to repair for every machine are required. Therefore, an MFR schema in which this information is included must be used. Also, it is very useful to be able to change the number of machines of any type of the factory easily. For that purpose, a schema in which the MFR_MACHINE objects have a **number** attribute should be used. The schema described as an example in Section 4.3, which is the schema installed in the CAFE data base, was assumed when writing the feasibility checking program.

Both the information about the factory and the information about the processes of the products to be produced is read from the relevant data base objects. The process flow data though may be cached in order to speed things up.

3.2.2 Calculating machine utilization

When calculating the machine utilization we need to know the number of hours per year a particular machine is available. For that purpose we need the efficiency of the machine, which we calculate with the following formula taken from [2].

$$e = \frac{mttf}{mttf + mttr}$$

where $mttf$ and $mttr$ are the mean time to failure and mean time to repair respectively.

Let the time required from a unit of product j on machine i be t_{ij}. Also, let the annual demand for product j be d_j and the total number of hours the factory is operating per year be h. Then, the utilization of machine i, denoted as u_i is:

$$u_i = \frac{\sum_j t_{ij} d_j}{e_i h} \times 100\%$$

In words, the number of hours the machine is operational in a year is the number of working hours in a year times the efficiency of the machine (eh). The number of hours a machine is needed in a year can be calculated by adding all the time required for the machine by each product times the demand for that product ($\sum_j t_{ij} d_j$). Then, the utilization of a machine is the fraction of hours required by the machine divided by the hours the machine is operational times 100.

3.3 Example of Checking the Feasibility of Factory Requirements

The Integrated Circuits Laboratory at MIT was used as the basis for the example that is described in this section. An MFR description of this factory was written, and objects were created in the CAFE data base. An MFR textual representation of ICL can be found in Appendix C. The schema used is as described in Section 4.3. This particular program can read the factory data from the data base, or directly from the MFR textual representation. This was accomplished by calling an MFR interpreter from within the program, and reading the relevant data from the data structures created. Several combinations of products were tried with different demands for each one of them. The processes used for the following example are processes that had been described via PFR, and had been installed in the data base.

Following is the input file used for this example. The first number, 250.0, indicates the number of days in a year the fab is operating, while the second number, 10.0, indicates the number of hours in a day it is open. Then, process names and demand in number of lots per year required follow.

```
250.0   10.0


DA                     155.5
MICROBRIDGE            133.4
PLASMATWO              12.3
WELL-OXIDE-WET-ETCH    20.0
JUNCTION-DRIVE         53.0
RESIST-FLOW            16.0
```

The command line is:

```
fsblt [-c] [-d] [-m] <infile> <mfrfile> [upbound lowbound]
```

The `-c` option specifies that the feasibility check against machine capacity will be performed, and the `-d` option that the processes will be read from the data base. Once they are read, the information required is cached, in a `<infile.cache>` file. If the `-d` option is not there, the program will look for this file and read the necessary process information from there. The `-m` option indicates that the factory data should be read from the data base. If this is chosen, the `<mfrfile>` will not be read. If one wants to write their own .cache file, they should use the following format:

1. For each process, a list of machines and time-durations in seconds must be specified as shown: `asher 5400`. If machines are visited multiple times, every time must be specified.
2. The descriptions of the processes must be in the same order as they appear in the input file. Every process must end with `- 0`.

A small example of a cache file follows.

```
     stepper      3600
   developer      3000
  microscope      1800
       oxide      1200
      tubeB5      3600
           -   0
       oxide      1560
ellipsometer       900
           -   0
         rca      7200
ellipsometer       900
           -   0
```

The `<mfrfile>` is the file in which the MFR textual representation of the factory in question is stored. The upper and lower bounds are the the bounds of utilization that will determine whether a machine is flagged in the output or not. The default values for them are 70.0 and 20.0% respectively. A sample run was done:

```
mstamato@hierarchy 98 % fsblt -c -d processes icl.mfr 80 15
reading processes from the data base...
Feasibility check 1 passed.
Feasibility check 2 failed: excessive demand for 2 machines.
```

Two output files summarize the results of the run. The **<infile.brf>** file indicates the total operating time for all processes, and whether the existence feasibility check was passed. Then, it flags the machines that are outside the utilization bounds specified, and the machines whose utilization are more than 100%. Finally, it indicates whether the feasibility check against machine capacity was passed. The brief output file of our example follows.

```
total operating time for process                  DA:     205.97
total operating time for process     microbridge:      22.77
total operating time for process       PLASMATWO:      14.25
total operating time for process WELL-OXIDE-WET-ETCH:   0.68
total operating time for process JUNCTION-DRIVE:        3.25
total operating time for process     RESIST-FLOW:       0.83


Feasibility check 1 passed.
machine              prometrix has low utilization:      1.56
machine                 dektak has low utilization:      2.89
machine                 varian has low utilization:     14.41
machine                    VTR has low utilization:      0.00
machine              pre-metal has low utilization:     11.38
machine                nitride has low utilization:     11.51
machine                 tubeB8 has low utilization:      0.00
machine                 tubeB6 has low utilization:      6.34
machine                    rca ** FAILED capacity check, utilization:    230.66 **
machine              etcher-3 has low utilization:       7.16
machine              etcher-2 has low utilization:       7.23
machine             developer has high utilization:  82.60
machine               stepper has high utilization:  89.45
machine                 asher ** FAILED capacity check, utilization:  128.14 **

Feasibility check 2 failed: excessive demand for 2 machines.
```

The `<infile.rprt>` file includes all the information of the brief file, and the utilization of all machines in the factory.

Chapter 4

The MFR Schema Representation

The MFR schema representation is a representation of the information required to create MFR textual representation interpreters and to install a schema in the data base. Please refer to Chapter 2 and to Figure 2-3. In this chapter we describe the data it consists of, and we offer a formal definition of its syntax.

4.1 Information Required About the Schema

In order to define the schema it is necessary to define all classes (object types) that are part of it. A keyword and a list of attributes must be specified for every class. The attributes themselves must have a keyword as well as the following information, which is organized in slots in the schema representation, specified:

type: An attribute type can be an `integer`, a `float`, a `boolean`, a `string`, a `varstring` or an `object`. `Varstring` and `string` are treated exactly the same way until the objects are stored in the data base. This is the case for `integer` and `boolean` as well.

1-1 vs 1-m: This slot will tell whether the attribute corresponds to a single attribute value or if it corresponds to a list of attribute values.

active vs passive: There may be cases where the same information is included in two attributes. For example, consider the schema described in Section 2.2. Assume the class CELL includes both an attribute that specifies the supercell in which the cell in question belongs to, and an attribute that specifies the subcells that belong to this cell. Even though it is required for application programs to have both attributes, it is redundant to store both of them since one of them would be enough to describe the hierarchy. In such a case, one of those attributes should be passive. This slot will be used by the schema interpreter if the schema has been installed in the data base.

invertible vs non-invertible: If an attribute of a particular class in the data base is invertible, there is a query that will retrieve the object(s) given the attribute's value. This slot is not used by the schema interpreter, but it should be specified anyway.

uniqueness: If an attribute is unique, two objects in the data base can't have the same value for this attribute. This slot is not used by the schema interpreter, but it should be specified anyway.

mutable vs non-mutable: If an attribute is mutable, its value can be changed in the data base. Attributes that describe dynamic data should be mutable; attributes that describe static data should not be mutable.

prefetched vs not prefetched: If an attribute is prefetched, its value is retrieved from the data base when the object is. If, on the other hand it is not prefetched, its value is only retrieved when the attribute is accessed. Passive attributes can not be prefetched. Therefore attributes that are not frequently used should not be prefetched. Again, this slot is not used by the schema interpreter, but it should be specified anyway.

child vs parent: If an attribute is an object, it is important for the schema interpreter to know whether this object is a parent object, or a child. The reason it is important is because the MFR interpreter created will be responsible for replicating identical objects that have a number attribute with value greater than 1. If the object(s) to be replicated have children objects, those objects will be replicated as well. The objects created that way (that is, objects replicated because their parents were replicated) will keep their original name (if such an attribute exists), but will have a different, newly created, parent. As an example consider a machine GCA that belongs to a cell PHOTO that has number 2, and assume machine is a child. The MFR interpreter will create cell PHOTO1 and PHOTO2, and two machines GCA, one with parent PHOTO1 and one with parent PHOTO2. If machine were a parent object, both PHOTO1 and PHOTO2 would have the same machine GCA as the value in their corresponding attribute.

The other reason it is important for the MFR interpreter to know whether an object is a parent or a child is because the MFR interpreter may be responsible (if the schema has been installed) for creating the objects in the data base. Objects of classes that are part of the hierarchical structure will be created recursively, but objects of classes that are not part of it will be created separately. Also, the MFR interpreter must make sure it does not create objects twice. As an example consider a schema that has objects of type `CELL`, of type `MACHINE`, that are children, and of type `TECHNICIAN` that are not part of the hierarchical structure but have an 1-m attribute of type `MACHINE`. Objects of type `MACHINE` will be created recursively when their parents are, but they should not be created again when an object of type `TECHNICIAN` is created.

Parent attributes must be 1-1.

default value: The MFR interpreter will give default values to attributes that have not been specified via the MFR textual representation. The default value of an attribute should therefore be part of the schema.

If the type of an attribute is a class, this class must be defined. If the class is not an MFR class, but a class that is part of the data base, this should be indicated.

Such attributes must be mutable, since they can not be specified before the MFR objects are created in the data base. As an example consider the the one in Section 4.3 where the attribute **lots** of class BUFFER is of type LOT. The class LOT is not an MFR class, but objects of type LOT exist in the CAFE data base, and they can be referred to.

A simple example of a class definition follows.

```
MFR_BUFFER  {
name     string, 1-1, act, inv, nonunq, nonul, nonmut, prf, ,null;
cell     MFR_CELL, 1-1, pas, inv, nonunq, nonul, nonmut, prf, par, null;
capacity   integer, 1-1, act, noninv, nonunq, nul, mut, prf, , 0;
lots         LOT, 1-m, act, noninv, nonunq, nul, mut, nonprf, , null;
number ;
}
```

Here, the class MFR_BUFFER is defined. The keywords for its attributes are **name**, **cell**, **capacity**, **lots** and **number**. Since the **number** attribute is part of this object, the **name** attribute is required. The slots for each attribute must appear in a prespecified order as discussed in Section 4.2, and they can take values as shown in Table 4.2.

4.2 Syntax of the MFR Schema Representation

The symbols defined in Table 4.1 will be used in this section. Tokens will be indicated in capital letters.

The syntax of the schema representation has a block structure. Each class is specified in a block of the following form:

class-definition : NAME { **attribute-definition-list** }

;

Table 4.1: Definition of symbols

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

So the keyword of the class (NAME) is specified, and then, within the brackets, the list of attributes are specified. Keywords for classes should be in capital letters.

All the **class-definitions** together make a **class-definition-list** that ends with 'end' as shown.

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

The list must end with the keyword 'end'.
end-of-file: 'end' ;

The **attribute-definition-list** is as shown:

attribute-definition-list : /* empty */
| **attribute-definition attribute-definition-list**
;

```
attribute-definition : NAME type COMMA single COMMA act COMMA inv
                           COMMA unq COMMA null COMMA mut COMMA
                           prf COMMA hie COMMA dfl TERMINAL
                       | NUMBER TERMINAL
                       | NONMFR TERMINAL
                       ;
```

COMMA is returned by the lexical analyzer when a ',' shows up, and TERMINAL when a ';' shows up. NAME is returned when a string without quotes appears.

An **attribute-definition** can be one of three things. It can be a comma separated list, with prespecified order, of values for all the slots that need to be specified; or it can be a single keyword which returns NUMBER and indicates that the class has an attribute named number with all the implications indicated in the previews section; or it can be a single keyword witch returns NONMFR indicating that the class is not part of the MFR schema but already exists in the data base. If an attribute with keyword 'number' will be created, it will be of type **integer**, 1-1 and with default value equal to 1.

The lexical analyzer will return NUMBER whenever the keyword 'number' appears in the schema representation. Remember that if such an attribute exists, the class must also include a singleton attribute with keyword 'name' of type string or varstring.

The slots of the **attribute-definition** are specified as shown in Table 4.2.

STRINGTYPE, VARSTRINGTYPE, INTEGERTYPE, BOOLEAN and FLOATYPE are returned by the lexical analyzer when string, varstring, integer, boolean or float respectively appear. ONE_TO_ONE and ONE_TO_MANY are returned when 1-1 or 1-m appear, ACT and PAS are returned when act or pas appear, etc. CNULL is returned when null appears, STRING is returned for a string within quotes, and INTEGER for an integer number.

Table 4.2: Possible values for each slot in **attribute-definition**.

---

type : STRINGTYPE | VARSTRINGTYPE | INTEGERTYPE |
BOOLEANTYPE | FLOATYPE | NAME ;
single : ONE_TO_ONE | ONE_TO_MANY;
act : ACT | PAS;
inv : INV | NONINV;
unq : UNQ | NONUNQ | INTEGER;
nul : NUL | NONUL;
mut : MUT | NONMUT;
prf : PRF | NONPRF;
hie : PAR | CHD | /* empty */;
dfl : STRING | INTEGER | FLOAT | CNULL | /* empty */;

4.3 An Example of the MFR Schema Representation

An example of the MFR schema representation follows. This is the representation of the schema that was installed in CAFE on January 1994. It consists of six classes, namely MFR_CELL, MFR_MACHINE, MFR_BUFFER, MFR_MAINTMODE, MFR_FAILUREMODE and MFR_TECHNICIAN. Attributes of MFR_CELL include children lists of MFR_CELLs, MFR_MACHINEs and MFR_BUFFERs. A hierarchy of cells is created as discussed in Section 2.2 and illustrated in Figure 2-1.

Reference of non-MFR objects was required, so the corresponding classes were specified as non-MFR classes. These classes are LOT, TIME, and ACTIVEOPINST.

```
/***********************************************************************

  Representation of the MFR schema curently installed.
```

```
*************************************************************************/

/* Factory cell */

MFR_CELL {
name        string, 1-1, act, inv, nonunq, nonul, nonmut, prf,, ;
supercell   MFR_CELL, 1-1, pas, inv, nonunq, nul, nonmut, nonprf, par, null;
subcells    MFR_CELL, 1-m, act, noninv, nonunq, nul, nonmut, nonprf, chd, null;
machines    MFR_MACHINE, 1-m, act, noninv, nonunq, nonul, nonmut, nonprf, chd,
            null;
buffers     MFR_BUFFER, 1-m, act, noninv, nonunq, nul, nonmut, nonprf, chd,
            null;
description varstring, 1-1, act, inv, nonunq, nul, nonmut, prf, ,
            null;
number ;
}

/*  Equipment in the factory */

MFR_MACHINE  {
name   string, 1-1, act, inv, nonunq, nonul, nonmut, prf,, ;
cell   MFR_CELL, 1-1, pas, inv, nonunq, nonul, nonmut, nonprf, par, ;
batch  integer, 1-1, act, noninv, nonunq, nul, nonmut, prf, , 1;
mttf   float, 1-1, act, noninv, nonunq, nul, nonmut, prf, , 1.0;
mttr   float, 1-1, act, noninv, nonunq, nul, nonmut, prf, , 0.0;
failuremodes MFR_FAILUREMODE , 1-m, act, noninv, nonunq, nul, nonmut, nonprf,
                   chd, null;
maintmodes MFR_MAINTMODE, 1-m, act, inv, nonunq, nul, nonmut, nonprf, chd, null;
status  string, 1-1, act, inv, nonunq, nonul, mut, prf, , "up";
detailed_status varstring, 1-1, act, inv, nonunq, nul, mut, prf, , null;
number ;
}

/* Storage area */

MFR_BUFFER  {
```

```
name    string, 1−1, act, inv, nonunq, nonul, nonmut, prf, ,null;
cell    MFR_CELL, 1−1, pas, inv, nonunq, nonul, nonmut, prf, par, null;
capacity   integer, 1−1, act, noninv, nonunq, nul, mut, prf, , 0;
lots       LOT, 1−m, act, noninv, nonunq, nul, mut, nonprf, chd, null;
number ;
}

/* Required maintenance operation */

MFR_MAINTMODE{
name           string, 1−1, act, inv, nonunq, nonul, nonmut, prf, , null;
maintmax         integer, 1−1, act, noninv, nonunq, nul, nonmut, nonprf, , 0;
description    varstring, 1−1, act, inv, nonunq, nul, nonmut, nonprf, ,
              null;
currentmaint   integer, 1−1, act, noninv, nonunq, nul, nonmut, prf, , 0;
}

/* Possible failure */

MFR_FAILUREMODE{
name        string, 1−1, act, inv, nonunq, nonul, nonmut, prf, , null;
mttf       float, 1−1, act, noninv, nonunq, nul, nonmut, prf, , 999.9;
mttr       float, 1−1, act, noninv, nonunq, nul, nonmut, prf, , 0.0;
description varstring, 1−1, act, inv, nonunq, nul, nonmut, nonprf, ,
           null;
last_failure TIME, 1−1, act, noninv, nonunq, nul, mut, nonprf, chd, null;
last_repair  TIME, 1−1, act, noninv, nonunq, nul, mut, nonprf, chd, null;
}

/* Staff that perform operations and maintenance */

MFR_TECHNICIAN{
name       string, 1−1, act, inv, nonunq, nonul, nonmut, prf, , null;
email      string, 1−1, act, inv, nonunq, nonul, nonmut, prf, , null;
machines   MFR_MACHINE, 1−m, act, noninv, nonunq, nonul, nonmut, nonprf, ,
           null;
```

```
present    boolean, 1-1, act, inv, nonunq, nonul, mut, prf, , 0;
busy       boolean, 1-1, act, inv, nonunq, nonul, mut, prf, , 0;
activeopinst  ACTIVEOPINST, 1-1, act, noninv, nonunq, nul, mut, nonprf, chd, null;
}

LOT{
nonmfr;
}

TIME{
nonmfr;
}

ACTIVEOPINST{
nonmfr;
}
end
```

Chapter 5

The MFR Textual Representation

The MFR textual representation provides a convenient way for describing a factory. The information specified via MFR, and its organization depends on the MFR schema that is being used. For a discussion of the MFR schema and its relationship with the MFR textual representation please refer to Chapters 2 and 4.

5.1 Syntax of the MFR Textual Representation

The symbols defined in Table 4.1 will be used in this chapter as well.

The syntax of the MFR textual representation is a C-like syntax, and it has a block structure. Each object, or group of objects of the same class, is specified in a block of the following form:

object-declaration-block : **class-name object-list** TERMINAL;

The **class-names** available are the keywords specified for classes in the schema representation. An **object-list** is a comma separated list of objects of the same class.

The **object-list** is defined next.

```
object-list : object
            | object COMMA object-list
            ;
```

object : IDENTIFIER { **attribute-list** } **specific-name-list** ;

The IDENTIFIER is a string without quotes. It is used by the MFR interpreter to keep track of the object definitions. Two object definitions of the same class can not have the same identifier. If the class in which the object defined belongs has an attribute of type string or varstring with keyword "name", and the class does not have an attribute "number", then the value of the "name" attribute will be the identifier of the object definition. If the class has an attribute "number" but its value is set to one (which is the default value), the value of the "name" attribute will again be the identifier of the object definition. If, on the other hand, the value of the "number" attribute is greater than one, several MFR objects will be created. The names for these objects can be either specified in the MFR textual representation as **specific-name-list**, or they will be generated by the interpreter from the identifier by adding a number suffix. If some of the those names are provided by the user, but they are not enough to name all the objects to be created, the interpreter will again use the identifier with a number suffix to name the rest.

Notice that an object definition in the MFR textual representation may result in the creation of several MFR objects.

The names of identical objects can be specified as a list between the closing bracket of the object and the COMMA or TERMINAL that follow.

```
specific-name-list : /* empty */
                   | specific-name specific-name-list
                   ;
```

specific-name : IDENTIFIER ;

Let us look at the **attribute-list**:

attribute-list : /* empty */

| **attribute attribute-list**

;

attribute : **attribute-key value-list** TERMINAL;

The set of valid **attribute-keys** is determined by the schema used. The value of the attributes can be specified as a comma separated list, since we need to allow "one to many" attributes.

value-list : **value**

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

;

value : **object** | INTEGER | IDENTIFIER | STRING | FLOAT ;

The **value** of an attribute can be an **object**, which means that objects can be defined in a nested fashion. Nesting can have arbitrary depth. Alternatively, the value can be the identifier of an object defined elsewhere. The interpreter is responsible for making sure the type of the value specified for an attribute is consistent with the type the attribute should have as specified by the schema.

The collection of **object-declaration-blocks** constitutes the MFR textual representation. The list must end with the keyword 'end'.

object-declaration-block-list : /* empty */

| **object-declaration-block object-declaration-block-list**

| **end-of-file**

;

If the same information about the hierarchical structure of the objects can be specified in two attributes, the user can specify this in any one of those two or both, since the interpreter will perform consistency checks. So for example consider a schema where machines are children of cells and an attribute of class CELL specifies the machines that belong to a cell but also an attribute of class MACHINE specifies the cell in which a machine belongs. The user can specify the cell in which a machine belongs in the attribute of MACHINE or in the attribute of CELL or in both places as long as there is consistency. It does not make any difference which of those attributes is the active one in the data base.

5.2 An Example of the MFR Textual Representation

An example of the MFR textual representation with the schema defined in section 4.3 follows. Several features of the MFR interpreter are illustrated through this example. The top level cell which represents the whole factory is ICL. It is decomposed in subcells that are leaf cells. That way, a two level hierarchy is implemented as shown in Figure 5-1.

ICL has five subcells; three of them are DIFFUSION and two are PHOTO. Cells DIFFUSION are specified within the same **object-declaration-block** as ICL is. The number is three but only one name is provided, bay1. So DIFFUSION cells will have names bay1, DIFFUSION1 and DIFFUSION2. The two PHOTO cells are specified in a nested fashion when MFR_MACHINE GCA is specified. There names are specified to be photobay1 and photobay2. Some buffers of cell ICL are declared in a nested fashion, and some are declared in the last **object-declaration-block**. Three machines tube (tube1, tube2 and tube3) are specified within the cell DIFFUSION. Since there are three DIFFUSION cells, nine MACHINE objects will be created, three for each DIFFUSION cell. Next, two machines, GCA1 and GCA2 are specified. Some MFR_FAILUREMODEs and MFR_MAINTMODEs are declared in a nested fashion. The next three **object-declaration-blocks** are for an MFR_FAILUREMODE object, an MFR_MAINTMODE object and an MFR_TECHNICIAN object.

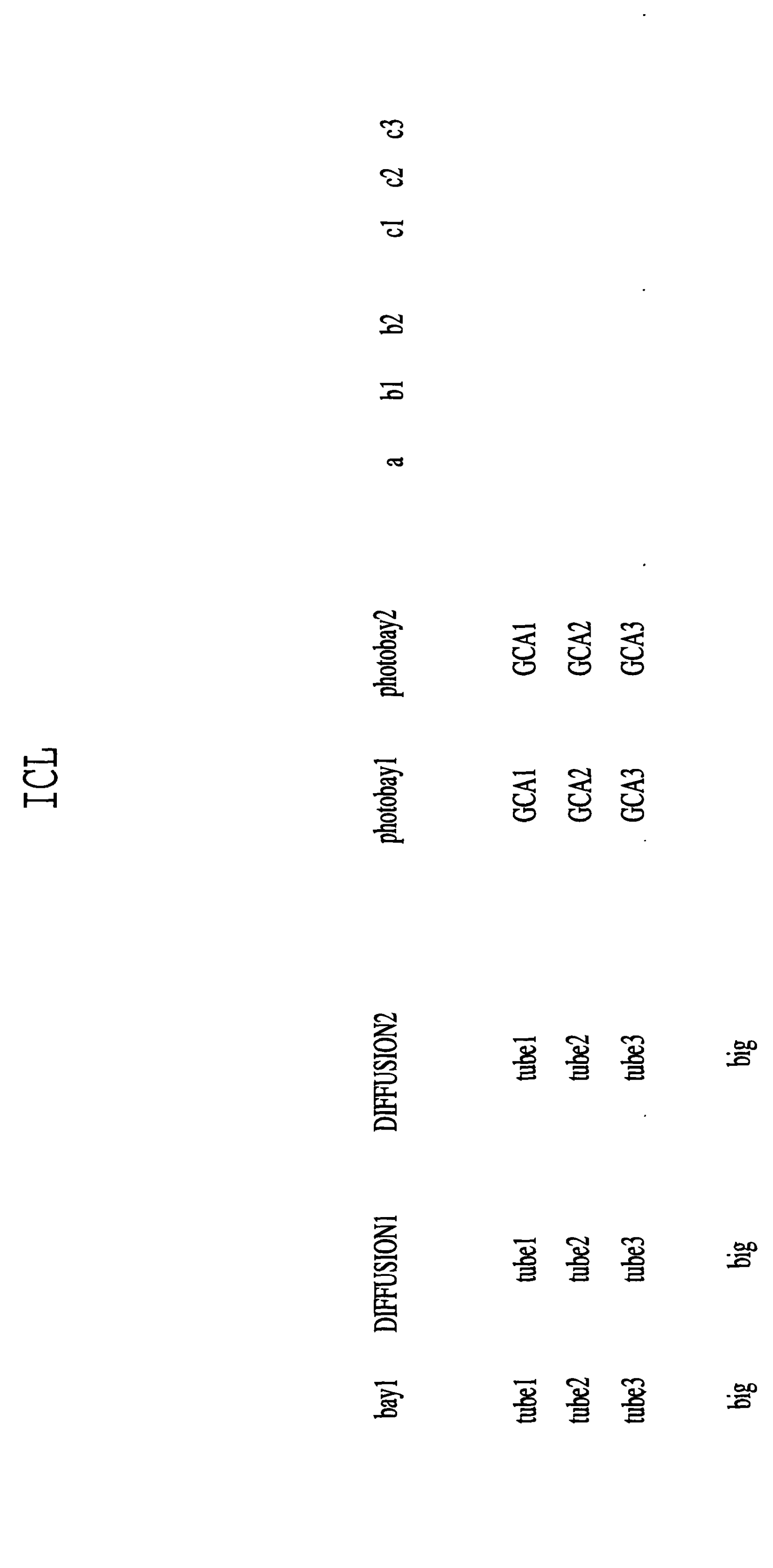

Figure 5-1: Pictorial representation of the factory defined in MFR textual form.

```
/* This is an example of an MFR textual representation */

MFR_CELL ICL{
name "icl";
subcells DIFFUSION, PHOTO;
buffers a{}, b{number 2;}, c, d{number 3;};
},

DIFFUSION{
number 3;
machines tube{mttf 3.3; number 3;};
}bay1;

MFR_MACHINE GCA{
cell PHOTO{number 2;} photobay1 photobay2;
failuremodes lamp{mttf 34.2; description "exploding lamp...";}, computer;
maintmodes focus{maintmax 3;}, lamp;
number 2;
};

MFR_FAILUREMODE computer{
description "computer failure can cause machine GCA to be down";
mttf 5.5;
mttr 0.4;
};

MFR_MAINTMODE lamp{
maintmax 56;
description "I have no idea how to maintain a lamp!";
};

MFR_TECHNICIAN me{
machines rca{cell DIFFUSION;}, GCA;
email "mstamato";
};

MFR_BUFFER big{
cell DIFFUSION;
},

c{
capacity 100;
};

end
```

Chapter 6

Programmer's Manual For the Schema and MFR Interpreters

This chapter is a guide to programmers who may want to modify an MFR textual representation interpreter created by the MFR schema interpreter, or for programmers who want to modify the MFR schema interpreter itself. One reason for modifying the MFR textual representation interpreter may be to use the factory data without storing it in the data base. Only minor changes in the main function are required for this.

6.1 The Schema Interpreter Architecture

Please refer to Figure 6-1 for an overview of the schema interpreter architecture.

The schema interpreter first parses the schema textual representation and stores all the information provided in computer memory. Then, it performs some consistency checks, prints a representation of the schema it just read, and finally proceeds to create the source code for the MFR interpreter.

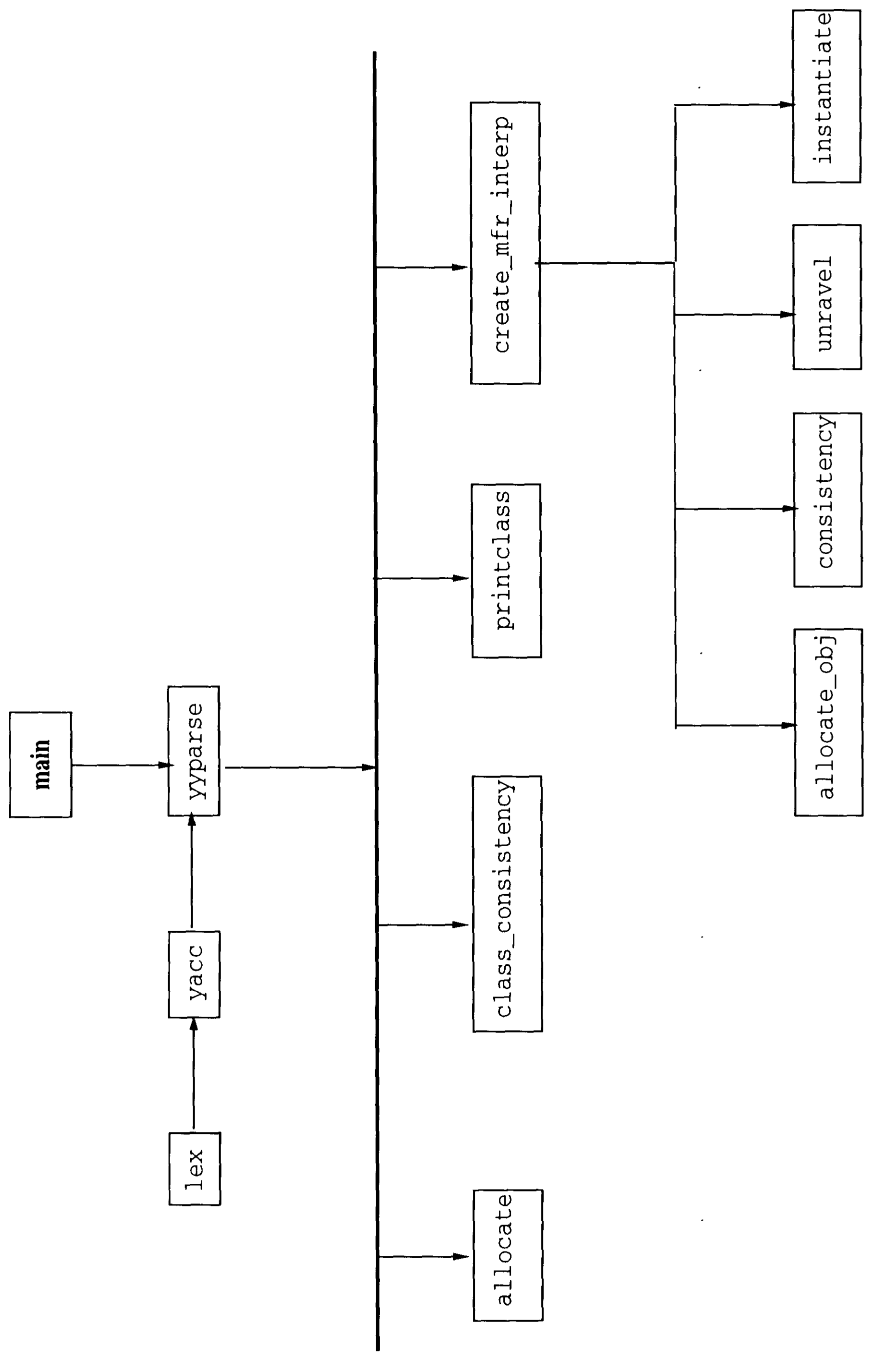

Figure 6-1: Architecture of the MFR schema interpreter.

6.1.1 The `main` function

The **`main`** function is responsible for reading the command line with which the schema interpreter is run, for running the input file through the C preprocessor cpp, for calling the function initialize, and finally for calling the parser. If the command line is not correct, the format of the command line will be echoed to the user.

All the files required will be opened by the main function. The default name for the output file, if it is not specified by the user is *out*. The input file is run through the C preprocessor cpp in order to strip the comments and to use the **`#define`** macro. The output of the preprocessor is stored in a **`<input.int>`** file. The old version of this file, if any, is removed. For all the above, the **`system`** function is used.

The parser function created by yacc is called *yyparse*. It reads the input from the external file **`yyin`**. For this reason the .int file is opened as yyin. Also, for the same reason, the output file is opened as an external file **`yyout`**.

The **initialize** function initializes the **`linenumber`** variable to 1, the **`error`** flag to 0, and the **`class_list_flag`** to 0. The **`error`** counts how many consistency errors are detected by the **class_consistency**. In this way the program does not need to exit when the first error is detected, but it can check all the classes, give appropriate error messages for all the errors, and then exit it **`error`** is greater than 0. The **`class_list_flag`** is used by the function that allocates memory in order to store information relevant to a class, **allocate_class**. The first time a class is allocated, the **`head_class`** is set to point to that class, and **`class_list_flag`** is set to 1.

6.1.2 The parser

The lex lexical analysis program generator and the yacc parsing program generator were used to create a lexical analyzer and a parser. In the lex file *clex* the tokens used by the parser are specified and the definitions of a string, a name, an integer and a float are given. In other words, the lexical analyzer returns a prespecified value for every token it recognizes so that the parser knows what was just read. Another job the lexical analyzer does, is that it keeps track of the line number of the input file it is reading. This is particularly helpful, because when syntax errors are detected by the parser, the line number of the syntax error is given as part of the error message. The way this is accomplished is by introducing an integer variable **`linenumber`**, initialize it to 1, and have the lexical analyzer increase it by 1 every time the new line character is recognized.

In the yacc file *cyacc* the syntax of the schema interpreted is specified, the information given by the schema textual representation is stored, and all the other functions are called. This can be done because yacc allows the execution of C commands when patterns are recognized. For example, when a *class-definition* pattern is recognized, the function that allocates memory to store all the information relevant to this class is called, and the NAME as it was read by the lexical analyzer is stored as the keyword for the particular class. Then, when an *attribute-definition* pattern is recognized, memory to store the information relevant to this attribute is allocated, the NAME is stored as the attribute keyword, and then all the values of the slots are stored. The values of the slots are stored as strings, in a way that they will be understood easily. So for example, if the attribute is one to one, "1-1" will be written in the '**`class->attribute->single`**' string. If, on the other hand, the attribute is one to many, "1-m" will be the value of **`class->attribute->single`**. More on the data structures will be discussed later in this section.

If a syntax error is detected, an error message that includes the current line number is sent to standard error.

When the token **end** has been recognized by the lexical analyzer, and *end-of-file* is returned, the functions that will perform the interpreter tasks will be called. Those functions are: **`class_consistency`**, **`printclass`** and **`create_mfr_int`**. They will be discussed later in this chapter.

6.1.3 The function `class_consistency`

The **`class_consistency`** function makes sure none of the rules specified in the schema are violated. The rules that this function checks are:

1. Attributes that are not of some object type can not be child or parent attributes. The parent vs children slot only applies to attributes that are of some object type.

2. Parent attributes can not be one-to-many. Objects can only have a single parent.

3. Attributes that are of some object type must have a valid one. A valid object type can be either an MFR class defined within the same schema, or a non MFR class specified in the schema textual representation.

4. At most one class can have children or parent attributes of the same type. This means that only one class can form the MFR hierarchical structure.

5. A class that has hierarchical structure must have both a children and a parent attribute of the same type as the class. This is an assumption that was made early in the design process of the interpreter.

6. All MFR classes must have at least one attribute. It would not make sense to have a class that does not have any attributes.

7. If the number attribute exists, a singleton attribute with keyword **name** and of type **string** or **varstring** must also exist. This attribute will be used to distinguish among the identical objects that may be created by the MFR textual representation interpreter.

The **class_consistency** function is also responsible for creating an attribute **number** and assigning the proper values in its slots. This will be done for the classes that it is appropriate.

Another task this function performs is to find the attributes that form the hierarchical structure, if any. If a class that has attributes of the same type as the class exists, the children attribute and the parent attribute that have the same class are found. Then global variables **lattr** and **hattr** of type attribute pointers, are set to point to those attributes. This is done because several functions need to use these pointers.

The source code for this function is in file *class_consistency.c*.

6.1.4 The function printclass

This function prints to the output file a representation of the schema after the checks of **class_consistency** are done. The source code for this function is in file **printclass.c** and is self-explanatory. The output of the example of Section 4.3 can be found in Appendix A.

6.1.5 The function create_mfr_int

From this routine, all the functions that actually create the MFR interpreter are called. Each one of those functions is responsible for creating a single file, so the source code of the MFR interpreter consists of the same files regardless of the schema. They are:

- **ilex** This function creates a **lex** file that can make the lexical analyzer necessary for the MFR interpreter.
- **iyacc** This function creates the **yacc** file that can make the parser necessary for the MFR interpreter.
- **interp** This function creates the header file for the MFR interpreter.
- **imain** This function creates the main function of the MFR interpreter.
- **def** This function will create the function that will provide the default values for the attributes of objects.
- **printmfr** This function will create the function that will print the MFR objects in the output file.
- **read_attr** The function responsible for reading attributes values , and making sure they are valid is created.
- **store** The Function that will actually store the values of attributes is created. This function is also responsible for dynamically allocating memory in order to store attributes that are not singleton.

- `imalloc` The routines that allocate memory for the structs as defined in the header file is created.
- `alloc` The function that will dynamically allocate memory for the new MFR objects, by calling the appropriate routines, is created.
- `utilities` Routines that handle stacks and strip strings from their quotes are created.
- `nest` Functions that can handle objects when defined in a nested fashion is created. The nesting can have arbitrary depth.
- `insert` Routines that insert MFR objects at an arbitrary location of the corresponding linked lists is created.
- `def_names` A function that provides default names to objects with number greater than one whose specific names are not given is created.
- `consistency` A function that performs various consistency checks among objects is created. This function will also create pointers pointing to the parent and children objects, if any.
- `unravel` This function creates the functions that will replicate identical objects. They will also set pointers to parent and children for the new objects.
- `replicate` Functions that are called while unraveling are created. They are responsible for replicating children objects, if their parents are being unraveled.
- `subobject_list` The function responsible for finding children of the objects of the class type that has hierarchical structure is created.
- `instantiate` If the schema currently used has been installed in the data base, this function will create the functions that will write to the data base. If the schema has not been installed, it will create a function that will print an error message if the user tries to write to the data base.

A more extensive discussion of the above functions can be found in Section 6.3.

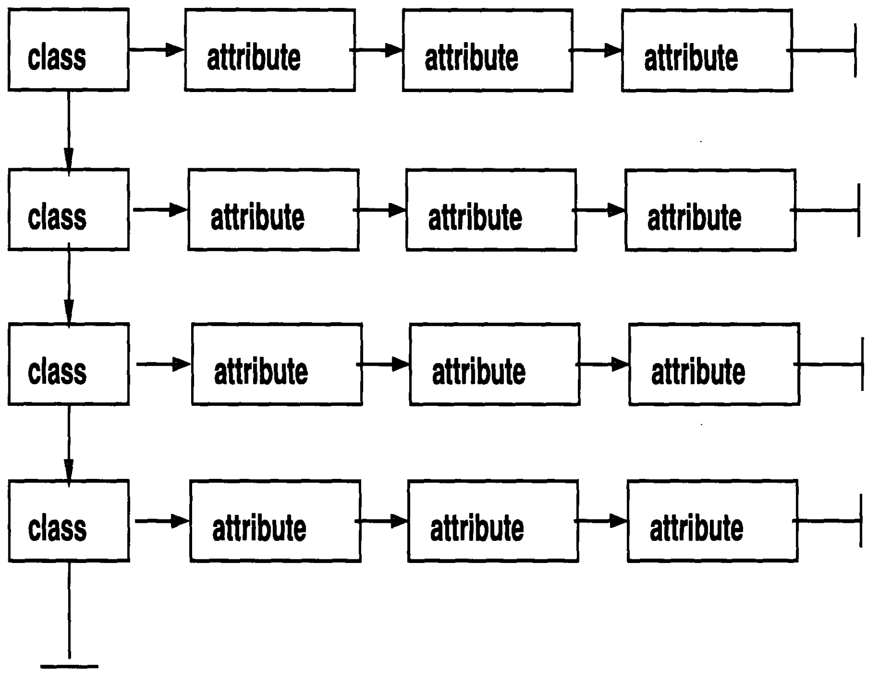

Figure 6-2: Data structure for the MFR schema interpreter.

6.2 The MFR Schema Representation Interpreter Data Structure

The MFR schema interpreter needs to store all the information relevant to the classes specified in the schema representation. This is accomplished with two `structs` that are related as shown in Figure 6-2.

The `class struct` forms a linked list of classes. There, the keyword of the class is stored as a string (`dbkey`); various flags that indicate, among other things, whether the class has a number attribute and whether it is an MFR class are present; and the pointer of the head of the linked list of attributes that correspond to the class is stored.

```
typedef struct class_struct *class_ptr_type;
struct class_struct {
   char dbkey[KEYLENGTH];
   char key[KEYLENGTH]; /* same as dbkey with the addition of an "s" at the end.
         This became necessary in order to avoid naming conflicts with data base types
         that occurred during the writing of instantiate.c */
   int attr_list_flag;
   int number, id, nonmfr;
   int unravel;
   attr_ptr_type attr, head_attr;
   class_ptr_type next;
   int hierarchy_flag;
};
```

The attribute struct forms a linked list of attributes. There, the keyword of the attribute is stored in dbtype. All the other information about the attribute provided through the schema representation is also stored within this struct.

```
typedef struct attr_struct *attr_ptr_type;
struct attr_struct {
   char type[KEYLENGTH], single[10], act[10], inv[10], unq[10];
   char nul[10], mut[10], prf[10], hie[10], key[KEYLENGTH];
   char def_type[10];
   int id, var, bool;
   char dbtype[KEYLENGTH];
   union dfl_value dfl;
   attr_ptr_type next;
};
```

6.3 The MFR Textual Representation Interpreter Architecture

Figure 6-3 is an overview of the MFR textual representation interpreter. A description of the files that make up the source code of this interpreter follows. A moderate effort was make to have good indentation on those files. Since the MFR textual representation interpreter depends on the schema that is being used, the following discussion refers to the way the schema interpreter creates the MFR textual representation interpreter.

6.3.1 The `main` function in *main.c*

The `main` function is responsible for reading the command line that calls the interpreter, for opening the necessary files, for running the input file through the C preprocessor cpp, for calling the `initialize` function (also in *main.c*) and finally, for calling the `yyparse` parser.

The `getopt` function provided by C is used to read command options. Then the number of arguments is checked, and an error message is printed if it is not correct. The input file, the file where the output of the cpp preprocessor (<infile.int>) will be stored and the output file are opened. Then, with the aid of the `system` function provided by C, the cpp preprocessor is called and its output will be the `yyin` file used by `yyparse`. In the function `initialize`, various flags are set. Those include flags for every class, that indicate that no memory allocation has yet been done for each particular class. This information is necessary for the `allocate_obj` function that needs to set the head of the object linked lists for every class.

In order to use the interpreter from another program, and use the C data structures it creates, one can not use the same `main` function. Instead, the following tasks must be done. The textual representation file must be run through the cpp preprocessor, the files `yyin` and `yyout` must be opened for the `yyparse` function to use and finally the same `initialize` and `yyparse` functions must be called. An example where this

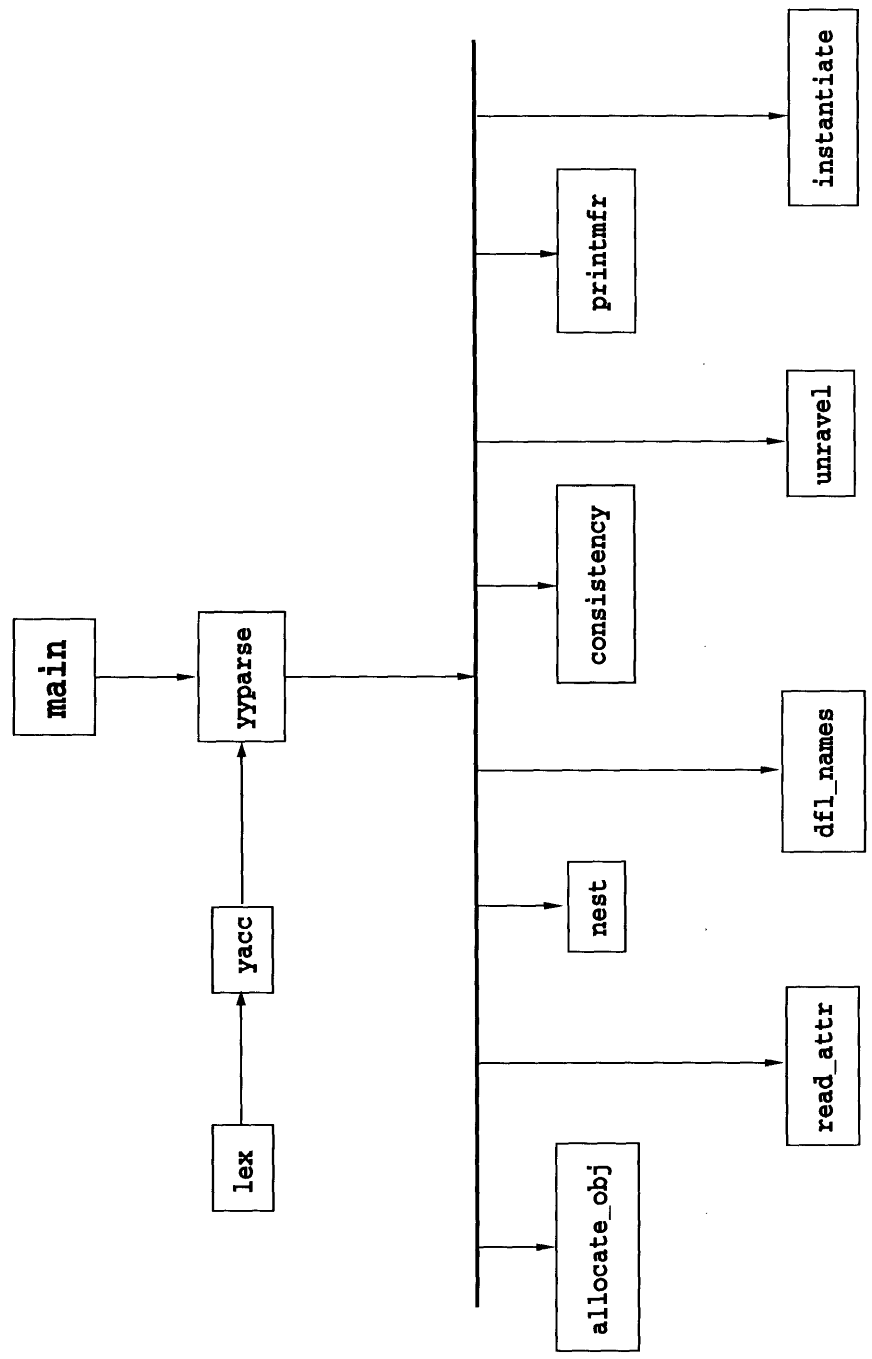

Figure 6-3: Architecture of the MFR textual representation interpreter.

was done is described in Chapter 7.

6.3.2 The lexical analyzer in file *clex*

A file *clex* is created by the schema interpreter. This file is written in the language understood by the lex lexical analysis program generator. There, all the keywords for the classes and for the attributes are being specified. Also, the definition of what constitutes a string, a float, an integer and an identifier are provided. Furthermore, blank spaces are ignored, the counter `linenumber` is advanced whenever the new-line character is read, and the "," the ";" the "{" and the "}" are recognized. For more information on lex, please refer to [3].

6.3.3 The parser in file *cyacc*

A file *cyacc*, written in the language understood by the yacc parsing program generator is created by the schema interpreter. There, the definition of the syntax of the MFR textual representation is provided. All the major functions of the MFR textual representation interpreter are called from within the parser, as patterns are recognized. For more information on yacc, please refer to [3].

When the keyword of a class is recognized, the global variable of type integer `obj` is set to a value that represents that class. This information is necessary for several functions that need to know which class they are dealing with. Also, when the pattern of a new object is recognized, the following actions are taken. The identifier of the object is stored, and if the object has been defined in a nested fashion (at any depth), the function `nest` is called that allocates memory for the object via the `insert` function, otherwise the function `allocate_obj` is called. The reason a distinction between nested and not nested objects is necessary is discussed in Section 6.3.8. The global variable `nest` keeps track of whether the object that is being read is a nested one.

As attribute keywords are recognized, the global variable `atr` is set to a value that corresponds to the attribute being read. If attributes with the same keyword exist in different classes, they will be given the same value. Also, as values are read, the global variable **val** will keep track of the type of the variable, i.e., whether it is an integer, a float a string or an identifier (string without quotes). Then, the functions **read_attr** and **store_attr** are called. The former will make sure that the type of the attributes value is valid and give an error message otherwise, while the latter will store that value.

If a syntax error occurs, the line-number is printed. When the keyword **end** is found, and if **error** is zero, the parser calls the following functions:

- **default_names** This function is responsible for providing default names to objects whose number attribute is greater than 1.
- **consistency** This function performs various consistency checks and sets pointers to children and parent objects.
- **unravel** and **subobject_list** These functions unravel objects whose number attributes are greater than one.
- **printmfr** This function will create a file with a representation of the factory that was just read.
- **instantiate** This function will be called if the interpreter is run with the **-s** option, and it will create data base objects that will represent the factory described in the textual representation.

6.3.4 The files *read_attr.c* and *store.c*

A single function, **read_attr**, is in file *read_attr.c.* This function is responsible for making sure that the type of value read as an attribute value is the type expected. First, the variable **atr** is checked, and then, if the variable **val** does not indicate the appropriate type, an error message is send to standard error, and the variable **error** is incremented by 1.

In file *store.c* the functions that store the values for the object attributes in computer memory exist. There are as many functions as classes plus one. The function

store_attr will determine which function to call, depending on the class in which the object being read belongs to. Then, the specialized functions for every class will determine the attribute that was just read, and store its value. If the attribute is **1-m** memory space will be allocated for the new attribute just read.

6.3.5 The files *allocate.c* and *insert.c*

There is a linked list of objects that corresponds to every class of the schema. A discussion on the data structure can be found in Section 6.4. The files *allocate.c* and *insert.c* include the functions that create the linked lists of all the classes. In both files, there are as many functions as classes in the schema. The functions in *allocate.c* assume that when they are called, the global pointer corresponding to the object of the class is the last object in the linked list, and they will put the new object as the next (and now last) in the list. On the other hand, the functions in *insert.c* make no assumption on the pointer of the current object, and they will put the new object before the current one. If the current one is the head of the class, the new object will become the head.

6.3.6 The file *malloc.c*

In file *malloc.c* all the memory allocation routines are located. Memory allocation routines are needed for the structs that correspond to every class, for the structs that correspond to attributes that are **1-m** and for stacks needed in the **nest** function. In all those routines, error messages are given in the case the computer runs out of memory.

6.3.7 The file *default.c*

The functions that provide default values for the attributes of objects are in this file. There are as many functions as classes, and the default values are as specified in the schema representation. These functions are called after memory is allocated for new objects from the functions in files *allocate.c* and *insert.c*.

6.3.8 The file *nest.c*

In this file the functions **nest** and **unest** are located. When an object specified in a nested fashion is recognized, the variable `obj` must be updated, but the type of the parent object must be stored. Since nesting can have arbitrary depth, a stack of values for the `obj` variable must be kept. When a new nested object is introduced, the current value of `obj` is pushed into the stack (in function **nest**), and when the definition of a nested object ends, a value is popped off the stack (in function **unest**). A stack is needed to keep the value of the variable **atr** as well. This is necessary because as the new object is stored and attributes are read, the value of **atr** will be updated.

In **nest**, the type of the new object is determined by finding the type of the parent object and the attribute through which the new object is being defined. Also, in this function, memory is allocated for the new object by calling functions in the *insert.c* file. The identifier and the pointer of the new object are kept in the struct corresponding to the parent object. In **unest**, the pointer of the class for the object that was nested is returned to the last object in the linked list.

6.3.9 The file *utilities.c*

In this file, several functions needed for the `obj` and **atr** stacks mentioned in Section 6.3.8 exist. Also, the function **strip_quotes** is there. This function is called every time a string is read, and its job is to delete the quotes on either side of the string.

6.3.10 The file *def_names.c*

In this file, there are as many functions as classes with a number attribute, plus the function **default_names** that calls all the rest. Each function loops through all the objects of the class it is responsible for, and gives default names to those that need them. The structs that correspond to classes with a number attribute include a linked list of specific names. The ones provided within the MFR textual representation are already stored there as they are read. As long as the number attribute indicates more identical objects than specific names, a new name is added to the list of specific names. The new names are the identifier of the object with a number suffix. A counter keeps track of the number suffix.

6.3.11 The file *consistency.c*

In this file, there are as many functions as classes, plus the function `consistency` that calls the rest. Special attention is required for the function that corresponds to the class (if any) with which the hierarchical structure of the MFR representation is achieved. This class can be recognized by the fact that it has children of the same object type. As pointed in Section 6.1.3 , there can be at most one such class.

The schema interpreter uses the following algorithm to create this file:

- **go through all classes**
  - **if the class is the hierarchical class**
    1. **write the code that finds the parent and children of the same (hierarchical) class and looks for consistency errors regarding those objects.**
    2. **write the code that finds children objects that are not of the same (hierarchical) class.**
  - **if the class is not the hierarchical class**
    1. **write the code that finds parent (not if they are of the hierarchical class type) and children objects.**

All functions loop through all the objects of the class they correspond. The consistency function that corresponds to the class with the hierarchical structure is called last. This function first finds the parent object of the same class. If the parent is not indicated as an attribute of the object, it loops through all the other objects of the same type and checks whether the object under consideration is a child of them. If it is, then the pointer and the identifier of the parent are stored. If it is a child in more than one object, an error message is printed in the output file, and `error` is incremented. If no parent is found, the object is assumed to be the topmost object in the hierarchy. When more than one object without a parent is found, an error message is printed in the output file, and **error** is incremented. If the parent object is indicated, the program checks for its existence, it stores its pointer, and it also checks that the object in question is not a child of yet another object.

All functions, including the one that corresponds to the hierarchical class, perform the following tasks. First, they find the parents of every object. If the parent is not indicated as an attribute of the object in question, it is found by looping through all the objects of the class in which the parent belongs. If the parent is found, its pointer and identifier are stored, otherwise, an error message is printed and **error** is incremented. Consideration is also made for the case a consistency error in the textual representation was made and multiple parents are found. Also, all attributes of some object type are checked. The objects they refer to are checked for existence and their pointers are stored.

6.3.12 The files *unravel.c*, *replicate.c* and *subobject_list.c*

These files include the functions responsible for creating multiple identical objects where this is appropriate.

In *unravel.c*, there are as many functions as classes with a number attribute, plus the function **unravel** that calls all the rest. The schema interpreter loops through all classes and writes the necessary code for all classes with the number attribute. The function that corresponds to the class (if any) responsible for the hierarchical structure of the data organization is different than the rest. This function loops through all objects of the hierarchical class, for every object it calculates the number of identical objects required and then creates those identical objects. The number of identical objects required due to its ancestors is found by climbing the hierarchical tree and multiplying the value of the **number** attribute of the parent of the object in question by the value of the same attribute of its parents parent, etc, until the topmost object is reached. Then new objects are created by allocating memory with the appropriate insert function according to the value of the **number** attribute, and copying all the information of the original object. For each of the new objects, a number of identical objects depending on the number of ancestors is created by calling the hierarchy replicate function. A complication occurs for attributes of some object type that are children because those objects can not simply be copied, but they must be replicated. This is done by calling the replicate functions described later in this subsection. It is assumed that when the unravel function corresponding to the hierarchy class is called, all the other objects have been unraveled according to their number attribute. This is ensured by calling this function last.

The unravel functions corresponding to non hierarchy classes with a number attribute are responsible for creating multiple objects according to the value of the **number** of each object. Also, they are responsible for updating the list of children of the parent of an object that is unraveled in this way. The new objects are created with the appropriate insert function and by copying all the information included in the original object. If children objects exist, they will be replicated by using a replicate function described later in this subsection.

In *replicate.c* the hierarchy replicate function and functions that replicate children of other objects exist. The hierarchy replicate function has an integer argument that tells how many new objects to create. Then it calls an insert function and copies all attributes. For attributes of some object type, other replicate functions are called. The other functions also have one argument, which is a pointer to the head of the list of objects they are to replicate. The type of this pointer is the same one as the pointer of the struct that the corresponding class uses to store the list of objects of the type this function will replicate. As the functions replicate the whole list, they create another list of the same type that maps the original list. Finally, they return a pointer to the head of this list.

File *subobject_list.c* consists of two functions responsible for the tree structure of the hierarchy class after unraveling has been done, and of functions responsible for finding multiple identical objects for attributes of classes that do not have a number attribute. In the process of unraveling new objects are created, and so the list of children for some objects is now larger. These lists of children for objects that belong to the hierarchical class, are updated by function `subobject_list`. This function first calls the function `identical_hobjs_sublist` which sets the parent of objects whose parent can be any one of multiple identical objects. A flag `found_super` that facilitates this process is included in the struct corresponding to the hierarchical class. Then, it loops through all objects of the hierarchical class, and updates the list of children.

Classes that do not have a number attribute, and are therefore not unraveled, may have attributes of some object type that need to include newly created identical objects. In file *subobject_list.c* functions that take care of this exist. There, `1-m` attributes of some object type are checked. For every object in the list of objects corresponding to such an attribute, a search for identical objects among the class the attribute refers to is done, and the ones found are added to the list. These functions are called from `subobject_list`.

6.3.13 The file *printmfr.c*

In this file, a function for every class and function `printmfr` that calls them are included. Each function prints all the objects of the class they correspond including the values of their attributes. The address where the objects are stored in computer memory are printed next to the their identifiers. An example of the output file can be found in Appendix A.2.

6.3.14 The file *instantiate.c*

This file consists of the functions that will instantiate objects in the data base. The objects that are part of the hierarchical structure, and their children are created in the data base recursively starting from the topmost one. In this way objects that belong to the hierarchical class, and objects that are children of those are instantiated. For example, for the schema described in Section 4.3, the objects of classes CELL, MACHINE, BUFFER, MAINTMODE and FAILUREMODE are created recursively, but the objects of class TECHNICIAN are not. The function `instantiate` calls the one that will start the recursion with the pointer to the topmost object as the argument. Then, it calls the functions that will create the objects that are not part of the hierarchical structure. If those objects have children though, they will be created before their parents.

The function that will start the recursion returns a pointer to an object that belongs to the hierarchical class. Then it calls functions (in the same file) that create the children of the object currently being instantiated. One of these functions is the function that creates children of the same class. This function calls the first one in order to create each object. That way the recursion starts. The functions that create the children return gestalt [7] type LISTs of them. These lists are necessary to instantiate the object currently under consideration. In order to make objects in the CAFE data base gestalt functions are called.

6.4 The MFR Textual Representation Interpreter Data Structure

The data structures of the MFR textual representation interpreter are constructed in the header file *interp.h*. In this file, the global variables used and the structs for storing the MFR objects in computer memory are declared. First, the structs for the linked lists of attributes with arbitrary number of values (`1-m`) are declared. If these attributes are of some object type, they include the identifier and a pointer to the object they refer to. Otherwise, a variable whose type depends on the type of the attribute, is included in the struct. Next, the structs where the names of multiple identical objects for classes that have a number attribute are declared. The struct definitions for the MFR objects follow. For each class, a struct that points to the next and previews objects, is declared. This way, a doubly linked list of objects for every class is formed. An overview of the data structure for the objects of any class is illustrated in Figure 6-4. In those structs, variables for the values of all the attributes that are not of some object type and pointers for attributes of a single object type are included. Also, pointers to the heads of the linked lists of `1-m` attributes and flags

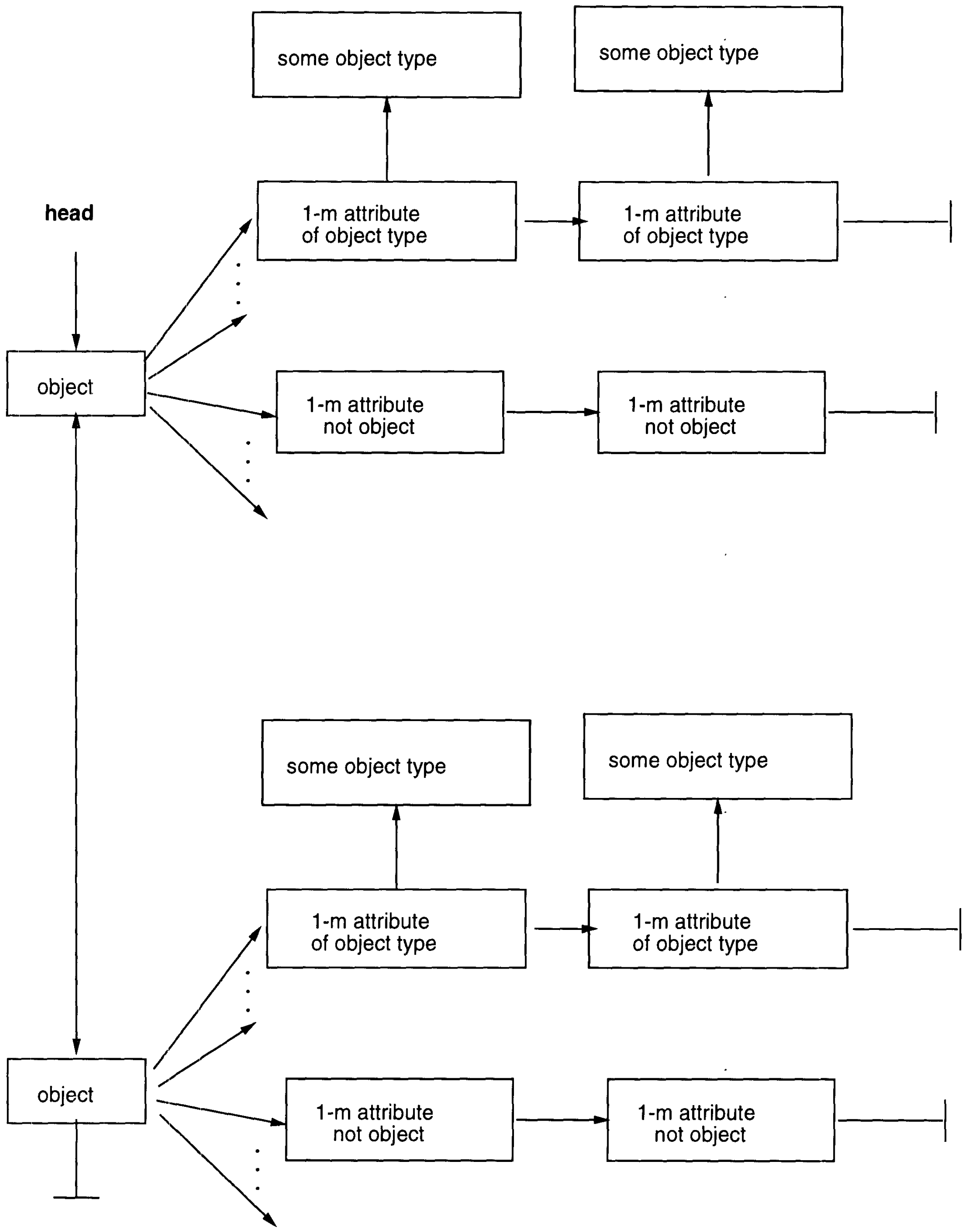

Figure 6-4: Data structure for the MFR objects.

that tell whether the head of those lists have been set are part of the structs.

In the same header file *interp.h*, the structs necessary for the stacks of `obj` and **`atr`** used by **`nest`** can also be found. Then, global variables where attribute values and object identifiers will temporarily be stored as they are read, are declared. These variables will be used by the parser to get values from the lexical analyzer. There is one such variable for every type, i.e. one for strings, one for integers etc. Also various flags that tell whether the first object of each class has been allocated, the number of consistency errors found so far, the line-number currently being read and more can be found in this file. Finally, pointers to the heads of the class doubly linked lists, and functions that return pointers are declared.

Chapter 7

Conclusions

7.1 Summary

In this thesis, a language/representation with which the resources pertaining to production systems can be described is introduced. This language is the Microsystems Factory Representation and it consists of the organization of static data, the static data itself, and the organization of dynamic data about factories. Thus, MFR defines the pieces of information, static and dynamic, that are important and the way they are related. This is the schema with which factories will be described. MFR also provides the static data about factories. An Example of static data is information about the characteristics of machines in the factory, as opposed to dynamic data which might be their status.

Once a factory schema has been specified, a textual representation of a factory, where the data about the factory are provided is available. The syntax of this representation depends of the schema, but it always has the same characteristics. An MFR textual representation interpreter parses the factory description, runs various consistency checks, and makes the data easily accessible. This data can be used either directly from the C data structures the interpreter provides, or from the CAFE data base [8] where the data can be stored. A program that creates MFR textual representation interpreters for any particular schema has been developed.

The MFR, along with the CAFE data base, can guarantee the consistency of factory data used by various applications. With the aid of MFR, we can have simulations that use the same factory data and the same schedulers as in real time control of a fab. That way simulations are more realistic, and schedulers can be tested extensively and debugged before they are put in use. Furthermore, MFR can be a very useful tool in the design of factories, since various checks and performance measures can be calculated by data provided via MFR.

7.2 Suggestions for Future Work

The question of what information is important about a factory has been left for the MFR user to answer. The way this data is structured though, may emphasize some aspects of production systems. For example, by organizing the factory data in a hierarchical fashion, we indicate that a factory can be decomposed and studied in pieces.

A potentially useful feature that may be added to MFR is the ability to specify functions that relate various attributes. This way, the value of some attribute may depend on other values. For example, the relationship of the efficiency of a machine as a function of its mean time to fail and mean time to repair (Section 3.2.2) could be part of the MFR schema. Then, two of the values of those attributes would be necessary, and the third would be calculated by the MFR interpreter. Another way this may be useful, is in the case of specific *mttfs* (mean time to failure) for different failure modes, and a computed summary *mttf* for each machine.

In Chapters 1 and 3, various applications that require MFR data are discussed. Those applications include programs that perform production feasibility checks, schedulers and others. These programs need to be developed such that they make full use of MFR and benefit from doing so. By using MFR, they provide their users an easy way of specifying and modifying factory data, and therefore facilitate experiments. Also, if the schema installed in the CAFE data base is used, they can provide the option of reading data from it.

Appendix A

Output Files

A.1 Output From the Schema Interpreter

The output from the schema interpreter for the example of section 4.3 is the following:

```
class: MFR_CELL
name    1-1  act  inv  nonunq  nonul  nonmut  prf  empty  -
supercell  MFR_CELL  1-1  pas  inv  nonunq  nul  nonmut  prf
           par  null
subcells  MFR_CELL  1-m  act  noninv  nonunq  nul  nonmut
          nonprf  chd  null
machines  MFR_MACHINE  1-m  act  noninv  nonunq  nonul  nonmut
          nonprf  chd  null
buffers  MFR_BUFFER  1-m  act  noninv  nonunq  nul  nonmut
         nonprf  chd  null
description    1-1  act  inv  nonunq  nul  nonmut  nonprf
               empty  null
number    1-1  act  inv  nonunq  nonul  nonmut  prf  empty  1
number yes
```

```
------------------------------------------------------------------------------
class: MFR_MACHINE
name    1-1  act  inv  nonunq  nonul  nonmut  prf  empty  -
cell  MFR_CELL  1-1  pas  inv  nonunq  nonul  nonmut  nonprf
      par  -
batch    1-1  act  noninv  nonunq  nul  nonmut  prf  empty  1
mttf    1-1  act  noninv  nonunq  nul  nonmut  prf  empty  1.000000
mttr    1-1  act  noninv  nonunq  nul  nonmut  prf  empty  0.000000
failuremodes  MFR_FAILUREMODE  1-m  act  noninv  nonunq  nul  nonmut
              nonprf  chd  null
maintmodes  MFR_MAINTMODE  1-m  act  inv  nonunq  nul  nonmut  nonprf
            chd  null
status    1-1  act  inv  nonunq  nonul  mut  prf  empty  up
detailed_status    1-1  act  inv  nonunq  nul  mut  prf  empty  null
number    1-1  act  inv  nonunq  nonul  nonmut  prf  empty  1
number yes
------------------------------------------------------------------------------
class: MFR_BUFFER
name    1-1  act  inv  nonunq  nonul  nonmut  prf  empty  null
cell  MFR_CELL  1-1  pas  inv  nonunq  nonul  nonmut  prf  par
      null
capacity    1-1  act  noninv  nonunq  nul  mut  nonprf  empty  0
lots  LOT  1-m  act  noninv  nonunq  nul  mut  nonprf  empty
      null
number    1-1  act  inv  nonunq  nonul  nonmut  prf
          empty  1
number yes
------------------------------------------------------------------------------
class: MFR_MAINTMODE
name    1-1  act  inv  nonunq  nonul  nonmut  prf  empty
```

```
        null
maintmax    1-1  act  noninv  nonunq  nul  nonmut  nonprf
            empty  0
description    1-1  act  inv  nonunq  nul  nonmut  nonprf
               empty  null
currentmaint    1-1  act  noninv  nonunq  nul  nonmut  nonprf
                empty  0
--------------------------------------------------------------------------------
class: MFR_FAILUREMODE
name    1-1  act  inv  nonunq  nonul  nonmut  prf  empty  null
mttf    1-1  act  noninv  nonunq  nul  nonmut  nonprf  empty  999.9000
mttr    1-1  act  noninv  nonunq  nul  nonmut  nonprf  empty  0.000000
description    1-1  act  inv  nonunq  nul  nonmut  nonprf  empty  null
last_failure  TIME  1-1  act  noninv  nonunq  nul  mut  nonprf  empty
              null
last_repair  TIME  1-1  act  noninv  nonunq  nul  mut  nonprf  empty
             null
--------------------------------------------------------------------------------
class: MFR_TECHNICIAN
name    1-1  act  inv  nonunq  nonul  nonmut  prf  empty  null
email    1-1  act  inv  nonunq  nonul  nonmut  prf  empty  null
machines  MFR_MACHINE  1-m  act  noninv  nonunq  nonul  nonmut
          nonprf  empty  null
present    1-1  act  inv  nonunq  nonul  mut  prf  empty  0
busy    1-1  act  inv  nonunq  nonul  mut  prf  empty  0
activeopinst  ACTIVEOPINST  1-1  act  noninv  nonunq  nul  mut
              nonprf  chd  null
--------------------------------------------------------------------------------
class: LOT
nonmfrname    1-1  act  inv  nonunq  nonul  nonmut  prf  empty
```

```
------------------------------------------------------------------------------

class: TIME

nonmfrname    1-1  act  inv  nonunq  nonul  nonmut  prf  empty

------------------------------------------------------------------------------

class: ACTIVEOPINST

nonmfrname    1-1  act  inv  nonunq  nonul  nonmut  prf  empty

------------------------------------------------------------------------------
```

A.2 Output From the MFR Interpreter

The output from the MFR interpreter for the example of section 5.2 is the following:

```
MFR_CELLs


identifier ICL, address 1078896, prev 0

name icl

supercell   0

subcells

            photobay1   1095496

            bay1   1101808

            DIFFUSION2   1105592

            DIFFUSION1   1081232

            photobay2   1083120

machines

buffers

            a   1079776
            b1   1080272
            c   1087672
            d1   1080872
            b2   1094176
            d3   1094656
            d2   1095136
description
number 1
specific_names

identifier PHOTO, address 1095496, prev 1078896
name photobay1
supercell ICL  1078896
subcells
machines
            GCA1   1096104
            GCA2   1099016
buffers
description
number 2
specific_names

identifier PHOTO, address 1083120, prev 1095496
name photobay2
supercell ICL  1078896
subcells
machines
            GCA1   1082536
            GCA2   1091264
buffers
```

```
description

number 2

specific_names

            photobay1

            photobay2


identifier DIFFUSION, address 1105592, prev 1083120

name DIFFUSION2

supercell ICL  1078896

subcells

machines

            stepper1   1106200

            ascer   1106904

            stepper3   1107608

            stepper2   1108312

buffers

            big   1109016

description

number 3

specific_names


identifier DIFFUSION, address 1101808, prev 1105592

name bay1

supercell ICL  1078896

subcells

machines

            stepper1   1102416

            ascer   1103120

            stepper3   1103824

            stepper2   1104528
```

```
buffers

            big   1105232

description

number 3

specific_names


identifier DIFFUSION, address 1081232, prev 1101808

name DIFFUSION1

supercell ICL  1078896

subcells

machines

            stepper1   1081840

            ascer   1086608

            stepper3   1089856

            stepper2   1090560

buffers

            big   1087312

description

number 3

specific_names

            bay1

            DIFFUSION2

            DIFFUSION1


MFR_MACHINEs


identifier stepper, address 1090560, prev 0

name stepper2

cell DIFFUSION  1081232

batch 1
```

```
mttf 3.300000

mttr 0.000000

failuremodes

maintmodes

status up

detailed_status

number 3

specific_names


identifier stepper, address 1089856, prev 1090560

name stepper3

cell DIFFUSION  1081232

batch 1

mttf 3.300000

mttr 0.000000

failuremodes

maintmodes

status up

detailed_status

number 3

specific_names


identifier stepper, address 1081840, prev 1089856

name stepper1

cell DIFFUSION  1081232

batch 1

mttf 3.300000

mttr 0.000000

failuremodes

maintmodes
```

```
status up
detailed_status
number 3
specific_names
            stepper3
            stepper2
            stepper1

identifier ascer, address 1086608, prev 1081840
name ascer
cell DIFFUSION  1081232
batch 1
mttf 1.000000
mttr 0.000000
failuremodes
maintmodes
status up
detailed_status
number 1
specific_names

identifier GCA, address 1091264, prev 1086608
name GCA2
cell PHOTO  1083120
batch 1
mttf 1.000000
mttr 0.000000
failuremodes
            lamp   1091968
            computer   1092624
```

```
maintmodes
            focus   1093280
            lamp   1093728
status up
detailed_status
number 2
specific_names

identifier GCA, address 1082536, prev 1091264
name GCA1
cell PHOTO  1083120
batch 1
mttf 1.000000
mttr 0.000000
failuremodes
            lamp   1083952
            computer   1085176
maintmodes
            focus   1084728
            lamp   1085712
status up
detailed_status
number 2
specific_names
            GCA2
            GCA1

identifier GCA, address 1096104, prev 1082536
name GCA1
cell PHOTO  1095496
```

```
batch 1
mttf 1.000000
mttr 0.000000
failuremodes
            lamp   1096808
            computer   1097464
maintmodes
            focus   1098120
            lamp   1098568
status up
detailed_status
number 2
specific_names

identifier GCA, address 1099016, prev 1096104
name GCA2
cell PHOTO  1095496
batch 1
mttf 1.000000
mttr 0.000000
failuremodes
            lamp   1099720
            computer   1100376
maintmodes
            focus   1101032
            lamp   1101480
status up
detailed_status
number 2
specific_names
```

```
identifier stepper, address 1102416, prev 1099016

name stepper1

cell DIFFUSION  1101808

batch 1

mttf 3.300000

mttr 0.000000

failuremodes

maintmodes

status up

detailed_status

number 3

specific_names


identifier ascer, address 1103120, prev 1102416

name ascer

cell DIFFUSION  1101808

batch 1

mttf 1.000000

mttr 0.000000

failuremodes

maintmodes

status up

detailed_status

number 1

specific_names


identifier stepper, address 1103824, prev 1103120

name stepper3

cell DIFFUSION  1101808
```

```
batch 1
mttf 3.300000
mttr 0.000000
failuremodes
maintmodes
status up
detailed_status
number 3
specific_names

identifier stepper, address 1104528, prev 1103824
name stepper2
cell DIFFUSION  1101808
batch 1
mttf 3.300000
mttr 0.000000
failuremodes
maintmodes
status up
detailed_status
number 3
specific_names

identifier stepper, address 1106200, prev 1104528
name stepper1
cell DIFFUSION  1105592
batch 1
mttf 3.300000
mttr 0.000000
failuremodes
```

```
maintmodes
status up
detailed_status
number 3
specific_names

identifier ascer, address 1106904, prev 1106200
name ascer
cell DIFFUSION  1105592
batch 1
mttf 1.000000
mttr 0.000000
failuremodes
maintmodes
status up
detailed_status
number 1
specific_names

identifier stepper, address 1107608, prev 1106904
name stepper3
cell DIFFUSION  1105592
batch 1
mttf 3.300000
mttr 0.000000
failuremodes
maintmodes
status up
detailed_status
number 3
```

```
specific_names

identifier stepper, address 1108312, prev 1107608
name stepper2
cell DIFFUSION  1105592
batch 1
mttf 3.300000
mttr 0.000000
failuremodes
maintmodes
status up
detailed_status
number 3
specific_names

MFR_BUFFERs

identifier b, address 1094176, prev 0
name b2
cell ICL  1078896
capacity 0
lots
number 2
specific_names

identifier b, address 1080272, prev 1094176
name b1
cell ICL  1078896
capacity 0
lots
```

```
number 2
specific_names
            b2
            b1

identifier d, address 1095136, prev 1080272
name d2
cell ICL  1078896
capacity 0
lots
number 3
specific_names

identifier d, address 1094656, prev 1095136
name d3
cell ICL  1078896
capacity 0
lots
number 3
specific_names

identifier d, address 1080872, prev 1094656
name d1
cell ICL  1078896
capacity 0
lots
number 3
specific_names
            d3
            d2
            d1

identifier a, address 1079776, prev 1080872
name a
cell ICL  1078896
capacity 0
lots
number 1
specific_names

identifier big, address 1087312, prev 1079776
name big
cell DIFFUSION  1081232
capacity 0
lots
number 1
specific_names

identifier c, address 1087672, prev 1087312
name c
cell ICL  1078896
capacity 100
lots
number 1
specific_names

identifier big, address 1105232, prev 1087672
name big
cell DIFFUSION  1101808
capacity 0
```

```
lots

number 1

specific_names


identifier big, address 1109016, prev 1105232

name big

cell DIFFUSION  1105592

capacity 0

lots

number 1

specific_names


MFR_MAINTMODEs


identifier focus, address 1084728, prev 0

name focus

maintmax 3

description

currentmaint 0


identifier lamp, address 1085712, prev 1084728

name lamp

maintmax 56

description I have no idea how to maintain a lamp!

currentmaint 0


identifier focus, address 1093280, prev 1085712

name focus

maintmax 3

description
```

```
currentmaint 0

identifier lamp, address 1093728, prev 1093280
name lamp
maintmax 56
description I have no idea how to maintain a lamp!
currentmaint 0

identifier focus, address 1098120, prev 1093728
name focus
maintmax 3
description
currentmaint 0

identifier lamp, address 1098568, prev 1098120
name lamp
maintmax 56
description I have no idea how to maintain a lamp!
currentmaint 0

identifier focus, address 1101032, prev 1098568
name focus
maintmax 3
description
currentmaint 0

identifier lamp, address 1101480, prev 1101032
name lamp
maintmax 56
description I have no idea how to maintain a lamp!
```

```
currentmaint 0

MFR_FAILUREMODEs

identifier lamp, address 1083952, prev 0
name lamp
mttf 34.200001
mttr 0.000000
description exploding lamp...
last_failure   0
last_repair   0

identifier computer, address 1085176, prev 1083952
name computer
mttf 5.500000
mttr 0.400000
description computer failure can cause machine GCA to be down
last_failure   0
last_repair   0

identifier lamp, address 1091968, prev 1085176
name lamp
mttf 34.200001
mttr 0.000000
description exploding lamp...
last_failure   0
last_repair   0

identifier computer, address 1092624, prev 1091968
name computer
```

```
mttf 5.500000

mttr 0.400000

description computer failure can cause machine GCA to be down

last_failure   0

last_repair   0


identifier lamp, address 1096808, prev 1092624

name lamp

mttf 34.200001

mttr 0.000000

description exploding lamp...

last_failure   0

last_repair   0


identifier computer, address 1097464, prev 1096808

name computer

mttf 5.500000

mttr 0.400000

description computer failure can cause machine GCA to be down

last_failure   0

last_repair   0


identifier lamp, address 1099720, prev 1097464

name lamp

mttf 34.200001

mttr 0.000000

description exploding lamp...

last_failure   0

last_repair   0
```

```
identifier computer, address 1100376, prev 1099720

name computer

mttf 5.500000

mttr 0.400000

description computer failure can cause machine GCA to be down

last_failure   0

last_repair   0


MFR_TECHNICIANs


identifier me, address 1086040, prev 0

name me

email mstamato

machines

            GCA2   1099016

            GCA1   1096104

            GCA2   1091264

            ascer  1106904

            ascer  1103120

            ascer  1086608

            GCA1   1082536

present 0

busy 0

activeopinst   0


LOTs


TIMEs


ACTIVEOPINSTs
```

Appendix B

Makefile for the MFR Interpreter

This Makefile can be used to compile the MFR Interpreter:

```
# BINDIR is where the executable goes
BINDIR = .

CAFEROOT = /fs/cafe/cafe
INCLUDE = $(CAFEROOT)/include
CFLAGS = -g -I$(INCLUDE) -I$(INCLUDE)/gdm
LIB = $(CAFEROOT)/lib/gdm.a $(CAFEROOT)/lib/cafe.a

CC = gcc -g
TARGET = $(BINDIR)/mfri

HEADERS = interp.h

SRCS = y.tab.c instantiate.c allocate.c malloc.c\
        default.c read_attr.c store_attr.c\
        printmfr.c nest.c insert.c utilities.c default_names.c\
        consistency.c unravel.c replicate.c main.c subobject_list.c
```

```
OBJS = y.tab.o instantiate.o allocate.o malloc.o\
        default.o read_attr.o store_attr.o\
        printmfr.o nest.o insert.o utilities.o default_names.o\
        consistency.o unravel.o  replicate.o main.o subobject_list.o

SRCS2 = dprint.c
OBJS2 = dprint.o

mfri : $(OBJS)
        $(CC) -o $(TARGET) $(OBJS)  $(LIB) -ly -ll -lm


$(OBJS): $(HEADERS)

lintout : $(SRCS)
        lint $(SRCS) $(CFLAGS) > lintout

lintout2 : $(SRCS)
        lint -bach $(SRCS) $(CFLAGS) > lintout

TAGS : $(SRCS) $(HEADERS)
        etags $(SRCS) $(HEADERS)
```

Appendix C

MFR Textual Representation for the Integrated Circuits Laboratory at MIT

```
/***********************************************************************

This is an MFR representation of ICL.

***********************************************************************/

MFR_CELL ICL {
subcells PHOTOLITHOGRAPHY_CELL, ETCH_CELL, DIFFUSION_CELL,
 METALIZATION_CELL, ION_IMPLANT_CELL, INSPECTION_CELL;
};


MFR_CELL PHOTOLITHOGRAPHY_CELL{
machines asher{mttr 50.0; mttf 624.2; /*number 2;*/},
stepper{mttr 51.0;  mttf 435.7;},
coater{mttr 31.0; mttf 594.6;},
HMDS{mttr 25.0; mttf 8735.0;},
developer{mttf 1440.0; mttr 72.0;};
},

ETCH_CELL{
machines etcher-1{mttr 32.0; mttf 453.7;},
etcher-2{mttr 154.0; mttf 952.2;},
etcher-3{mttr 64.0; mttf 423.1;};

},

DIFFUSION_CELL{
machines rca{mttr 48.0; mttf 2141.7; /* number 3;*/},
tubeA1{mttr 30.0; mttf 8730.0;},
tubeA2{mttr 44.0; mttf 2876.0;},
tubeA4{mttr 0.0; mttf 99999.9;},
tubeA5{mttr 205.0; mttf 1032.1;},
tubeA6{mttr 115.0; mttf 1344.8;},
tubeA7{mttr 99.0; mttf 448.4;},
tubeA8{mttr 390.0; mttf 1799.5;},
tubeB1{mttr 30.0; mttf 8730.0;},
tubeB5{mttr 30.0; mttf 8730.0;},
tubeB6{mttr 83.0; mttf 4297.5;},
tubeB8{mttr 0.0; mttf 99999.9;};

},

METALIZATION_CELL{
machines nitride{mttr 145.0; mttf 1314.7;},
 oxide{mttr 89.0; mttf 425.9;},
 pre-metal{mttr 29.0; mttf 640.4;};
},

ION_IMPLANT_CELL{
machines VTR{mttr 155.0; mttf 1304.7;},
varian{mttr 150.0; mttf 945.1;},
implanter{mttr 72.0; mttf 720.0;};

},

INSPECTION_CELL{
machines ellipsometer{mttf 99999.9; mttr 0.0;},
 nanospec{    mttf 99999.9; mttr 0.0;},
         microscope{mttf 17520.0; mttr 120.0;},
 dektak{},
 osi{},
 prometrix{};

};

end
```

Appendix D

Programmer's Manual for the Production Requirements Feasibility Program

This Appendix provides a guide to the program which checks for the feasibility of meeting production requirements of factories. The goals of this program are set in Chapter 3.

There are two versions of this program, one that links with the CAFE data base and gives the option of reading data from it, and one that does not. The latter one can be used as a stand alone program. Both versions have the same architecture, except that the former one has some extra routines for getting MFR and PFR information from the data base. These routines are **`read_processes`**, **`read_mfr_machines`**, **`read_descr`** and **`get_fringe`**.

D.1 The Data Structures

The data structures of this program are constructed in the header file *feasibility.h*. An overview of them is illustrated in Figure D-1.

A linked list of processes is constructed by the **`processflow_struct`**. There, the total operating time (**`ideal_cycle`**), the demand for the product (**`demand`**), a name

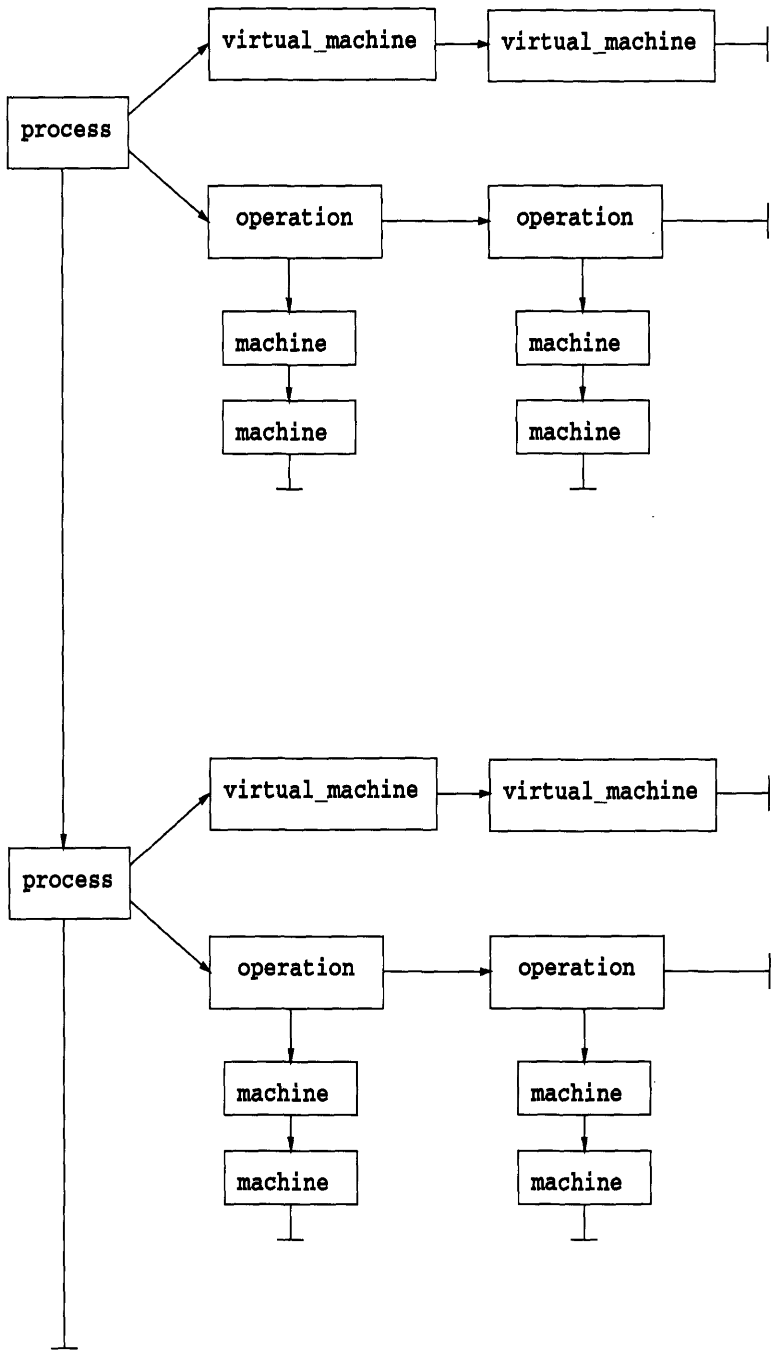

Figure D-1: Data structures of the production requirements feasibility program.

(**prsfl**), various flags as well as pointers to the head of linked lists of operations (**head_descr**)and virtual machines (**head_vmachine**) are stored. In the first version of the program that is linked with the CAFE data base, a pointer to the corresponding PROCESSDESCRIPTION is part of this struct as well.

The struct **descr_struct** contains information for each operation. It consists of a generic machine name (**mchn**), a pointer to a list of machine pointers (**atr_head_machine**), and the time required for the operation (**tint**). The linked list of machine pointers is constructed by the **atr_machine_struct** which has a pointer to a **machine_struct**.

The **machine_struct** forms an independent linked list of all machines in the factory. It includes a name (**mchn**), the mean times to failure (**mttf**) and repair (**mttr**), the efficiency (**e**), and the total demand required by this machine (**tot_demand**).

The **vmachine_struct** consists of **r** and **p** which are the reciprocal of the mean times to repair and failure respectively, **mu** which represents the rate of production for the virtual machine, and **buffer** which is the space of the downstream buffer.

D.2 The Program's Architecture

An overview of the program's architecture is shown in Figure D-2. The MFR interpreter is used to read data form a textual representation of the factory to be tested. In order to do that the source code of the MFR interpreter is linked together with the rest of the program. The only changes made are in the **main** function which is merged with the one of the production feasibility program.

D.2.1 The main function

A description of the tasks performed by the **main** function follows. First, the command line of this program is read with the aid of the **getopt** function provided by C. Then, the cache file where the information about the processes of the products to be produced is stored, is opened. If the **-d** option was used, and the processes are to be read from the data base, the cache file will be opened as a write file, otherwise it will be opened as a read file. Next, the report files (long and sort form), the MFR textual

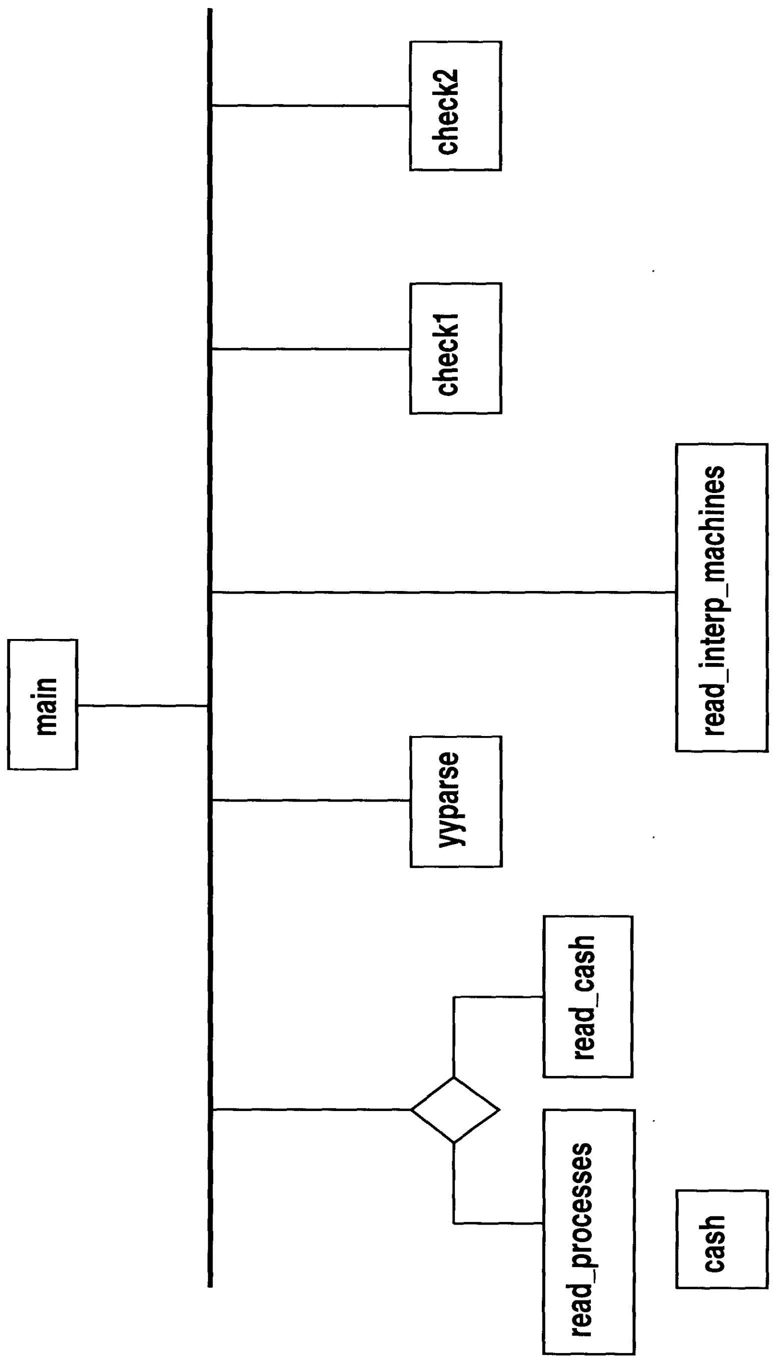

Figure D-2: Architecture of the production requirements feasibility program.

representation file and the MFR output file are opened. The textual representation file is run through the C preprocessor cpp and the output is the **yyin** used by the parser. Then, the function **initialize** is called, and the global variables **upbound** and **lowbound** referring to the upper and lower bounds of utilization for which machines will be flagged, are set. Next, the input file is read and the processflow names are stored as a linked list. A discussion of the data structures can be found in D.1.

If the process data are to be read from the data base, the function **read_processes** which finds the PROCESSDESCRIPTION objects, the function **read_descr** that finds and reads the fringe of PROCESSFLOWs as well as the function **cache** are called. Otherwise, the function **read_cache** which gets information from a cache file is called. If the factory data are to be read form the data base, **read_mfr_machines** is called, if not, the mfr interpreter (**yyparse**) and **read_interp_machines** which gets data form the interpreters data structures are called. Next, **check1**, the function that checks the machine existence feasibility is initiated. Finally, if the **-c** option was recognized, **check2** which performs the capacity feasibility check is called.

D.2.2 The function read_processes

This functions loops through all the processes read from the input file and uses **create_predicate** provided by gestalt [7] to find the PROCESSDESCRIPTIONs needed. The names the query looks for are the ones read from the input file capitalized. The last version of the processdescriptions is found, and its pointer is stored as part of the **processflow** struct.

D.2.3 The functions read_descr and get_fringe

Function **read_descr** is responsible for getting all the information required from the data base for the current process. The pointer to the current process is a global variable, so this function does not have any arguments. First, the fringe of PROCESSFLOWs is found by **get_fringe**. This function takes the FLOWROOT of the PROCESSDESCRIPTION object as an argument, iterates through the list of subflows and if it finds a leaf flow it adds it to the fringe. Otherwise, it calls itself recursively.

Once the fringe is found, **read_descr** loops through it and stores the relevant information. For every leaf PROCESSFLOW, memory is allocated, and the machine name as well as the time duration of the PROCESSFLOW are stored. This way a linked list of leaf **descr** is created.

D.2.4 The functions cache and read_cache

Function **cache** creates the cache file. It loops through all processes, and then through all the operations. For each operation, the machine name and the time duration is seconds are printed on a line of their own. If the machine name is null, "NULL" is printed for the machine name. At the end of every process a - 0 is printed.

Function **read_cache** reads the cache file and stores the data. It allocates memory and creates the linked list of the operations for every process.

D.2.5 The functions read_mfr_machines and read_interp_machines

Both these functions get and store data regarding the machines in the factory under consideration. Also, both construct a linked list of machines where the relevant information is stored. The data stored, are the machine names, mean time to failure and mean time to repair.

Function **read_mfr_machines** reads this information from the CAFE data base by looping through all MFR machines in it. On the other hand, **read_interp_machines**, reads this information by looping through all the machines in the linked list created by the MFR textual representation interpreter.

D.2.6 The function check1

This routine checks for machine existence feasibility, assigns a list of possible machines to every operation and calculates the hours of operation required from each machine. It goes through all operations of every process, finds the machine(s) that can perform the operation, and makes a linked list of them. The machines are recognized by a string compare of the first n characters, where n is the number of characters of the machine specified in the operation. This is done because the MFR interpreter, in the case of multiple identical machines, will add a number suffix to the generic name of a machine type.

If the machine name for a particular operation is null, a warning is printed, the existence feasibility check does not fail, and no time is added to the demand of any machine. If it is not null, the hours demanded from the machine(s) that can do the operation is incremented by the time required by the operation times the demand for the product in question divided by the number of identical machines. If the machine name is not null and no machine is found, an error message is printed in the report file, and the counter of missing machines is incremented. For every process, the total operating time is calculated and printed in both the long and brief report files.

D.2.7 The function check2

This function calculates the utilization of machines in the factory, flags the ones that are not within the prespecified bounds, and makes the capacity feasibility check fail for the ones that have more than 100% utilization. The utilization is calculated by dividing the total number of hours required from each machine by the machine efficiency times the number of hours the factory operates in a year. The machine efficiency is calculated by the following formula:

$$e = \frac{mttf}{mttf + mttr}$$

where r and p are the reciprocal of the mean time to repair and mean time to failure respectively. For more information on those calculations please refer to Section 3.2.2.

D.2.8 A routine that unwinds the processes

This program was written under very general assumptions on the factory and the routing of the products. Thus, use of the same machine for several processes, re-entrance, and multiple identical machines are allowed. All these characteristics make the problem of performance measure calculations very difficult. A routine has been written that approximates the system with several transfer lines, one for every product.

Function **`trlineapprox`** loops through all products, calls **`cr_virtual_machines`** which crates a transfer line for each one, and prints information about the machines that it consists of in a *.vmchn* file. The approximation is based on an idea developed in [1].

In **`cr_virtual_machines`** processes are unwound to a transfer line. A linked list of virtual machines that make up this transfer line is created. The parameters provided for these virtual machines are the inverse of the mean times to repair and failure, their rate of operation, and the downstream buffer space. The first virtual machine, is a machine whose purpose is to control the release of parts in the line. It is very reliable, it has a big buffer and its rate of operation is the rate of the demand for the particular product. The other virtual machines correspond to the operations the product type has to go through. If the machine for the operation is null, a reliable and fast machine is assigned. In particular its speed is a hundred times the demand. If it is not null, the mean times for failure and repair of the virtual machine are the ones of the real machine where the operation is to be done. The rate of the virtual machine is set to be the inverse of the time duration for the operation times the fraction of this time duration over the total demand required from the actual machine. Note that if a machine is one of several identical ones, the total demand for each machine is equally distributed among those machines. This is taken care of in **`check1`**.

Bibliography

[1] Sherman X. Bai. *Scheduling Manufacturing Systems with Work-in-process Inventory Control.* PhD thesis, Massachusetts Institute of Technology, September 1991.

[2] Stanley B. Gershwin. *Manufacturing Systems Engineering.* Prentice Hall, Inc, 1994.

[3] Tony Mason John R. Levine and Doug Brown. *Lex and Yacc.* O'Reilly and Associates, Inc, 1992.

[4] Asbjoern M. Bonvik. Design proposal for the CAFE scheduling application. *CIDM Memo No. 94-6*, 1994.

[5] Donald E. Troxel, Jeffrey M. Kinecht, James E. Marguia, Thomas M. Ventura, George R. Young, Gregory T. Fisher, Thomas J. Lohman.
CAFE Installation at Lincoln Laboratory and Expanded PFR Based IC Fabrication. *CIDM Memo No. 93-14*, 1993.

[6] Duane S. Boning and Michael B. McIlrath.
Guide to the Process Flow Representation, version 3.1. *CIDM Memo No. 93-17*, 1993.

[7] Michael L. Heytens, Rishiyur S. Nikhil.
GESTALT: An Expressive Database Programming System. *ACM SIGMOD Record*, 18(1), March 1989.

[8] Michael B. McIlrath, Donald E. Troxel, Michael L. Heytens, Paul Penfield Jr., Duane S. Boning, and R. Jayavant. CAFE—the MIT Computer Aided Fabrication Environment. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 15(2):353–360, May 1992.

[9] SEMATECH. *SWIM (J58) Symposium Meeting Minutes*, Wyndham Hotel, Austin, Texas, February 18 1994.