Practical Training Report:

A SOFTWARE LIBRARY FOR LOGIC CIRCUITS

DESIGN AND IMPLEMENTATION OF A DATA STRUCTURE FOR BOOLEAN FUNCTIONS AND SEQUENTIAL MACHINES IN C++

H. Goossens

Coach : P. Konieczny
Supervisor : Dr L. Jozwiak
Period : May 1993 - November 1994
SUMMARY

In this report the design of a software library for logic circuits is discussed. At Eindhoven University, in the digital information systems group, there was a need for a user-friendly and transparent software library for logic optimization in a modern (object-oriented) programming language. The reason for this was that object oriented languages have lots of advantages compared to classic languages as Pascal and C, so the existing programs had to be replaced by a new library written in C++. In this report the differences between the two kinds of languages and some properties of C++ will be explained.

The software library had to contain data structures and algorithms for two kinds of logic circuits: boolean functions and sequential machines. The last were also separated in Moore and Mealy machines. The aim of the work described in this report was to design and implement the data structures for each of the circuits functions as well as the algorithms to read descriptions of that circuit from a text file and store them in these data structures. A boolean function is read following the PLA format. Sequential machines follow the so called KISS format. The PLA and KISS syntaxes will be explained in this report.

It appears that the PLA format is a good model for a boolean function. This model is implemented in the libraries data structure in the class Pia. A good model for sequential machines is their descriptions in input set, output set, state set, transition function and output function. So classes for Moore and Mealy machines are build from objects that represent these sets and functions. The implementation of these classes and their member functions will be discussed. Only basic functions have been designed. It appears that for many objects the container classes array and list are needed. For these classes the implementation of the Borland Container Library is used. This library defines an object Array and List. A little introduction on these objects is given.

An important part of the input functions is the error detection. As many errors as possible should be detected in the PLA and KISS input. A chapter is dedicated to error detection.

It appears that with the classes based on the mentioned models, a user-friendly library was created. Because the classes are very similar to the models we use for the different logic circuits, a transparent implementation was designed.
CONTENTS

1. INTRODUCTION ......................................................................................................................... 1

2. MODERN PROGRAMMING AND C++ ...................................................................................... 2
   2.1 Introduction .......................................................................................................................... 2
   2.2 The basics of modern programming ...................................................................................... 2
       2.2.1 Modular programming ................................................................................................. 2
       2.2.2 Data abstraction ......................................................................................................... 2
       2.2.3 Object oriented programming ..................................................................................... 3
   2.3 Modern programming in C++ .............................................................................................. 3
       2.3.1 Modular programming ................................................................................................. 3
       2.3.2 Data abstraction ......................................................................................................... 4
       2.3.3 Object oriented programming ..................................................................................... 5
   2.4 Some important features of C++ ......................................................................................... 5
       2.4.1 Abstract classes .......................................................................................................... 6
       2.4.2 Access control ............................................................................................................ 6
       2.4.3 Streams ..................................................................................................................... 7

3. BORLAND CONTAINER CLASSES ......................................................................................... 8
   3.1 The Array class .................................................................................................................. 8
   3.2 The List and ListIterator classes ....................................................................................... 9

4. INPUTFORMATS ....................................................................................................................... 11
   4.1 The PLA format .................................................................................................................. 11
       4.1.1 Commands .................................................................................................................. 11
       4.1.2 Logical description ..................................................................................................... 12
       4.1.3 Example ..................................................................................................................... 12
   4.2 The KISS format .................................................................................................................. 13
       4.2.1 Mealy machines ......................................................................................................... 13
       4.2.2 Moore machines ....................................................................................................... 14

5. CLASSES FOR LOGIC CIRCUITS ....................................................................................... 15
   5.1 A class for boolean functions ............................................................................................ 15
Appendix B COMPLETE PROGRAM LISTING

I_pla.h ..........................................................74
I_pla.cpp ......................................................75
I_seq.h ..........................................................79
I_seq.cpp ......................................................81
I_set_fu.h ......................................................94
I_set_fu.cpp ...................................................98
I_elems.h ......................................................114
I_elems.cpp ....................................................118
I_term.h .......................................................119
I_term.cpp ....................................................121
packarray.h ..................................................128
packaray.cpp ................................................130
I_suppor.h ......................................................131
I_suppor.cpp ..................................................132
IDefs.h ........................................................133
strmable.h ....................................................134
I_muex.cpp ....................................................135
1. INTRODUCTION

At Eindhoven Technical University the digital information systems group EB is doing research in some different areas. One such area is the logic optimisation of both sequential machines and boolean functions. Examples are decomposition of sequential machines and finding minterms of boolean functions. Especially when realistic circuits with lots of variables are involved, the help of computers is indispensable. So many programs have already been written to manipulate logic circuits. Most of these programs were written in Pascal or C and functioned well.

Nowadays object oriented programming is being used more and more. Object oriented languages have some very important advantages compared to languages as Pascal and C. Object oriented languages seem to become the programming languages of the future. Because of this it was decided at EB that from now programs for logic optimisation will be written in an object oriented language. For C++ is used by many people, this language was chosen. So a software library for logic optimisation in C++ had to be made. In the rest of this report the library will be referred to as the logic library.

The first step in this project was writing functions that could read a circuit specification from a certain external text file into an internal data structure. For boolean functions the so called 'PLA' format is used in such a text file. Moore and Mealy machines can be specified in 'KISS' syntax. The input functions have to check if the input contains no syntax- or other errors. The next step was writing a function that made the input terms of a sequential machine non-overlapping. This has a lot of advantages for further processing.

The above was the initial goal of this project. In practice however the main problem was to find a data structure, or better object structure, to store and manipulate the different kinds of circuits. This structure had to be made user-friendly and transparent because the library has probably to be changed in the future for new features and applications. The biggest part of this report is about the design of the data structure. The data structure was build layer by layer, starting at the top.

This process is described in the chapters 5 and 6. The top layer contains objects for Pla's, Moore and Mealy machines. People that are only interested in the top layer can concentrate on the first of the two chapters. In chapter 6 the underlaying objects are described. Before we come to these chapters a little introduction on object oriented programming and C++ is given in chapter 2. After that the 'PLA' and 'KISS' formats are explained. Finally in chapter 7 special attention is payed to the error detection in the different input functions.
2. MODERN PROGRAMMING AND C++

2.1 Introduction

In the preface was stated that C++ has a lot of advantages compared to Pascal and C. That is true because C++ provides mechanisms that support some modern programming techniques: modular programming, data abstraction and object oriented programming. Modular programming is also possible in Pascal and C. Data abstraction and object oriented programming however ask for exceptional efforts when used writing C and Pascal programs. This chapter explains the techniques mentioned above and how they are implemented in C++. Of course this chapter is not a complete C++ manual. Only very important concepts that are essential for this report are mentioned. In this way the following chapters can be read without a complete knowledge of the C++ language. For those who want to learn C++, I recommend Bjarne Stroustrup's "THE C++ PROGRAMMING LANGUAGE".

2.2 The basics of modern programming

2.2.1 Modular programming

Over the years the emphasis in the design of programs has shifted from the design of procedures towards the organisation of data. The concept of using procedures should be familiar to the reader, so it isn't explained any further here. Procedures create merely order and clearness in algorithms. When the size of programs increased, more clearness in data became important. "Data hiding" became a major subject in programming. The use of modules follows this data hiding principle. A set of related procedures with the data they manipulate is often called a module. Only parts of the module that are known outside the module can be used by the program and programmer. The other parts can only be used inside the module. So the module hides information. This is only interesting in the design of large programs. Standard Pascal does not support modular programming. Both C and C++ are in the contrary suited for working with modules.

2.2.2 Data abstraction

It can be easily seen that user-defined types are structures that are very suitable to be build in a module. We can put the data structure together with the procedures for this type in one module. When no special care is taken this way of programming causes problems however. These problems concern especially declaring and destroying variables (or better objects) of such a user-defined type. When a variable of a build-in type is declared some actions are taken automatically by the compiler. When the program leaves the block in which the variable was declared, it destroys the variable automatically. One would like that user-defined types (or also called abstract data types in literature) behave in the same way. If we can declare and destroy variables of user-defined type in the same way as we would do that with build-in types, this is called data abstraction. This is also supported by C++. It is not supported by C or Pascal.
Data abstraction is almost necessary in case there are more objects of one abstract data type.

2.2.3 Object oriented programming

At last we would like to have a mechanism that lets modules inherit properties from other modules. A language that provides this mechanism is called an object oriented language. With this data inheritance we can create similar modules by inheriting their common properties from a base module and adding specific properties to each of them. We could for example create models for the shapes circle and square. Circles and squares have some properties in common. For instance their color. They also have different properties like how they are drawn. We can put the color in a base model called "Shape" and put the different drawing functions in the modules "Circle" and "Square".

2.3 Modern programming in C++

At this point it is clear what the basics are in modern programming. C++ appears to support all these concepts. In the following paragraphs will be shown how these mechanisms can be coded in C++.

2.3.1 Modular programming

In fact C++ knows two ways to create modules. At the first place files can be used to create modules and to hide information. For small programs the complete program text can be placed in one file. For large programs it's better to split the program into different parts. In C++ there are, roughly said, three kinds of file types. These three kinds of files are applications, implementations and interfaces. The interfaces, or also called header-files, contain declarations of types, variables and functions. These files usually have the extension ".h". Other C++ files generally get the extension ".cpp". The function definitions are generally in the implementation files. Without knowing what the implementation is, we can use the concepts from a header file in another file. C++ provides the preprocessor command "#include < >" by which we can include a header file. The implementation is hidden for the user. He has to know only the interface to use it's implementation. To complete a working program a main function is needed. This function is usually in the so called application file. After including the appropriate header files and linking the application file with the corresponding implementations a working program is be obtained. The advantages of this file structure are not only information hiding, but also the fact that a program can be changed in an easier way. After one of the files has been changed, only that file has to be recompiled. All other files can be used again. In this report the design of some different header files and their implementations will explained. These files together form the logic library.

The other way to create modules in C++ is the "class" construction. Such a class is the definition of a special kind of type. It is something like a record in Pascal, but contains data as well as procedures that manipulate that data. Variables of this type are usually called objects. Each variable and procedure (or better function, because C++ has only functions) in the class definition is called a member. These members can be local to the class i.e. they can only be used by other members of the same class. In C++ the word "private" is used for this. The word "public" is used for members that can be used outside the class. With this knowledge we could design for example a class "stack" in which we can store integers. In our program we would like to ma-
Manipulate the stack only with the functions "push" and "pop", as is usual for a (LIFO) stack. The interface for the stack class should become something like this.

```cpp
class stack {
private:
    int s[9]; // array of 10 integers to store the stack
    int top; // the stack top index

public:
    void push(int i); // function to push integer i on the stack
    int pop(void); // function that returns and deletes top of stack
};
```

This class hides the data that is on the stack. From outside the class the array s can not be accessed, because it's private. Only the public functions push and pop can be used from outside the class. On the contrary push and pop can use array s and integer top.

From this little program we learn how to declare a function. First specify a return type, then the function name, followed by the arguments the function takes. In the declaration the argument-names may be omitted. The return type of push is void, which means that push doesn't return any value. Pop returns an integer, but takes no arguments.

We could now write the following piece of program to show how applications with this class can be made.

```cpp
S.push(10); // push 10 on stack S
int i = S.pop(); // declare integer i and assign it to the result of pop
```

The major advantage of this class concept is that, when we decide that in some application an array is not suitable anymore for storing the stack elements, we don't have to change the whole program. Only the stack class has to be changed. The use of its public functions remains the same.

### 2.3.2 Data abstraction

In the above form the definition of the stack class can't be used yet. The problem how to declare an object of the stack type. This is because we haven't defined what to do when such an object is created. This can be done by a constructor. This is a function that returns nothing. The constructor's name and the class name have to be identical. With this constructor the object is initialized. It is automatically called when an object is declared. The stack example needs a constructor that creates an empty stack. This means that top equals -1. This is coded in the following way.

```cpp
class stack {
    ... public:
        stack(void) { // constructor with no arguments
            top = -1; // make top -1
        }
    ...};
```

When an object isn't needed anymore, it should be destroyed. When the block in which an object was declared is left, the compiler destroys all data members. In the case when we work with pointers, it is not enough to destroy these pointers. We have
to free the occupied memory explicitly. For this we use the destructor function. By
the destructor we can define what has to be done when an object is destroyed. The
destructor also has the same name as the class but is preceeded by a "~". In the
stack example no destructor is needed. Because of these constructor and destructor
mechanisms classes can be used in the same way as build-in types like integers etc.

2.3.3 Object oriented programming

To be an object oriented language, C++ has to support inheritance. This is obtained
by the concept of derived and base classes. Common properties can be put in a base
class, from which other, more specific classes can be derived. When we would like to
make a program for manipulating different shapes (see 2.2.3) we would get some
code like the following.

```cpp
class shape {
private:
    color c; // color of the shape
...
public:
    set_color(color);
    ...
};

class circle : public shape {
    ...
    void draw(void); // draw the circle on the screen
    ...
};
```

In this example shape is the base class. By putting ": public shape" behind the class
name "circle" we make it a derived class of shape. Now circle has all the properties
that shape has. Next to that, it has its own function to draw the circle on the screen.
Think of this as circle being a shape with some extra properties.

Because of the properties mentioned above, C++ has some advantages that help to
tackle a general problem in computer science. This problem is that humans naturally
think differently about problems and their solutions than computers do. Computers are
only able to manipulate bits or, at a higher level, to do some arithmetic operations.
Humans on the other hand create models of complex things and want to manipulate
these models. A modern programming language like C++ provides a link between
these two ways of functioning. One can create classes that represent models of com­
plex things. These classes use other classes of a lower level that are also models of
complex things. The algorithms that we invent for our model can be implemented al­
most literally if we want. The classes at the bottom level provide the mapping of the
complex model onto integers, characters, etc so that the computer can work with
them. Later we will see how this works for te models of boolean functions and se­
quential machines.

2.4 Some important features of C++

Now the basic concepts of C++ have been explained, some other features that are im-
portant in the design of the logic library should be clarified. With this and the explana-
tion above the reader should be able to understand the concept of the design in this
report without an extensive knowledge of the C++ language.
2.4.1 Abstract classes

As we know, base classes can be defined in C++. But in fact these classes are not different from any other class. In other words, each class can be a base class. So we could declare objects of a base class. In many cases this is not wanted. For example why would we declare an object of the shape class (see paragraph 2.3.3). We couldn't even draw it on the screen. To prevent this from happening the shape class can be made abstract. Objects of an abstract class can't be declared. To make a class abstract we use virtual functions. If a function is marked virtual, this means that the definition of that function may be redefined in a derived class. Virtual functions that are left undefined are called pure virtual. Classes with pure virtual functions are abstract classes. Because objects of abstract classes can't be declared, derived classes of an abstract class have to define the pure virtual function. If not, the derived class is also abstract. In general, virtual functions are used when a function is desired in a certain class, but the implementation depends on some extra information. Reconsidering the shape example, we would like to make draw a virtual function of shape.

```cpp
class shape {
    private:
        color c;          // color of the shape
    ...
    public:
        set_color(color);
        virtual void draw(void) = 0;
        ...
};

class circle : public shape {
    ...
        void draw(void) { /* definition */ }

    ...
};
```

The function draw in shape is made pure virtual by putting "= 0" behind it.

2.4.2 Access control

Members of a class can be private, public, but also protected. Private members can only be accessed by other members of the class. This isn't always an appropriate situation. Sometimes other functions should have access to these members too. These functions can be made friends of the class. For example consider the next class.

```cpp
class complex {
    private:
        float re, im;       // Real and imaginary part of complex number
    public:
        ...
        friend complex operator+(complex, complex);
        ...
};
```

The function to define an add operator takes two complex numbers. Adding the two numbers is very simple. Add both real and imaginary parts. Normally this wouldn't be possible, because the add-function couldn't access re and im. Now that the function is made a friend of the class complex, it can access the two variables. So private members of a class can be accessed by friends and other members of the same
class. Public members can be used by every function. A protected member however can only be used by member functions and friends of the class in which it is declared and by member functions and friends of classes that are derived from this class.

2.4.3 Streams

C++ has no input/output functions of its own. Therefore usually the I/O functions from the stream library are used. This library is not a part of the language itself, but it is used very general and also in the logic library. How to use these streams is explained in this paragraph.

A stream can be considered as a queue of things. These things can be of a build-in type, like integer or char, but also of a user-defined type. There are some standard streams defined by the stream library. Those are "cout" for standard output, "cerr" for error output and "cin" for standard input. Normally the screen is used for the two output streams and the keyboard is used for standard input, but this can be changed by redirection. Files can also be represented by streams. Object to be output can be put on a stream by the << operator. For example

```
cout << 10;
```

prints "10" on the screen. With >> something can be get from a stream. For example

```
stack S;
cin >> S;
```

reads an object S of the user-defined type stack from the standard input. This is possible when the operator >> is redefined for the stack class. Further details on streams will be explained when necessary. Notice that in the logic library all I/O is done with streams.
3. BORLAND CONTAINER CLASSES

The Borland software company provides all kinds of libraries to make programming in C++ easier. One of these libraries is the container class library. A container class is a class that holds objects of some (other) type. Examples of container classes are arrays, lists, sets, etc. In the design of the logic library some container classes are used. We need arrays and lists. For this purpose we use the predefined classes that are in the Borland library, so we don't have to program our own container classes. The predefined array has several advantages. First we can store all kinds of objects in it, as long as these objects are derived from the class Object. The array doesn't take a fixed part of the memory. In fact the array is implemented as an array of pointers to its elements. The users doesn't notice this however. This non-constant size is another advantage of the array.

The container library is based on a class called Object. This abstract class defines some elementary properties that an object of any type should have. For example a print function, a input function, the definition for equality, etc. All these properties (functions) are defined virtual. We have to use this base class to derive classes of our own that we want to store in a container. How this works can be seen in chapter 6. Because we need the Array class and the List class in this report, these two classes will be clarified below. For details on the Borland Container Classes refer to the Borland Programmer's Guide [3].

3.1 The Array class

With the Array class we can, as the name suggests, create arrays of objects that are based on Object. This Array class is derived from a class called AbstractArray, which is indirectly derived from Container. In the logic library we need several arrays of different objects. The public members of Array, AbstractArray and Container that are used in this report will be discussed.

**add**

```cpp
virtual void add(Object& toAdd);
```

(Array) Adds the object toAdd at the first available index at the end of the array.

**addAt**

```cpp
void addAt(Object& toAdd, int atIndex);
```

(Array) Adds the object toAdd at the specified index. If that index is occupied, the previous object is deleted.

**constructor**

```cpp
Array(int Upper, int Lower = 0, sizeType Delta = 0);
```

(Array) Creates an array and makes all elements "NOOBJECT". The upper-bound of the array becomes Upper. The lower bound becomes Lower and is set to 0 by default. The optional argument Delta is the size by which the array can grow during run-time. If the array was declared too small and adding a new object causes an overflow, extra memory can be allocated for the array if Delta isn't zero.
**3.2 The List and ListIterator classes**

The List class provides a structure to create lists of objects. ListIterator provides functions to iterate the list. With ListIterator we can step through the list and read its elements. Here the functions that are used in this report will be discussed.

### List

- **add**
  
  ```cpp
  void add(Object& toAdd);
  ```

  Adds the given object at the head of the list. The added object becomes the new head.

- **constructor**
  
  ```cpp
  List();
  ```

  Creates an empty list.

### ListIterator

- **constructor**
  
  ```cpp
  ListIterator(const List& toIterate);
  ```

  Creates an iterator for the given list. The starting and current elements are set to the head of the list.

- **current**
  
  ```cpp
  virtual Object& current();
  ```

  Returns a reference to the object that is the current element.

- **operator ++**
  
  ```cpp
  virtual Object& operator++ ();
  ```

  Advances the iterator one position in the list and returns the current element after incrementing.

- **operator int**
  
  ```cpp
  virtual operator int ();
  ```
Conversion operator to test if the end of the list is reached.

restart

virtual void restart();

Sets the iterator to the head of the list.
4. INPUTFORMATS

As we know, the input functions of the logic library have to read text files with descriptions of boolean functions or sequential machines. For boolean functions the "PLA" format is used. Sequential machines can be described in "KISS" format. To design appropriate input functions we have to know how these formats are defined. The formats were chosen because they are used very generally.

4.1 The PLA format

As the name suggests, the PLA format follows the implementation of a physical Programmable Logic Array. Each line in the function description corresponds to one row in the physical matrix. The final boolean function is obtained by or-ing the all the function lines. In the PLA file the logical description is preceded by a command block. In the command block some general parameters are set. Usually each line of the file contains one command or one product term. Comment-lines can be added after a '#' or a '"' as first character of a line.

It has to be said that the logic library doesn't follow the original PLA format. Commands as .mv and .symbolic are not implemented. The library uses the format as described below.

4.1.1 Commands

The following commands are defined in PLA format. The commands (except the .e command) have to be in the first lines of the file. Once the logical description has started, no commands are allowed anymore. There is no special order for the commands and not every command has to be used. The only restriction is that .i and .o always have to be in the PLA file and that they should be on the first two lines. The [d] and [s] after the commands denote respectively a decimal number and a string. Notice that commands can be in lower and upper case.

.i [d] Specifies the number of input-variables

.o [d] Specifies the number of output-variables/-functions

.ilb [s] ... [s] Specifies the names of the input-variables. There must be as many names, seperated by white-space, as there are input-variables. This command may be longer than one line.

.ob [s] ... [s] Specifies the names of the output variables. There must be as many names, seperated by white-space, as there are output-variables. This command may be longer than one line.

.type [s] Sets the interpretation of the logical description. This command has to be followed by one of “f”, “r”, “fd”, “fr”, “dr”, “fdr”. The default type is “fr”.

.phase [s] [s] is a string of as many '0's and '1's as there are output-variables. It specifies which polarity of each output function should be used in a minimization. A 1 specifies that the ON-set should be used. A 0 specifies that the OFF-set should be minimized.

.p [d] Specifies the number of product terms that will follow.
4.1.2 Logical description

Each function description line exists of a string of input values and output values. The number of inputs and outputs should be as specified by the .i and .o commands. For inputs the values '0', '1' and '-' can be used. For outputs also '~' is available. With these characters the boolean function can be described. The input values represent a term. Each position in input part of the string corresponds to an input-variable. A '1' means that the corresponding variable is in the term. A 0 means that the complement of the variable is in the term. A '-' denotes that the variable doesn't appear in the term. The output values correspond to the output functions and describe the function values for the given input term. The boolean function is completely described when its ON-set, OFF-set and DC-set are provided. The ON-set of a boolean function is the set of minterms that imply that the function is 1. The OFF-set implies that the function is 0. For all term in the DC-set the output function is undefined. In the PLA format we don't have to specify all three. We can describe a function in one of the following ways.

Providing the ON-set. In this case the OFF-set should be computed as the complement of the ON-set. The DC-set remains empty. This case is indicated by the "type f" command. An input term is in the ON-set of a function, if the corresponding output value is 1. 0, - and ~ mean that the term is not in the ON-set.

Providing the OFF-set. This is like the above option. It is indicated by "type r".

Providing the ON-set and DC-set. For this use the default option "type fd". For each output, a 1 means that the input term is in the ON-set and a - means that the term is in the DC-set. A 0 and a ~ are not defined. The OFF-set is the complement of the union of the ON-set and DC-set.

Providing the OFF-set and DC-set can be done by "type dr". A 0 means OFF-set. A - means DC-set.

Proving the ON-set and the OFF-set. This is indicated by "type fr". Logically a 1 means that the input term belongs to the ON-set and a 0 means that the input term belongs to the OFF-set.

Providing all three is indicated by the option "type fdr". A 1 corresponds to the ON-set, a 0 to the OFF-set and a - to the DC-set.

When a ~ is used for an output this implies that the given input term is not defined for that output function, regardless of the type.

4.1.3 Example

To clarify the Pla format an example follows of some arbitrary boolean function. In Figure 4.1 we see the PLA implementation of this function.
As we see, this is a function with 3 input variables, named i1, i2 and x, and 2 output functions, named o1 and o2. The 5 product terms and output combinations describe the following functions.

\[
\begin{align*}
& o1 = i1i2x + \overline{i1}i2x + \overline{i1i2}x \\
& o2 = x + i1i2x + \overline{i1}i2x + i1i2x
\end{align*}
\]

### 4.2 The KISS format

The KISS format is used to describe sequential machines. Both Moore and Mealy machines can be described in this format. Most of the commands are the same as in the PLA format. So the header of a KISS file is almost the same as for PLA files. Only the .phase command is not supported in KISS. On the other hand the number of states has to be specified in the following way.

\[s \ [d]\] Specifies the number of states

#### 4.2.1 Mealy machines

For Mealy machines the function description lines are separated in four fields, an input field, a current state field, a next state field and an output field. Each line represents a transition of the machine. The input field again is the representation of a term. For this input term the machine goes from the specified current state to the specified next
state and generates the specified output. Both current state and next state can be character strings or don't cares ('-' or '*').

For example the Mealy machine in Figure 4.2 is coded as below.

```
# example of a Mealy machine
.i 2
.o 3
.p 2
.s 2
11 s1 s2 101
-0 s2 s2 010
```

**Figure 4.2 Example of a Mealy machine**

4.2.2 Moore machines

For Moore machines we have to define the state transitions and the output function separately. For this there are two kinds of description lines. The lines that describe the state transitions have the same form as the function description lines of a Mealy machine. Except that all output functions should be defined 0. Their value doesn't matter however. Also the description of the output function has the same structure. In this case however we use the word "void" instead of the next state. An example of how this works is given below. This example is based on the Moore machine of Figure 4.3.

```
# Transition function:
11 s1 s2 000
-0 s2 s2 000
# Output function:
-- s1 void 101
-- s2 void 010
```

**Figure 4.3 Example of a Moore machine**

This machine is described as follows.

```
.i 2
.o 3
.s 2
.p 4
```

This example is based on the Moore machine of Figure 4.3.
5. CLASSES FOR LOGIC CIRCUITS

Now that we know something about C++ and about the syntax of the inputfiles, we can start the design of the logic library. We begin at the top level, what means with the design of classes for the three different logic circuits. From this the will follow what kind of classes of a lower level we need. We will discuss these classes later. For each circuit we will discuss the kind of model that we want to create for that circuit in words. Than the structure of the class as it was coded will be explained and next the implementation of the member functions follow. The members that we need in other classes will follow from these implementations. In this way the library is designed in logical steps.

Of course it isn't possible to construct a program in a strict top-down way. The construction is an iterative process. We can't write the implementations of the upper level functions, when we don't know the implementation of the lower level classes exactly.

5.1 A class for boolean functions

5.1.1 A model

As we have seen the PLA format is already model for a boolean function. The function is modeled by a collection of combinations of inputs and outputs. This structure is also usable for the class we want to design. Because of this the class will be called "Pia". In the Pia class the collection is implemented as an array of inputs and an array of outputs. The arrays are prefered because they give faster access to the data than lists do. The input and output with the same array-index belong together, so that we don't need to make an explicit link between an input and the corresponding output. Of course we also want to store the information that is in the header of the PLA file. We need a special object for this. The design of this will follow later.

In chapter 2 the modularity of C++ was shown. From that we know that we can define special operations on a model and that we can put the model and the operations on it together in one class. For the model of a PLA as described here, we need the following functions.

- Read the description of the boolean function from a file in PLA format.
- Print the description of the function in PLA format.
- Interpret the function description as specified by the "type" command.
- Delete variables
- Test if the object is a valid boolean function. We need this check, because we can't do operations on a non-valid boolean function.

At this point we restrict ourselves to these operations because, now only a basis for the software library is designed. Other functions will be added in the future.
5.1.2 The PLA class

Now we have a model of the PLA and a specification of the functions we would like to have, we can create the Pla class. The C++ code for the class is the following. Some details of it will be discussed. This class definition can be found in the file "_pla.h".

class Pla : public Streamable {

private:
    GeneralInformation info;    // some information
    TermArray *itable, *otable; // the input and output table

public:
    Pla(void);                  // constructor
    ~Pla(void);                 // destructor
    void inputFrom ( istream& in ); // read Pla from input in PLA
    void printOn ( ostream& out ) const; // print Pla in PLA format
    void establishIrrelevant(void); // change irrelevants conform
    void removeIrrelevant(void);  // remove irrelevants
    bool valid(void) const;  // return if pla is valid
    void delVar(int var);     // delete variable at location var

private:
    void init(void);          // allocate space
    void flush(void);         // destroy data
    bool checkInfo(void);     // check info after input
    bool checkInputs(void) const; // check input terms
    void subst(bit oldb, bit newb); // substitute old bit by
                                    // new bit
};

Now some interesting details of this class will be clarified.

The first interesting part is the ": public Streamable" behind the class name in the first line. This makes Pla a class that is derived from Streamable. The purpose of this is that we can make the input and output of a Pla object as simple as the I/O of build-in types. We can just use the >> and << operators. These operators are overloaded by the Streamable class. Because of this construction the function inputFrom is called when >> is used and printOn is called when << is used (see streamable.h in appendix B). For example we could code the following.

Pla P; // declare an object of type Pla and call it P
cin >> P; // read Pla from standard input

The private data members form the model for the boolean function as we have discussed in the previous paragraph. The object "info" of the type GeneralInformation stores the information from the header of the PLA file. TermArray is, as the name says, an object that represents an array of terms. Actually only the inputfields of the PLA file are terms, but the output fields are very much the same as we have seen in chapter 3. Therefore we can use an array of terms for both. The asterixes ("*") before the names of the array, mean that not the real object is declared, but a pointer to the object is declared. This is done because at time of declaration of the arrays we have to know the number of elements in the array. We only know that number after we read the .p command in the PLA file. At that time the real array can be declared. This is done in the private function init. Because we use pointers we have to free the memoryspace that is used by the arrays manually. This is done by the function flush.
The other private functions are needed by some public function. From the implementation of these public functions the meaning of them will become clear.

In paragraph 4.1.2 the meaning of 0, 1, and ~ in the outputfields of the PLA file has been described. This meaning depends on the type of the PLA, except for ~. This character always means irrelevant. The elements of the TermArrays, objects of the class Term, exist of 'bits' that represent 0, 1, or - or ~. The function establishIrrelevant substitutes all bits that mean irrelevant for the specified type by a real irrelevant value. The function removeIrrelevant does the complementary. It substitutes all real irrelevants by other values, of course according to the specified type.

In some-function declarations the word const is used. A const behind the function name means that this function can't modify the object.

The type "bool" is not a C++ standard type. It is defined, together with some other types and constants, in a definition file. This file is called "_defs.h" and is included in the file "_pla.h".

5.1.3 Implementation of the Pla class

At this point we have a definition of the functions and data members of the Pla class. We can't however make an application with it, because we have no implementations of the functions. These implementations can be found in the file "_pla.cpp" that is in appendix B. Some of the member functions are printed and explained below. The functions "checkInfo" and "checkInputs" consider about error checking. They will be explained in chapter 7.

Constructor Pla

The constructor of the Pla is very simple. The Pla has to be initialized empty.

_PLa :: Pla(void) : Streamable(), itable(NULL), otable(NULL) {
    info.Ctype = Platype;
} // constructor

Because there are no arrays to point to, the pointers itable and otable are made null. Also the base of Pla, Streamable, is initialized empty by ": Streamable()".

We also want to store the type of circuit. This is done in info.Ctype.

Destructor ~Pla

The destructor is also very simple. It just calls the function "flush". This function clears the object.

_Pl a :: ~Pla(void) { flush(); }

inputFrom

Assuming that the input doesn't include any errors, InputFrom is based on the following simple algorithm. Chapter 7 will explain the error detection and the error handling in this function.

  Clear the object
  Read information from header
  Initialize the object
  While the end of the input is not reached
CLASSES FOR LOGIC CIRCUITS

Read input and output
Add them to the appropriate arrays

This algorithm is used to implement inputFrom in the following way. Some parts (error checking) have been replaced by "..." to make the structure of the function more clear.

```cpp
void Pla :: inputFrom ( istream& in ) {
    cerr << "Reading input:
";
    flush();
    in >> info;
    ... info.Ctype = Platype;
    checkInfo();
    init();
    while (in.good()) {
        in >> ws;
        if (in.eof()) {
            break;
        }
        else if (in.peek() == '#') {
            in.ignore(MAXINT, '\n');
            info.linenumber++;
        }
        else if (in.peek() == '.') {
            in.ignore(); // '.'
            if (in.get() == 'e') break; // end is reached
        }
        else {
            strstream linestream;
            toLine(linestream,in);
            info.linenumber++;
            linestream >> ws; // skip whitespace
            ... Term * I = new Term(info.ninputs);
            Output * O = new Output(info.noutputs);
            linestream >> *I;
            ... linestream >> *O;
            ... itable->add(*I);
            otable->add(*O);
            ... }
    }
    // while
    } // inputFrom
```

First the object is cleared by "flush". Then the header of the PLA file is read by "in >> info". Next the while loop is entered. As long as the input is good, i.e. no end of file, the inputs and outputs are read by "linestream >> *I" and "linestream >> *O". This linestream is used because we want to read the input file line by line. After the input and output have been read, they are added to the appropriate array by "add".

printOn

```cpp
void Pla :: printOn ( ostream& out ) const {
    if (valid()) {
        info.printOn(out);
        for (int i = 0; i < itable->getItemsInContainer(); i++) {
            out << (*itable)[i] << " "
        }
    }
} // printOn
```
Because of the use of "printOn" and the [ ] operator that is defined on TermArray printing becomes very simple. Now we can just print the header by one command, followed by the description lines that exist of an input/output pair indexed by i. Of course it is checked first if the Pla can be printed, if it is valid.

**establishIrrelevant**

In 5.1.2 we have seen the purpose of "establishIrrelevant". It has to substitute all bits that mean irrelevant for some type of PLA by real irrelevants. This can be done in the following way.

```cpp
void Pla :: establishIrrelevant(void) {
    cerr << "Substitute irrelevants ...
";
    if (valid()) {
        switch (info.Otype) {
            case f : subst(dontcare, irrelevant); break;
            case fd : subst(zero, irrelevant); break;
            case r : subst(dontcare, irrelevant); break;
            case dr : subst(one, irrelevant); break;
            case fr : subst(dontcare, irrelevant);
        }
        cerr << "Ok\n\n";
    } else cerr << "Pla not valid\n";
    // establishIrrelevant
}
```

The output type of the PLA is stored in info, in the variable Otype. Depending on Otype the appropriate call to the function "subst" is done. As we have seen is "subst" a private function of the Pla class, that substitutes all bits of a certain value by another value in all terms of otable. One remark has to be made concerning the "switch" statement. This statement works approximately like "case" in Pascal. Normally each case has to be terminated by "break". After "break" the program continues after the '{' that closes the switch-block. If no "break" is used, the program continues at the following line and executes also the next command. So if Otype is "f", the function "subst" is called twice. Both don't cares and zero's are replaced by irrelevants. In this case this is the correct procedure.

**removeIrrelevant**

This function does the complementary of the above function. Again the function "subst" is used.

```cpp
void Pla :: removeIrrelevant(void) {
    cerr << "Removing irrelevants ...
";
    if (valid()) {
        switch (info.Otype) {
            case f:
            case fd: subst(irrelevant, zero); break;
            case r:
            case dr: subst(irrelevant, one); break;
            case fr: subst(irrelevant, dontcare);
        }
        cerr << "Ok\n\n";
    }
```
The conditions for a valid PLA are that the header information has to be valid and that both input and output arrays exist. This is coded in the following way.

```cpp
bool PLA :: valid(void) const
    { return (info.valid() && itable && otable); }
```

### delVar

This function deletes a variable from the PLA. That variable is specified by a number. The variables are numbered as follows. If there are n input variables and m output-variables, the inputs are numbered from 0 to n-1 and the outputs are numbered from n to n+m-1. After distinguishing if the specified variable is an input or an output, the variable has to be removed from all inputs or outputs. In paragraph 3.1 we have seen that we can use "forEach" for such an operation. This is done in the function delVar.

```cpp
void PLA :: delVar(int var) {
    cerr << "Deleting variable " << var << " ...
";
    if (valid()) {
        if (var < 0) { // do nothing
            } else
            if (var < info.ninputs) {
                itable->forEach(delete_var, &var);
                info.ninputs--;
            } else
            if (var < info.ninputs + info.noutputs) { // var -= info.ninputs;
                otable->forEach(delete_var, &var);
                info.noutputs--;
            } cerr << "Ok
";
        } else cerr << "Pla not valid\n\n";
    } // del_var
```

The number of input- and output-variables are respectively stored in "ninputs" and "noutputs" in the object "info". With these parameters it is determined if "var" is an input or an output. Then "forEach" is called with "delete_var" as the action function and "&var" as argument. We can't use "var" itself, because the argument has to be of the "void*" type. With "&" the address of "var" is taken, so "&var" is a pointer. Now we need a function "delete_var". This function can be found also in the file "l_pla.cpp" and is printed below.

```cpp
static void delete_var(Object& o, void* v) {
    Term& term = (Term&) o;
    for (int i = *(int*)v+1; i < term.nelems; i++)
        term.set_var(i-1, term.get_var(i));
    term.nelems--;
}
```
As we see “delete_var” has the right arguments for an action function, according to what was stated in paragraph 3.1. The function however has to remove a variable, specified by a number, from a Term object. Hence the arguments have to be converted to the proper type. This is done by putting for example “(Term&)” in front of “o”. “o” is of the type reference to Object and is now converted to reference to Term. Pointer “v” is converted in the same way from “void*” to “int*”. The rest of the function is very simple. All variables from place v+1 are copied one place forward by using “set_var” and “get_var” to set and to get the values of the specified variables.

**init**

We have seen that the constructor doesn’t initialize the arrays, because at time of declaration of the Pla object, the array sizes are still unknown. After reading the file header of the PLA file, the arrays can be initialized. This is done by “init” as was showed in function “inputFrom”.

```cpp
void Pla :: init(void) {  
    itable = new TermArray(info.ntransitions, 0, info.transitiondelta);  
    otable = new TermArray(info.ntransitions, 0, info.transitiondelta);  
} // init
```

The statement “new” allocates space for the objects to which “itable” and “otable” point to. At the same time these objects are initialized because “new” calls the constructor “TermArray”. The class TermArray is derived from Array and the meaning of the arguments of the TermArray constructor is the same as for the Array constructor. “ntransitions” is the number that was specified after the “.p” command in the PLA file. This command is optional. So if no number was specified, a constant predefined value is taken. Because there is a risk that this predefined value is too small, a deltavalue is set for the TermArray. The predefined values are set in the function “checklnfo” that will be explained in chapter 7.

**flush**

When the Pla object isn’t used anymore, it has to be cleared. That means that the space that was used for the arrays has to be freed explicitly. After all “itable” and “otable” are pointers, so destroying these variables doesn’t deallocate the memory space.

```cpp
void Pla :: flush(void) {  
    info.flush();  
    if (itable) delete itable;  
    itable = NULL;  
    if (otable) delete otable;  
    otable = NULL;  
} // flush
```

Deallocation is carried out by “delete”. Because this has only to be done if the arrays exist (has been initialized), this is checked first. If an array wasn’t initialized, its pointer is equal to null. It has the integer value 0.

**subst**

“subst” has to substitute some specified value by some other specified value in all output patterns. This could be done using “forEach”, as in “delVar”. Then however we would need an extra action function and we would have to do some type conversion
in that action function. Because the outputs are in an array, there is a simpler way to iterate the objects, as is shown below.

```c
void Pla :: subst(bit oldb, bit newb) {
    for(int n = 0; (*otable)[n] != NOOBJECT; n++) {
        for (int i = 0; i < info.noutputs; i++)
            if ((*otable)[n].get_var(i) == oldb)
                (*otable)[n].set_var(i, newb);
    }
} // subst
```

The array is iterated by a for loop. This loop ends if the [] operator returns "NOOBJECT". "NOOBJECT" is a code that of course means that there is no object at the specified index. Because the outputs were added consecutively, no object can be interpreted at the end of the array.

### 5.2 Classes for sequential machines

#### 5.2.1 A model

A sequential machine can be defined in a mathematic way (see for example [4]) as a five-tuple:

$$M = (I, S, O, \delta, \lambda)$$

with

- $I$ - a set of inputs
- $S$ - a set of states
- $O$ - a set of outputs
- $\delta : I \times S \rightarrow S$ - a transition function
- $\lambda : I \times S \rightarrow O$ - an output function (for a Mealy machine)

or

- $\lambda : S \rightarrow O$ - an output function (for a Moore machine)

In the literature, the output function for Mealy machines is also called $\beta$. To distinguish the two different kinds of output functions we'll consequently use $\beta$ for the Mealy output function in this report. From the literature we also know a third kind of sequential machine: the statemachine, which has no output function. This machine is defined as

$$M = (I, S, \delta)$$

From this definition we see that the statemachine is a basis for both Mealy and Moore machines.

The mathematical model can directly be mapped on a class structure. We need a class StateMachine that has $I$, $S$ and $\delta$ as data members. From this class the classes Mealy and Moore are derived. In these classes the output functions $\beta$ or $\lambda$ are added. The set of outputs $O$ is common for Mealy and Moore machines. Therefore this set was also made a member of StateMachine. This is however against the definition of a statemachine. This may be a problem in the future.
Now we have a class structure, we can think of the operations that we want to define on sequential machines. It depends on the implementation of the function if it will become a member of StateMachine or a member of both Moore and Mealy. If the function does not concern the output function it can be a member of StateMachine. Else it should be in both Moore and Mealy with different implementations.

- Read the description of the sequential machine from a file in KISS format. The format depends on the type of machine. So the function must be member of both Moore and Mealy class.
- Print the description of the sequential machine out in KISS format. This function is also different for Moore and Mealy.
- Test if the object is a valid sequential machine. This check is different for Moore and Mealy machines.
- Test if the inputs cover the complete boolean space. If this is not the case the transition and output function of the sequential machine are not fully specified. These functions should be fully specified. So we want to check.
- Interpret the output patterns as specified by the "type" directive. This is an operation on the output set, which is a member of Statemachine. So the function should be a member of StateMachine.
- Make the input set multiple exclusive. The input set is a set of cubes, representing input terms of the circuit with a certain number of input variables. A set of cubes is called multiple exclusive if there is no cube (with the same number of variables) that is covered by two cubes in the set. Such a set is also said to have no overlapping elements. There aren't two cubes that have a common part. For example the cubes 0-1 and -11 are overlapping. The common part is 011. A multiple exclusive input set has several advantages in further optimization processes. For example such an input set can be considered as a set of symbolic inputs. This function changes the input set and also the δ and β functions. Because of the last it has to be a member of Moore and Mealy.
- Complete the transition and output functions. A KISS file doesn't specify a complete sequential machine. Look for example at the two following lines.

10-- s1 s3 001
10-1 s2 s4 010

These two lines mean that δ(s1, 10--) = s3 and δ(s2, 10-1) = s4. So nothing is specified for input pattern 10-1 and current states other than s2. These transitions are left don't care. We see however that 10-- covers 10-1, so δ(s1, 10-1) should be s3. This operation should be also performed on the output function. This is called completing. For a Mealy machine δ and β are completed. For Moore machines only δ is completed.

Of course there are many other functions to define on sequential machines, but they are left for future implementation. At this moment it is important to create a basis for the library. However there is one concept we want to add. With the above structure (a δ and a β or λ function) we can find a next state and an output according to the specified current state and input very easy. Finding current state/input pairs that imply a certain next state or output is not so easy. The complete function has to be scanned on that specified next state or output. This situation may occur in future applications. Therefore it was decided to add relations that are the inverse of the transition and output functions to the classes StateMachine, Moore and Mealy. The δ⁻¹ relation re-
turns a list of input/current state pairs for a given next state. The $\beta^{-1}$ relation returns a
list of input/current state pairs for a given output and $\lambda^{-1}$ returns a list of current states
for a given output. We need special functions to create each of these inverse rela-
tions. Functions to print them are also provided.

5.2.2 The Mealy class

From the model discussed above, the following C++ class was created. This class can
be found in the file "_seq.h".

class Mealy : public StateMachine, public Streamable {
private:
    BetaFunction beta_function;
    InverseBeta inverse_beta;
public:
    Mealy(void); // constructor
    Mealy(Mealy& ); // copy constructor
    ~Mealy(void); // destructor
    void inputFrom(istream& ); // read Mealy mach in
    void printOn(ostream& ); // print Mealy mach in
    bool valid(void) const; // return if Mealy
    void checkFunctions(void); // check if function
    void completeFunctions(void); // completes output-
    void multipleExclusive(void); // make inputs multiple
    void establishInverseBeta(void); // creates inverse beta
    void printInverseBetaOn(ostream& ) const; // print Inverse beta
private:
    void init(void); // allocate space
    void flush(void); // destroy data
    bool checkInfo(void); // check info after
    int mergeFunctions(int i, int j); // merge delta and beta
    int countTransitions(void) const; // count the number of
}; // class Mealy

The model that we created above is mapped directly on the class structure. We see
that indeed Mealy is derived from StateMachine and also from Streamable. This is
done exactly the same way as we did with the Pla class (paragraph 5.1.2). The data
members are object for $\beta$ and $\beta^{-1}$. An interesting detail is that an extra constructor and
a function "checkFunctions" are provided. The extra constructor copies another Mealy
machine. It creates a new Mealy object and initializes it with the values of the machine
to be copied. The function "checkFunctions" checks if the machine is specified cor-
rectly, if there are no two different next states defined for the same input/current state
pair. This is also done automatically by the functions "completeFunctions" and
"multipleExclusive". If one of these functions is used, "checkFunction" is not needed
anymore. Therefore the function is not called automatically after reading a KISS file and
the user has to specify explicitly if he wants to use this function. So it has to be a
public member, and not private as the corresponding function checkInputs in the Pla
class.
5.2.3 The Moore class

The Moore class is very much the same as the Mealy class, as can be seen below. It provides the same kind of functions. Only now object for $\lambda$ and $\lambda^{-1}$ are the data members.

class Moore : public StateMachine, public Streamable {
private:
    LambdaFunction labda_function;
    InverseLambda inverse_labda;
public:
    Moore(void); // constructor
    Moore(Moore&); // copy constructor
    ~Moore(void); // destructor
    void inputFrom(istream&); // read Moore machine in
    void printOn(ostream&); // print Moore machine in
    bool valid(void) const; // return if Moore machine is valid
    void checkFunctions(void); // check transition function
    void completeFunctions(void); // complete transition function
    void multipleExclusive(void); // make inputs multiple exclusive
    void establishInverseLabda(void); // create inverse labda
    void printInverseLabdaOn(ostream& out) const; // print inverse labda
}

private:
    void init(void); // allocate space
    void flush(void); // destroy data
    bool checkInfo(void); // check info after input
    int countTransitions(void) const; // count the number of transitions when printed
}; // class Moore

5.2.4 The StateMachine class

StateMachine is the base class for Moore and Mealy. It contains objects for I, S, O, $\delta$ and $\delta^{-1}$.

class StateMachine {
protected:
    InputSet input_set;
    OutputSet output_set;
    StateSet state_set;
    DeltaFunction delta_function;
    InverseDelta inverse_delta;
    GeneralInformation info;
public:
    StateMachine(void); // constructor
    StateMachine(StateMachine&); // copy constructor
    ~StateMachine(void); // destructor
    virtual void inputFrom(istream&); // see derived classes
    virtual void printOn(ostream&); // see derived classes
    bool validStateMachine(void) const; // return if Mealy machine is valid
    bool checkDeltaFunction(void) const; // check on state collision
CLASSES FOR LOGIC CIRCUITS

```cpp
void establishIrrelevant(void); // substitute '=' in
   // outputs
bool fullySpecified(int); // checks complete covering
virtual void multipleExclusive(void) = 0; // see derived classes
void establishInverseDelta(void); // creates inverse delta
   // relation
void printInverseDeltaOn(ostream & ) const; // print Inverse
   // delta
protected:
void initStateMachine(void); // allocate space
void flushStateMachine(void); // destroy data
void completeDeltaFunction(void); // completes delta function

}; // class StateMachine
```

Of course the information from the header of a KISS file has also to be stored. Because the header of a PLA file looks almost the same as the header of a KISS file, the same class (GeneralInformation) is used for this.

The StateMachine class is made pure virtual. This means that objects of the type StateMachine can't be declared. In the future there may be a need for objects of StateMachine. Then the definitions for printOn, InputFrom and multipleExclusive have to be changed.

5.2.5 Implementation of the Mealy class

The implementation of the member functions of the Mealy class can be found in the "I_seq.cpp" file. Some of these functions, like the constructor and the destructor, don't differ very much from the corresponding functions for the PLA class. They aren't explained below. The code of these function can be found in appendix B. Only the interesting functions are discussed here. "checkFunctions" is explained in chapter 7.

**InputFrom**

Assuming that the input doesn't include any errors, InputFrom is based on the following simple algorithm. Chapter 7 will explain the error detection and the error handling.

- Clear the object
- Read information from header
- Initialize the object
- As long as the end of the input is not reached
  - Read input, current state, next state and output
  - Add them to the appropriate set
  - Set δ and β functions

This algorithm is used to implement inputFrom in the following way.

```cpp
void Mealy:: inputFrom( istream& in) {
    char CS[80], NS[80];
    String *str;
    int i, cs, ns, o;
    stringstream msg; // stream for error messages
    cerr << "Reading input:\n";
    ... flush();
    in >> info;
    bool ok = in.good();
    info.Ctype = Mealytype;
    checkInfo();
```
init();
while (in.good()) {
    in >> ws; // skip whitespace
    if (in.eof()) {
        break;
    }
    else if (in.peek() == '#' || in.peek() == '"') // skip comments
        in.ignore(MAXINT, '\n');
    else if (in.peek() == '.') {
        in.ignore(); // '.
        if (in.get() == 'e') break; // end is reached
    }
    ...
}
else {
    stringstream linestream;
toLine(linestream,in);
    info.linenumber++;
    linestream >> ws; // skip whitespace
    Term * I = new Term(info.ninputs);
    Output * O = new Output(info.noutputs);
    linestream >> *I;
    ...
    linestream >> CS >> NS;
    ...
    linestream >> *O;
    ...
    i = input_set.addTerm(*I);
    ...
    o = output_set.addOutputTerm(*O);
    if (state_set.isDontcare(String(CS))) {
        ...
        ns = state_set.addNextState(*str);
        for (int s = 0; s < info.nstates; s++) {
            delta_function.set(i, s, ns);
            beta_function.set(i, s, o);
        }
    }
    else {
        str = new String(CS);
        ...
        cs = state_set.addState(*str);
        str = new String(NS);
        ...
        else {
            ns = state_set.addNextState(*str);
            delta_function.set(i, cs, ns);
            beta_function.set(i, cs, o);
        }
    }
    ...
}
} // while

We see that reading the information in the file header is done again in >> info. Again the file is split into lines by the function toLine. The input and output patterns and states are read from this new stream.
First the input and output patterns are added to the input and output sets. Then the current state is checked. If the current state is specified don’t care this means that for the given input the δ and β functions are equal for all current states. For example

"101 * s2 0001" means \( \forall s \in \mathbb{S}(\delta(s, 101) = s2 \land \beta(s, 101) = 0001) \)

This is implemented in a for loop. If the current state is not a don’t care, the current and next states are simply added to the state set and the appropriate values for the δ and β functions are set.

An interesting detail is that the elements of the different sets are indexed like the elements in an array. In fact the sets are arrays which have special properties to make them act like sets. The array structure provides direct access to the data in the sets, because of the use of indexes.

### printOn

```cpp
void Mealy::printOn(ostream& out) const {
    cerr << "Printing Mealy machine:\n";
    if (valid()) {
        GeneralInformation infoCopy(info);
        infoCopy.ntransitions = countTransitions();
        out << infoCopy;
        for(int s = 0; s < info.nstates; s++)
            for (int i = 0; i < input_set.nTerms(); i++) {
                StateIndex ns = delta_function.get(i, s);
                if (ns)
                    out << input_set[i] << " "
                        << state_set.StateName(s) << " "
                        << state_set.NextStateName(ns) << " ";
                OutputIndex o = beta_function.get(i, s);
                if (o) out << output_set[o] << endl; // output is specified
                else { Term T(info.noutputs); out << T << endl; } // output is don't care
            }
        out << ".e" << endl;
    } else cerr << "Mealy machine not valid\n";
}
```

We want to print the ".p" command in the header of the printed file, so we have to know the right number of transitions. This number may have changed because of some operations on the Mealy machine. Therefore the exact number is counted by a special function "countTransition". Now the file header can be printed, followed by the description lines of the machine. If we just printed all combinations of inputs and current states and their δ and β values, an enormous number of lines with a next state being don’t care would be found in the output. To avoid this it is checked if the δ function of each combination was defined. If not, nothing will be printed. Though combinations for which the next state explicitly was defined don’t care must be printed. Hence the δ function must have different values for undefined and don’t care, although the have the same meaning for the description of a machine. We will see that the value for undefined is taken zero. For outputpatterns the values undefined and don’t care are both zero. Don’t care outputpatterns aren't actually stored in the output set. If some output appears to be don't care, it can't be printed by printing some ele-
ment of the output set. Therefore a new Term object, which has only don't care bits, is declared. This Term is printed.

**completeFunctions**

In paragraph 5.2.1 the purpose of this function was explained. We have seen that if one input is covered by some other input the $\delta$ and $\beta$ functions of the covering input have to be merged with the functions of the covered input. The implementation of this is the following.

```cpp
void Mealy::completeFunctions(void) {
    if (valid()) {
        bool error = FALSE;
        for (int i = 0; i < input_set.nTerms(); i++) {
            for (int j = 0; j < input_set.nTerms(); j++)
                if (input_set[j] < input_set[i] && i != j) {
                    error |= mergeFunctions(j, i);
                }
        }
        if (error) Fatal("Machine description is incorrect");
    } else cerr << "Mealy machine not valid\n";
} // completeFunctions
```

We see that covering is checked with the $<$ operator. The function "mergeFunctions" merges both $\delta$ and $\beta$ function of input $i$ onto those of input $j$. It returns an error if the merging is not possible. This is the case for example if we try to merge the $\delta$ functions for inputs 1-0 and --0 if $\delta(s1, 1-0) = s2$ and $\delta(s1, --0) = s4$. This defines that $\delta(s1, 1-0)$ is both $s2$ and $s4$. In this report this error is called state collision, or output collision for output functions. It means that the description of the machine was incorrect. The result of mergeFunctions is "or"-ed with the old error by "|=". So if an error occurs error stays true. This denotes an incorrect machine description and the program is aborted in the function Fatal. We see that completeFunctions does implicitly the same as checkFunctions.

**multipleExclusive**

In paragraph 5.2.1 we have seen that this function has to generate a set of non-overlapping inputs that represents the original input set. This can be done by splitting each pair of overlapping inputs into three new parts: the overlapping part and the two parts that remain from excluding the overlapping part from the two original inputs. When we look at input terms as sets of minterms they cover, this would look like Figure 5.1.
Figure 5.1 Splitting two terms in three parts

If for example input 1 is 0-0-1 and input 2 is -1-11, then the overlapping part is 01011. Input 1 represents the set of minterms \{00001, 00011, 01001, 01011\}. If we remove 01011 from this set, the set \{00001, 00011, 01001\} remains. This is also represented by the set \{000-1, 0-001\}. In the same way for input 2 the set \{11-11, -1111\} remains. This procedure has to be followed for every possible pair of overlapping terms in the input set. The main problem is here that the input set grows during the process. This could mean that for every term we add to the input set we would have to check the complete set if there is a term that overlaps the new term. The following algorithm tackles this problem.

For the explanation of the algorithm we need the following definitions. In these definitions the lower case characters are representing cubes. The capitals represent sets.

**Cube**
A cube is the binary representation of a product term of a boolean function. If the boolean input space has n dimensions or, in other words, if the boolean function has n input variables, the cubes are elements of \{0, 1,-\}. They are n-dimensional vectors.

**Covering**
A n-dimensional cube \(c_1\) covers another cube \(c_2\) iff
\[ \forall_{1 \leq i \leq n} [c_1[i] = c_2[i] \lor c_1[i] = -^*] \]
This is annotated as \(c_2 < c_1\).

**\(B^n\)**
The set of all cubes of length n.

**Mutually exclusive**
Two cubes \(c_1\) and \(c_2\) are mutually exclusive iff
\[ \forall_{c \in B^n} \neg(c < c_1 \lor c < c_2) \]
This is annotated as \(me(c_1, c_2)\). If \(c_1\) and \(c_2\) are not mutually exclusive a cube \(c\) exists that is covered by both \(c_1\) and \(c_2\). This cube is the overlapping part of \(c_1\) and \(c_2\), \(c = c_1 \land c_2\).

**Multiple exclusive**
A set of cubes \(C\) is called multiple exclusive iff
\[ \forall_{a, b \in C} me(a, b) \]
This is annotated as \(me(C)\). By definition \(me(\emptyset)\) is true.

We can also define a cube \(c\) being multiple exclusive to a set \(B\): \(me(c, B)\). This is true iff
\[ \forall_{b \in B} me(c, b). \]
From the definition of \(*\) follows that
\[ c = x * y \Rightarrow [ me(x, B) \Rightarrow me(c, B) \land me(y, B) \Rightarrow me(c, B) ] \]
CLASSES FOR LOGIC CIRCUITS

\[ me(c, \varnothing) \] is defined true.

**Expansion**
The expansion of a cube \( c \) is the set \( C \) of cubes \( a \): \[ \{ a \mid a < c \} \]
This is annotated as \( \text{exp}(c) \). For example \( \text{exp}(-0) = \{ 1-0, 0-0, -10, -00, 100, 000, 010, 110 \} \).
The expansion of a set \( C \) of cubes, \( \text{exp}(C) \), is equal to \[ \bigcup_{c \in C} \text{exp}(c) \]

**\( l_{\text{pre}} \)** The set of cubes that is the input set of the Mealy machine before making it multiple exclusive.

**\( l_{\text{post}} \)** The set of cubes that is the input set of the Mealy machine after making it multiple exclusive.

Now we the postcondition of the function `multipleExclusive` has become very simple: \[ me(l_{\text{post}}) \land \text{exp}(l_{\text{post}}) = \text{exp}(l_{\text{pre}}) \]. In words this comes down to the following. At the end of the function the machine's input set is multiple exclusive and it represents the same set as the function started with.

This post condition can be reached by the following algorithm. \( S \) is still the state set of the machine.

```plaintext
Initialize \( M = \varnothing \) and \( N = l_{\text{pre}} \)
while \( \lnot (N = \varnothing ) \) {
  Initialize \( E = \varnothing \)
  Take some \( c \) from \( N \)
  \( A = N \setminus c \)
  Delete \( \delta(s,c) \) and \( \beta(s,c) \) for all \( s \in S \)
  while \( \exists a \in A \land \neg me(c,a) \) {
    \( B = A \setminus \{a\} \)
    Delete \( \delta(s,a) \) and \( \beta(s,a) \) for all \( s \in S \)
    compute cube \( x = c \ast a \)
    compute \( \delta(s,x) \) and \( \beta(s,x) \) for all \( s \in S \)
    compute set \( Y \) for which \( \text{exp}(Y) = \text{exp}(c) / \text{exp}(x) \)
    compute \( \delta(s,y) \) and \( \beta(s,y) \) for all \( y \in Y \) and \( s \in S \)
    compute set \( Z \) for which \( \text{exp}(Z) = \text{exp}(a) / \text{exp}(x) \)
    compute \( \delta(s,z) \) and \( \beta(s,z) \) for all \( z \in Z \) and \( s \in S \)
    \( E = E \cup Y \cup Z \)
    \( A = B \)
    \( c = x \)
  }
  \( N = A \cup E \)
  \( M = M \cup \{c\} \)
}
\( l_{\text{post}} = M \)
```

The inner while loop has the following invariant (inv1) for the cube sets.

\[ \text{inv1} = [ me(c,M) \land me(c,E) \land \text{exp}(\{c\} \cup M \cup A \cup E) = \text{exp}(l_{\text{pre}}) ] \]

This is proved in appendix A. The invariant (inv2) of the outer while loop is
inv2 = [ ∀me_M : me(m, N) ∧ me(M) ∧ \( \exp(M \cup N) = \exp(l_{pre}) \) ]

This is also proved in appendix A. When the loop condition \( N = \emptyset \) has become true, inv2 leads to the postcondition:

\[ \text{inv2 } \Rightarrow \\
\{ N = \emptyset \} \\
\text{me}(M) \land \text{exp}(M) = \text{exp}(l_{pre}) \Rightarrow \\
\{ l_{post} = M \} \\
\text{me}(l_{post}) \land \text{exp}(l_{post}) = \text{exp}(l_{pre}) \]

Of course \( \delta \) and \( \beta \) have to change from functions defined on \( l_{pre} \) to functions defined on \( l_{post} \). Therefore the functions for cubes C and a are deleted and replaced by the appropriate functions for cube x and the cubes in \( Y \) and \( Z \). This can easily be seen, so this isn't proved.

As we see, a lot of checks have to be done to make an input set multiple exclusive. This algorithm is efficient in two ways. In the first place it doesn't check cubes that are in the set \( E \). Second it does never check the cubes in set \( M \) again. So when a term of the input set is splitted into new terms which are added to the input set, the function doesn't have to check the complete set on multiple exclusivity to the new terms.

Now we come to the implementation of the above algorithm. As we have mentioned before, the sets are represented by arrays. Because of this we have direct access to the data by indexing. The sets \( M \), \( N \), \( A \), \( B \), \( E \), \( Y \), and \( Z \) are all stored in the same array. That is the array that stores the set of input terms \( \text{input}_\text{set} \). The elements at index 0..\( i-1 \) belong to \( M \), the elements in \( i..\text{limit}-1 \) belong to \( N \). The rest belongs to \( E \). The number of elements of \( \text{input}_\text{set} \) can be obtained by the member function "\( n\text{Terms} \)". A consists of the elements in position \( j..\text{limit}-1 \). This division can be seen in Figure 5.2.

The sets \( B \), \( Y \) and \( Z \) are not given explicitly.

![Figure 5.2 Division of input_set](image)

The implementation of the above algorithm is now as follows.
void Mealy::multipleExclusive(void) {
    cerr << "Making circuit multiple exclusive ";
    if (valid()) {
        int error = 0;
        Term *T2;
        for (int i = 0; i < input_set.nTerms(); i++) {
            int limit = input_set.nTerms();
            for (int j = i+1; j < limit; j++) {
                Term oldTi(input_set[i]);
                Term oldTj(input_set[j]);
                Term *Common = new Term(oldTi * oldTj);
                if (*Common != 0) { // not multiple exclusive
                    if (oldTj < oldTi) {
                        int n = 0;
                        while (n < info.ninputs) { // add non-overlapping
                            // terms
                            T2 = new Term(subtract(oldTi,oldTj,n));
                            int t2 = input_set.addTerm(*T2);
                            error |= mergeFunctions(t2,i);
                        }
                        input_set.setTerm(i,oldTj); // replace Input[i]
                        // by common term
                        error |= mergeFunctions(i,j);
                    }
                    else if (oldTi < oldTj) {
                        int n = 0;
                        while (n < info.ninputs) { // add non-overlapping
                            // terms
                            T2 = new Term(subtract(oldTj,oldTi,n));
                            int t2 = input_set.addTerm(*T2);
                            error |= mergeFunctions(t2,j);
                        }
                        error |= mergeFunctions(i,j);
                    }
                    else {
                        int n = 0;
                        while (n < info.ninputs) { // add non-overlapping
                            // terms
                            T2 = new Term(subtract(oldTi,*Common,n));
                            int t2 = input_set.addTerm(*T2);
                            error |= mergeFunctions(t2,i);
                        }
                        n = 0;
                        while (n < info.ninputs) { // add non-overlapping
                            // terms
                            T2 = new Term(subtract(oldTj,*Common,n));
                            int t2 = input_set.addTerm(*T2);
                            error |= mergeFunctions(t2,j);
                        }
                        input_set.setTerm(i,*Common); // replace Input[i]
                        // by common term
                        error |= mergeFunctions(i,j);
                        cerr << ".";
                        input_set.deleteTerm(j);
                        delta_function.deleteForInput(j);
                        beta_function.deleteForInput(j);
                        limit--;
                        j--;
                    }
                }
            }
        }
    }

    if (!error) cerr << "Result: Circuit is multiple exclusive\n\n";
    else cerr << "\nResult: Circuit is ambiguous\n\n";
}
else cerr << "Result: Mealy machine not valid\n\n";
The first while loop is implemented as a for loop in line 6. As we can see easily from Figure 5.2 the loop-condition \((N = \emptyset)\) equals \(i < \text{input\_set.nTerms()}\). \(i\) is initialised to 0 and so \(M = \emptyset\) and \(N\) is the complete input set. Then in line 7 \(E\) is initialised empty. The cube taken from \(N\) is always the cube at position \(i\). So in the second for loop \(j = i+1\) is initialised which comes down to making \(A = N / c\). Now this loop checks all cubes in \(A\) on being multiple exclusive with cube \(c\). If \(c\) is not multiple exclusive with some \(a\) in \(A\), we can distinguish three cases. First \(a < c\), second \(c < a\) and third none of both.

In the first case \(x\) is equal to \(a\). So now we don't have to compute \(x\), \(Z\) and the corresponding \(\delta\) and \(\beta\) functions. \(Y\) is computed and its elements directly added in the lines 17 to 19. Note that if some term is computed that is already in \(M\) or in \(A\), this term hasn't to be add to \(E\) anymore. This is done automatically by addTerm in line 17. For computing the \(\delta\) and \(\beta\) functions we use the function mergeFunctions again like in completeFunctions. After all we have a similar situation here. The function subtract creates the elements of \(Y\) one by one. How this works will be explained in chapter 6.

In the second case \(x\) is equal to \(c\). Now we have to compute only \(Z\). This is done in a same way as above, with subtract.

In the last case both \(Y\) and \(Z\) are computed in two while loops. Then \(x\) ("Common") is assigned to \(c\) (element at index \(i\)). After computing the sets \(Y\) and \(Z\) and updating the functions, term \(a\) can be deleted from \(N\) by deleteTerm. Also the \(\delta\) and \(\beta\) function for that term are deleted. Because \(N\) has decreased, the numbers \(j\) and \(\text{limit}\) also have to be decreased by one. The dot that is printed each time on the errorstream shows that the computer is still busy.

After the second for loop has been ended \(c\) is added to \(M\), because \(i\) is increased by one. \(\text{Limit}\) is set again so \(E\) is added to \(A\). After the first for loop is ended the input set is multiple exclusive, unless there was an error during merging the functions.

**establishInverseBeta**

This function creates the \(\beta^{-1}\) relation, a list of input/state pairs for each output. The implementation is very straightforward. First inverse_labda is initialised to the right size. Then the \(\beta\) function is scanned and each pair is added to the right list by the "set" memeberfunction of inverse_labda.

```cpp
void Mealy::establishInverseBeta(void) {
  if (valid()) {
    inverse_beta.init(output_set.nTerms());
    for (int i = 0; i < input_set.nTerms(); i++)
      for (int s = 0; s < info.nstates; s++)
        inverse_beta.set(beta_function.get(i,s),i,s);
  } else cerr << "Mealy machine not valid\n\n";
} // establishInverseLabda
```

**printlnverseBetaOn**

This function prints the \(\beta^{-1}\) relation, if one is present. Besides it shows a way to work with the inverse\_labda object. This object consists of an array of lists. These lists contain the input/state pairs for each outputpattern. If we want to work with lists, we have to create an object ListIterator (see paragraph 3.2). This object holds the current position in the list.

```cpp
void Mealy::printlnverseBetaOn(ostream& out) const {
```
if (inverse_beta.valid()) {
    InputIndex i;
    StateIndex cs;
    bool EndOfList;
    for (int o = 0; o < output_set.nTerms(); o++) {
        ListIterator &it = inverse_beta.initIterator(o);
        out << output_set[o] << " is output for following input-state pairs: " << endl;
        EndOfList = inverse_beta.get(it,i,cs);
        while (!EndOfList)
            out << "{" << input_set[i] << "," << state_set.StateName(cs) << "}";
        EndOfList = inverse_beta.get(it,i,cs);
        out << endl;
        delete it;
    }
    else cerr << "Create inverse beta first\n\n";
} // printInverseLabdaOn

We see that for each outputpattern in the output set a ListIterator is created with the
function initIterator. Then pairs are read from the list one by one, until the list is ended.
Note that we don't obtain the real inputs and states, but the index i in the input set and
index cs in the state set.

mergeFunctions

For Mealy machines we often want to do the same operation merge on the $\delta$ and the
$\beta$ function, for example in multipleExclusive. Therefore this function was created. It
merges the $\delta$ function for some inputs i and j and does the same for the $\beta$ function.
Besides it does some error checking. If the functions for i and j can't be merged, it re­
ports an error and returns this.

int Mealy::mergeFunctions(int i, int j) {
    int errorstate = 0;
    int error = 0;
    errorstate = delta_function.merge(i, j);
    error = errorstate;
    if (errorstate) {
        stringstream msg;
        msg << "collision in deltafunction for inputs " << input_set[i] << " and " << input_set[j]
             << " at state " << state_set.StateName(errorstate-1);
        Error(msg);
    }
    errorstate = beta_function.merge(i, j);
    error = errorstate;
    if (errorstate) {
        stringstream msg;
        msg << "collision in betafunction for inputs " << input_set[i] << " and " << input_set[j]
             << " at state " << state_set.StateName(errorstate-1);
        Error(msg);
    }
    return error;
} // mergeFunctions

countTransitions

As we have seen in printOn, we want to print the Mealy machine in KISS format.
Therefore we want to specify the right number of transitions after the .p directive. Be­
cause the number of transitions may be unknown or changed during the program, it is
recounted at the moment of printing. In printOn we have also seen that a line is printed if there is a next-state specified in the $\delta$ function for some input and current state. This is also used for counting the number of transitions.

```cpp
int Mealy::countTransitions(void) const {
    int count = 0;
    if (valid()) {
        for(int s = 0; s < info.nstates; s++)
            for (int i = 0; i < input_set.nTerms(); i++) {
                StateIndex ns = delta_function.get(i,s);
                if (ns) count++;
            }
    }
    return count;
}
```

5.2.6 Implementation of the Moore class

A Moore machine looks very much the same as a Mealy machine. The only difference is the output function. Because of this the implementation of the Moore class is also very much the same as the implementation of the Mealy class. It would be exaggerated to describe all member functions in detail. So only the differences between the Mealy and the Moore implementation will be explained. If necessary, the function will be printed in the text, else they can be found in appendix B.

InputFrom

In paragraph 4.2 we have seen the differences and similarities between descriptions for Mealy and Moore machines. The input procedures are based on the same principles. First the file header is read, then the description lines are read one by one and the information is stored. Moore machines have two kinds of description lines. One for the $\delta$ function and another for the $\lambda$ function. In a $\lambda$ (output) description line the next state has to be "void". This is checked immediately after the input, current state, next state and output have been read. If the next state is indeed equal to "void", then the output index is stored in the $\lambda$ function at the index of the specified current state. In this case a warning is given when the input pattern isn't a full don't care term. In other versions of the KISS format it is possible to specify some Mealy-like Moore machine. Which means that the output depends on the current state and the input. In this program only pure Moore machines are allowed, so inputs should be don't care. If the next-state is not "void", the $\delta$ function is stored just as in the input function for Mealy machines. The only difference is of course that the output is not stored.

```cpp
void Moore :: inputFrom( istream& in) {
    char CS[80], NS[80];
    String * str;
    int i, cs, ns, 0;
    strstream msg; //stream for error messages
    cerr << "Reading input:\n";
    ...;
    flush();
    in >> info;
    bool ok = in.good();
    info.Ctype = Mooretype;
    checkInfo();
    init();
    while (in.good()) {
        in >> ws; // skip whitespace
```
if (in.eof()) {
    break;
}
else if (in.peek() == '#') || in.peek() == 'n')  // skip comments
    in.ignore(MAXINT, 'n');
else if (in.peek() == '.') {
    in.ignore(); // '
    if (in.get() == 'e') break;    // end is reached
    else {
        ...
    }
}  
else {
    stream linestream;
    toLine(linestream, in);
    info.lineno++;
    linestream >> ws;    // skip whitespace
    Term * I = new Term(info.ninputs);
    Output * O = new Output(info.noutputs);
    linestream >> *I;
    ...
    linestream >> CS >> NS;
    linestream >> *O;
    ...
    if (state_set.isVoid(NS)) {
        str = new String(CS);
    ...
    } else {
        cs = state_set.addState(*str);
        o = output_set.addOutputTerm(*O);
        Term DC(info.ninputs);
        // create a don't care term
        ...
        labda_function.set(cs, o);
    }
    }  
else if (state_set.isDontcare(String(CS))) {
        i = input_set.addTerm(*I);
        ...
        str = new String(NS);
        ...
        else ns = state_set.addNextState(*str);
        for (int s = 0; s < info.nstates; s++) {
            delta_function.set(i, s, ns);
        }
}]  
else {
    i = input_set.addTerm(*I);
    ...
    str = new String(CS);
    ...
    else cs = state_set.addState(*str);
    str = new String(NS);
    ...
    else {
        ns = state_set.addNextState(*str);
        delta_function.set(i, cs, ns);
    }
    ...
    }
    }
}  // while
...
} // inputFrom
printOn

Printing a Moore machine is split in three parts. First the header information is printed, then the δ function and at last the λ function. If an output isn't specified for a certain state, the get function returns the value 0 for o. In this way only the specified values of the λ function are printed.

```cpp
void Moore :: printOn( ostream& out) const {
    cerr << "Printing Moore machine:\n";
    if (valid()) {
        GeneralInformation infoCopy(info);
        infoCopy.ntransitions = countTransitions();
        out << infoCopy;

        for(int s = 0; s < info.nstates; s++)
            for (int i = 0; i < input_set.nTerms(); i++) {
                StateIndex ns = delta_function.get(i,s);
                if (ns) { // not undefined
                    out << input_set[i] << " ";
                    out << state_set.StateName(s) << " ";
                    out << state_set.NextStateName(ns) << " ";
                    out << output_set.zeroOutputTerm() << endl;
                }
            }
        for (s = 0; s < info.nstates; s++) {
            OutputIndex o = lambda_function.get(s);
            if (o) { // output is specified
                out << Term(info.ninputs) << " " << state_set.StateName(s)
                    << " " << output_set[o] << endl;
            }
        } else cerr << ".e" << endl;
    } else cerr << "Moore machine not valid\n\n";
    // printOn
}
```

completeFunctions

Because the λ doesn't depend on inputs, only the δ function has to be completed. This is done by the function completeDeltaFunction, which is a member of StateMachine.

multipleExclusive

This function is almost exactly the same as for Mealy machines. The only difference is that MergeFunctions, which merges both δ and β functions, is replaced by delta_function.merge. After all the λ function is independend of the input set.

establishInverseLambda

This function creates for each output pattern a list of states that implies that output. With this inverse λ relation, we can find states that generate certain outputs very fast. The algorithm is exactly the same as for establishInverseBeta.

printInverseLambdaOn

This function prints the λ⁻¹ relation. How this works can be read in the previous paragraph at the description of printInverseBetaOn.
countTransitions

printOn shows that a line of the δ function is printed if the next state is specified and a line of the λ function if the output is specified. These conditions are also used in countTransitions.

5.2.7 Implementation of the Statemachine class

The class StateMachine is the base class for the Moore and Mealy classes. It contains the input set, the output set, the state set and the δ function for both kinds of machines. All member functions, except fullySpecified, have equivalents in the Mealy and Moore classes. Explanation of these functions can be found in paragraph 5.2.5 "Implementation of the Mealy class".

fullySpecified

This function checks if the input set covers the complete boolean space. This is necessary for certain further processing on sequential machines. In other words we have to check if the input set covers all 2^n minterms if the machine has n inputs. For a fairly large machine that has for example 20 inputs, this would come down to checking 1,048,576 minterms. This is an unacceptable number, so we have to choose another method. We could create a set R of terms and initialize it with a don't care term. Now R covers the complete boolean space. Then we look at the first term of the input set and compute a number of terms so that these terms together with the input term cover R, while they have no overlap with the input term. Here we 'subtract' the input term from the don't care term. We delete the don't care term from R and replace it by the terms we computed. R is now the part of the boolean space that is not covered by the first input term. Then we take the second input term. Look if it has an overlap with some term in R and if so the common part is subtracted from that term in R. This is done for all terms in R. R is now the part of the boolean space that is not covered by the first two input terms. So we continue for all input terms. If at the end R is empty, then the boolean space is fully covered by the input set. If R is not empty, then it contains the terms that should be added to the input set. To get this algorithm more clear, an example is given below.

The machine has 3 inputs and the following input set was specified \{ 10-, -11, 111, 1-0, --0 \}. The algorithm starts with R = \{ -- \}. In table 1 the progress of the algorithm is shown. The first column shows R. The second shows if the input term overlaps an element of R. If a common term is given, the terms have an overlap. The third column shows the current input term and Result shows the result of subtracting the common part from the term in R.

Table 1 Example of algorithm of fullySpecified

<table>
<thead>
<tr>
<th>R</th>
<th>Common</th>
<th>Input term</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>10-</td>
<td>10-</td>
<td>-1-, 0--</td>
</tr>
<tr>
<td>-1-</td>
<td>-11</td>
<td>-11</td>
<td>-10</td>
</tr>
<tr>
<td>0--</td>
<td>011</td>
<td>00-, 00-</td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As we see 001 is not covered by the input set. The term marked with * isn't added to R because it is covered by 0-0 which is already in R. We can check this covering while adding a term. So this doesn't cost efficiency. We can go one step further and check if a term, that we want to add, can be combined with a term in R to make a new term (merging). For example 0-1 and 0-0 can be combined to 0--. We do this in order to make R as small as possible. The smaller R is, the faster the function progresses. We can never say how big R will grow. In practice it shows that R never grows very large.

Of course subtracting 0-0 from 0-0 (marked by **) has no result. So 0-0 is just deleted from R. In general we can say that terms in R, that are covered by input terms must be deleted from R. The algorithm as described above, was coded as below.

```cpp
bool StateMachine :: fullyspecified(bool DeleteOnError) {
    cerr << "Checking covering of boolean space:\n";
    if (validStateMachine()) {
        TermSet Ta(info.ninputs,8*info.ninputs,8*info.ninputs);
        Term * t = new Term(info.ninputs);
        Ta.addTerm(*t);
        for (int i = 0; i < input_set.nTerms(); i++) {
            Term* Ti; -
            Ti = &(input_set[i]);
            int limit = Ta.nTerms();
            for (int k = 0; k < limit; k++) {
                Term Tk(Ta[k]);
                Term T2(info.ninputs);
                Term T1(*Ti * Tk);
                if (Tk < *Ti) {
                    Ta.deleteTerm(k);
                    k--;
                    Emi t--;
                } else
                if (T1 != 0) {
                    cerr << '.';
                    Ta.deleteTerm(k);
                    k--;
                    limit--;
                    int j = 0;
                    Term * T2;
                    while (j < info.ninputs+1) {
                        T2 = new Term(subtract(Tk, T1, j));
                        Ta.merge_addTerm(*T2);
                    }
                } else {
                }
            }
        }
        int ok = (Ta.nTerms() == 0);
        if (ok) cerr << "\nResult: Complete boolean space is covered by inputs\n\n";
        else {
```
The parameter DeleteOnError determines if the machine has to be deleted, if it shows that the boolean space is not covered totally. If this parameter is set TRUE, the program stops here, else it continues even if the input set isn’t complete. The termset Ta is used for R and Ti is the current input term. We see that Ta is initialized with a don’t care term, because the Term constructor creates a term with only don’t cares. Then a loop is started, so that all terms in the input set are processed. Followed by a loop for each term in R. The results of processing one input term that are added to R, are added at positions numbered from the parameter limit. These results don’t need to be checked for that input. This is a similar situation as described for multiple Exclusive. Now it is checked if the input term covers the term of Ta. If so, this term is deleted. If not, it is checked if the two terms overlap and the new terms of R are computed by subtract. A dot is printed here to show that the program is busy. When all input terms are checked, Ta should be empty. If the number of terms in Ta isn’t zero, we know that the boolean space wasn’t covered completely. The terms that rest in Ta should be added to the input set.

**initStateMachine**

This function really initializes the data members ofStateMachine. Now the numbers of states and transitions are known, the sizes of input_set, output_set, state_set and delta_function can be estimated. This is necessary because these objects are based on arrays. An upperbound for the size of input_set, output_set and delta_function is the number of transitions, specified by the .p directive. At least when we are reading the machine from a file. When we are making a circuit multiple exclusive, we see however that the size of the input set increases dramatically. Therefore we have to specify a delta-value, so that the input_set and delta_function can grow if necessary. The meaning of the delta-value is explained in paragraph 3.1. The delta-value is specified in the variable transitiondelta of the info object.
6. SUPPORTING CLASSES

6.1 Introduction

In the previous chapter we have seen the implementations of the Pla, Mealy and Moore classes and their member functions. In these implementations many objects were used that are not standard C++ objects. These objects (their class definition and member implementation) will be discussed in this chapter.

First we will list the objects that were used in the previous chapter. For the Pla class were used objects of the classes TermArray, Term and GeneralInformation. The objects that were used in Mealy, Moore andStateMachine were of the classes

- InputSet,
- OutputSet,
- StateSet,
- DeltaFunction,
- BetaFunction,
- LambdaFunction,
- InverseDelta,
- InverseBeta,
- InverseLambda,
- GeneralInformation,
- Term.

These classes will be discussed one by one in the following paragraphs.

InputSet and OutputSet have many similarities, so they have a common base class, called TermSet. In this chapter the program code will not be inserted in the text anymore. The code can be found in appendix B.

6.2 InputSet

6.2.1 Class definition

As mentioned InputSet is fully based on TermSet. It has only two constructors and a destructor of its own. The rest of the member functions are those of TermSet

6.2.2 Member functions

The member functions are very simple. The InputSet has an unknown size at the time the constructor is called. Therefore, when this size is known (when the header of the description file has been read), the InputSet is initialised by the function init, which is a member of the base class TermSet.

Because the base class destructor is automatically called by the destructor of InputSet, we only need an empty destructor for InputSet.
6.3 TermSet

6.3.1 Class definition

The TermSet class represents a set of terms, or maybe better cubes. It is used for storing both inputs and outputs of logical circuits. This may look strange, but in fact both input and outputs are represented by strings of 0, 1, -, and -. Such an input or output can be stored in an object that is called "Term", which will be discussed later in this chapter. Sets of inputs and outputs are derived from TermSet.

Because computers can't store sets directly, it was chosen to store the set in an array. A list structure would also be possible, but this has several disadvantages. In an array we can directly access an element. In a list we have to scan the list to find an element. Besides each input and output has its own index number in the array to which can be referred to in for example the \( \delta \) and \( \lambda \) functions. The array is of the type TermArray, a special type of array that only contains Term objects. TermArray is derived from the Array class of the Borland Container Library, so that we can apply the Array member function on TermArray.

For TermSet we have the same initialization problem as we have for Pla, Moore and Mealy. At the time the circuit object and the input- and output sets are created, the size of the arrays is unknown. Therefore only a pointer to the TermArray is declared. Memory space can be freed and the array can be really initialized later by the member "init". A constructor that also really initializes the array is added, because it is needed in StateMachine::fullySpecified, as we have seen. This constructor and init free memory space for an array of length nt in which Terms of nbits bits can be stored. If delta is specified other than zero, the array can be expanded by the number of delta Terms.

The set can be manipulated in some different ways. Terms can be added by addTerm. This function returns the index where the Term was added. Terms can be deleted from the set by deleteTerm and the value of Terms can be changed or set by setTerm. The function merge_addTerm is added for use in StateMachine::fullySpecified. Its goal is to add terms to the set and obtaining the smallest possible set that covers the same part of the boolean space. So it looks for terms in the set that can be merged to the term that is to be added. If it finds one, the terms are merged and the new value is set. Terms that are covered by terms that are in the set, don't have to be added. If both situations are not the case the term is added to the set in a normal way. At last we have function subst that substitutes all bits of a certain value by bits of another value. This is needed to change the outputset for certain outputtypes. This subject is discussed at paragraph 5.1.3.

We also need functions that give information about the set. nTerms reports the size of the set and the [ ] operator returns the Term at the specified location.

zeroTerm returns a term that only consists of zeros. This is used to print Moore machines in KISS format. isDontcare checks if the argument is a don't care term. These function were made members of TermSet to have there names protected. It would be a better choice to make them members of the Term class. Above all they should be static members.
6.3.2 Member functions

The constructors and destructor are very simple and don't need any explanation. The function nTerms and the [] operator use standard functions of the Array class. subst is the same function as in paragraph 5.1.3. The other functions will be clarified shortly.

**init**

初始化函数分配一个大小为nt-1并从下标0开始的数组。该数组可以被扩展由delta。因此，所有的下标都在范围0到nt-1+delta。

**valid**

此函数返回指针Terms转换的bool类型。这意味着如果指针是NULL则返回FALSE，否则返回TRUE。默认构造函数使指针Terms NULL，因此当集未初始化时，TermSet不是有效的。

**zeroTerm**

此函数创建一个不要关心的Term，并逐位将所有位设置为零。

**addTerm**

如果数组不为空，则可以添加新的Term。在集中的每个元素只能出现一次。数组被扫描以查找新的Term是否已经在集内。如果这是情况，其数组下标被返回。否则，通过add函数添加新的Term。新的Term的位置现在是数组中的Term数减一。

**merge addTerm**

新的Term被逐个与集中的元素合并。这是由Term类的merge函数完成的。merge返回可以合并的两个Term。如果m不是zero，则函数add设置marked。然后检查新的Term是否被term t覆盖。如果这是情况，则marked也被设置。当所有Term在集内被检查时，marked也被设置。如果它是零，则这意味着新Term未被添加。因此，它必须以正常方式添加。else它被添加并且它的位置被返回。

**deleteTerm**

Term是删除的由Array成员函数destroy。它删除了在给定位的Term对象，并将实际的arraysize减一。所以，元素的下标由删除的Term也减少了。

**setTerm**

一个新Term *T是创建并复制了来自给定Term。这是必要的因为第二个参数不是Term而是其引用。如果 
Terms wasn't copied the set wouldn't have the Term of its own. And changing the term outside the set would result in changing its copy in the set too.

**isDontcare**

The Term DC is a term of nb don't cares. Term T is converted to a new Term: test. This is because T can also be of the type Output which is derived from Term. If this is the case the equality check always fails, because this also checks the types and Term is not equal to Output. Now that T is converted, the equality check works correct.

### 6.4 OutputSet

#### 6.4.1 Class definition

The output set has two differences to the input set. First the meaning of the bits in the "outputterms" depends on the type that was specified in the KISS file. We want to store this type in the object so that we can use it in functions as establishIrrelevant. So the function init also takes the OutputType as argument. Notice that if needed also a function removeIrrelevant could be added, just as in the Pla class.

The second difference is that we want to be able to mark output functions unspecified or don't care. We have seen the advantage of this in the printing function for Moore machines. The value for unspecified was chosen zero. It was chosen to store the Outputs in an array that starts at index 0, like all other arrays in the software library. This causes a problem. Therefore a new add function and a new [ ] operator are needed that deal with this problem. It had been better to store the Outputs in an array with lower bound 1.

#### 6.4.2 Member functions

The constructors (and init) and destructor are again very simple. EstablishIrrelevant is copied from the Pla class, so the only interesting functions are addOutputTerm and the [ ] operator.

**addOutputTerm**

If the given term is don't care, then 0 is returned and the term is not added. If it is not don't care, the term is added as normal and the location in the array + 1 is returned. For the user of OutputSet is seems that at location 0 a don't care term is stored and at the following locations the other terms.

**[] operator**

As we have seen above, the indexes of the outputs are increased by one compared to the real positions in the array. To get the correct outputs out of the set, the indexes have to be decreased by one.
6.5 StateSet

6.5.1 Class definition

The state set of a sequential machine stores the different states of the machine. In this implementation the names of the states are stored as strings in an array. An object of the type StringArray is used for this. The number of states is stored in ns. The declaration and initialization is done in the same way as was done in the TermSet class.

Because of possible undefined next states in the δ function of a machine, we have the same problem as for OutputSet. Besides we want to distinguish undefined next states from don't care next states. It was chosen to give undefined next states the value 0 and don't care states the value 1. For current states we don't have this problem. However we want to store both current and next states in the same set/array of course. So we need different functions for adding current states and next states. This is also the case for functions that return the statename that is at a certain index.

isDontcare and isVoid are used in the input functions for Mealy and Moore machines to determine if a certain state is a don't care state are are "void" state.

isFull and hasMember are also used in these input functions. They return if the StateSet is full and if a specified state is already in the set.

The function printOn is used for testing. It prints the complete StateSet.

6.5.2 Member functions

Most functions are trivial. Only the adding and retrieving functions will be discussed.

addState
This function should be used to add a current state. It just adds the state if it isn't in the set already and returns the location of the state.

addNextState
This function should be used to add next states to the set. If this next state is don't care, it isn't physically added to the set, only the index 1 is returned. Other states are normally added to the array, but the index that is returned is increased by two. It seems now that the set for nextstates has a don't care state at location 1 and the other states at the following locations.

StateName
To retrieve the name of a current state at a certain index, use StateName with that index.

NextStateName
Because next states are numbered different than current states, we need a special function for retrieving the name of a certain next state. If the specified location is 0, then the next state is undefined: "ud" is returned. If the location is 1, this means a
don't care state and "**" is returned. All other states can be found in the array at loca-
tions that are two less than the specified index.

6.6 DeltaFunction

6.6.1 Class definition

We can think of a δ function as a two dimensional array like in Table 2. Because of
the fact that we stored the states and inputs in arrays, we can implement this idea
rather easy.

Table 2 Example of a δ function

<table>
<thead>
<tr>
<th>Current state</th>
<th>state1</th>
<th>state2</th>
<th>state3</th>
<th>state4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101-1</td>
<td>state2</td>
<td>state3</td>
<td>*</td>
<td>state1</td>
</tr>
<tr>
<td>---1101</td>
<td>*</td>
<td>state2</td>
<td>state1</td>
<td>*</td>
</tr>
<tr>
<td>-01-01-</td>
<td>*</td>
<td>*</td>
<td>state1</td>
<td>state4</td>
</tr>
</tbody>
</table>

We can replace the inputs by their indexes in the InputSet and the current states by
their indexes in the StateSet. The next states can also be represented by indexes. So
the δ function can be implemented by a two dimensional array of indexes. The width
of the table is a fixed value, ns, that depends on the number of states in the machine.
The length of the table depends on the number of inputs in the input set. This number
grows while the machine is read from a KISS file and in functions like multipleExclu-
sive.

Except for the standard functions (constructor, destructor, init, flush, valid), we need
functions to manipulate the DeltaFunction object. We can set a certain next state for a
given input and current state with set. The function get returns the index of the next
state for a specified input/current state pair. The corresponding name of that state can
be get by using StateSet::NextStateName.

In several applications we need to delete inputs from the input set (think of multiple-
Exclusive). In that case we also want to delete the corresponding row from the δ func-
tion table. This can be done by deleteForInput. We could also think of deleting a col-
umn from the table. This wasn't needed until now, and therefore not implemented.

Also in multipleExclusive we saw the need for a function merge. This function merges
the δ functions of input i and j and places the result in row i. If the two rows can't be
merged it returns index of the state that causes this error increased by 1. So if it re-
turns 0, the two functions were merged succesfully. The member distinct checks if the
rows i and j could be merged. It returns the same as merge.

6.6.2 Member functions

The interesting functions are the following.
set
First an object nexts of the type State is created. Then it is checked if there is already a row for input i in the deltatable. If this is the case, nexts is added in that row at position cs. If the row doesn't exist, it is created, the next state is added in the right place and next the complete row is added to the delta table. For the row an object of the type StateArray is taken.

get
This function uses the [ ] operator that is defined on StateTable. This returns an object of the type StateArray, or row i in the table. On this array also a [ ] operator is defined that returns the index of the next state at the position cs.

merge
To get clear what merge does exactly an example is given in Table 3 and Table 4. This shows the result of merging rows i and j to row i. The inputs i and j are for example 100 and -0-. For the use of this function see paragraphs 5.2.1 and 5.2.5.

<table>
<thead>
<tr>
<th>Table 3 Delta table before merge</th>
<th>Table 4 Delta table after merge</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
</tr>
<tr>
<td>j</td>
<td>4</td>
</tr>
</tbody>
</table>

There are two possibilities for row i. The row is already in the table or it isn't. In the first case the next states of row j are copied to row i, only if the next states in row i and j are not conflicting. This means that for a certain current state one of the two next states has to be unspecified or don't care (value 0 or 1), or the two next states have to be the same. If this is not true for some state, the index of this state plus one is returned, so that the error can be reported.

If row i doesn't exist yet, it is created, filled with the elements of row j and added to the delta table.

If row j isn't in the delta table an error is reported.

distinct
This function does the same checks as merge does. Therefore if distinct returns no error, the two δ functions for inputs i and j are not conflicting.
6.7 BetaFunction

6.7.1 Class definition

The β function of a Mealy machine is very similar to the δ function. Especially now we use indexes for both next states and outputs in both functions. Therefore the implementation of the BetaFunction class is almost identical to that of the DeltaFunction class. Only now a two dimensional array of the type OutputTable is used.

6.7.2 Member functions

Of course also the member functions of BetaFunction have the same structure as the DeltaFunction members. As mentioned BetaFunction is based on the two dimensional array OutputTable. This table is build of rows of the type OutputIndexArray which consists of OutputIndexType objects.

6.8 LabdaFunction

6.8.1 Class definition

The λ function of a Moore machine can be seen as a one dimensional array. For each state an output is specified. In this implementation not the output itself but its index in the output set is stored in the λ function. The OutputIndexArray is used for this purpose. ns is the number of states and also the size of the OutputIndexArray labda. This array is initialized in init. The set and get functions are used of course to set the λ function to the correct values and to get these values back.

6.8.2 Member functions

The member functions are very simple. They aren't explained here.

6.9 InverseDelta

6.9.1 Class definition

In paragraph 5.2.1 the purpose of the δ⁻¹ relation was explained. This δ⁻¹ relation was implemented in the InverseDelta class. It is based on an object of the class IndexPairTable. This class implements the δ⁻¹ relation as an array of lists of index-pairs. These index-pairs represent the corresponding input and current state. So for each state there is a list of pairs of an input and a current state that have that state as next state in the δ function.

Besides the indispensable constructors, destructor and init, flush and valid functions, we need a function to build the δ⁻¹ relation and one to read information out of it: set and get. Next a function printOn was added, that was used for testing. The function initIterator creates an object ListIterator for the list that is in position s of the IndexPairTable array. As was described in 3.2, the list iterator is a kind of pointer that points to some element of a list. The iterator can be increased to go to the following elements of the list. This iterator is needed in get to read elements from the list. The
function get returns if the list, from which an element was read, is at its end. It also
returns the components of the indexpair in the InputIndex i and StateIndex cs.

6.9.2 Member functions

init
The array IndexPairTable is set to the right size. Then the array is filled with empty
lists of the type IndexPairList.

set
In 6.5 we have seen that the real next states are numbered from 2, because the num-
bers 0 and 1 were reserved for undefined and don't care. So if the index of the next
state nst is larger than one (not undefined or don't care) and it's not too large, the
given indexpair (i, cs) is added to the list that corresponds to nst. nst is decreased by
two, because the array in IndexPairTable is initialized with lowerbound 0.

initlterator
This function uses the standard initlterator function from the Borland Library.

get
From the declaration of get can be seen that the index of the next state isn't specified.
This isn't necessary because we use the Listlterator. This iterator points to some
element in some list. When we use the "current" member function of Listlterator we
get the element that the iterator points to. So we don't have to specify the list explic-
tely. The integer conversion of the Listlterator is used to test if the list is ended, if the
iterator is beyond the last element.

Notice that get only returns one pair of a list. To get the complete list, a loop has to be
used.

6.10 InverseBeta
The \( \beta^{-1} \) relation has exactly the same structure as the \( \delta^{-1} \) relation. Therefore the class
declaration and the member functions of InverseDela and InverseBeta are almost
identical. Only some different names are used.

6.11 InverseLabda

6.11.1 Class definition
The \( \lambda^{-1} \) relation for a Moore machine can be also implemented as an array of lists.
These lists don't consist of indexpairs like for \( \delta^{-1} \) and \( \beta^{-1} \), but of single indexes. It
shows for each output in the ouputset which state causes this output. Except for this,
the InverseLabda class is identical to the InverseDelta and InverseLabda classes.
6.11.2 Member functions

Also the member function are almost identical. The only difference is that the output index 0 is decreased by one in the functions get and set. This is because the outputs in the output set are numbered from 1. Undefined and don’t care outputs have the number 0.

6.12 GeneralInformation

6.12.1 Class definition

The class GeneralInformation stores mainly the information that is in the header of a PLA or KISS file. Because these file-headers are very similar, we can use the same object for both. In the class definition we can find the following variables. For each variable its PLA or KISS command is given.

Table 5 Explanation of variables in GeneralInformation

<table>
<thead>
<tr>
<th>Variable</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ninputs</td>
<td>.i</td>
<td>Number of input bits</td>
</tr>
<tr>
<td>noutputs</td>
<td>.o</td>
<td>Number of output bits/ -functions</td>
</tr>
<tr>
<td>ntransitions</td>
<td>.p</td>
<td>Number of transition/description lines</td>
</tr>
<tr>
<td>nstates</td>
<td>.s</td>
<td>Number of states</td>
</tr>
<tr>
<td>*phase</td>
<td>.phase</td>
<td>Phase of the output functions</td>
</tr>
<tr>
<td>*InputNames</td>
<td>.ilb</td>
<td>Names of inputbits</td>
</tr>
<tr>
<td>*OutputNames</td>
<td>.ob</td>
<td>Names of output bits/ -functions</td>
</tr>
<tr>
<td>Otype</td>
<td>.type</td>
<td>Type of interpretation of outputs (fd, fr, dr, etc)</td>
</tr>
</tbody>
</table>

The InputNames and OutputNames are pointers to arrays of strings. These arrays are initialized later. We have to chose this construction, because we don’t know the size of the arrays (ninputs and noutputs) at compilation time.

We also want to store some other values. First we have Ctype, which stores the type of the circuit. The type can be "Unknown", "Sequential", "PlaType", "Mooretype" and "Mealytype".

The purpose of transitiondelta has already been explained in paragraph 5.2.7 at the function initStateMachine.

Linenumber is included in GeneralInformation to keep track of the position in the KISS or PLA file while reading a machine. This linenumber is used in error messages, so that the user can find the errors fast.

The member functions inputFrom and printOn are, as their names say, used for reading the file header of a KISS or PLA file and for printing a KISS or PLA file header. Note that GeneralInformation is derived from Streamable, so we can use the operators << and >> for printing and reading the header information. To make the two I/O functions shorter, the private member functions were added.
6.12.2 Member functions

Constructor
The constructor calls flush to initialize all variables to 0.

valid
To have a valid machine the output type, the circuit type and the number of input- and outputbits have to be specified. So this is checked by the function valid.

inputFrom
inputFrom is based on the principle that all commands are at the beginning of the KISS/PLA file and that they start all with a '.'. Besides there is only one command allowed in each line. So the input is split into parts of one line. If this line starts with a '.', a command will follow. Else the command section is ended and the description of the circuit itself begins. Except for the error detection, the reading of the commands is actual very trivial. After all commands have been read, some checks are done. This and the error detection will be explained in chapter 7.

printOn
For the information of the user, the type of circuit and the value of transitiondelta are printed by printOn as comments. After that the other variables of GeneralInformation are printed in a KISS/PLA format.

readOutputTypeFrom/printOutputTypeOn
The only way to read and print the outputtype correct is to select the right type with an if-else construction.

readPhaseFrom
As we know from the description of the PLA format (4.1.1), .phase is followed by a number of 0's and 1's that is equal to the number of outputbits. Therefore we read noutputs characters from the inputstream, look if they are 0 or 1 and set the bits of *phase to the appropriate values with set_var. Because the bits in the inputstream may be seperated by whitespace (tabs and blanks), we read this whitespace before we read the character by in >> ws.

readInputNamesFrom/readOutputNamesFrom
The problem of reading the input- and outputnames is that they may take more than one line. In inputFrom we have splitted the inputstream into a stream of one line (linestream) and the rest of the inputstream. First names are read from the first stream until this stream is ended. When it appears that the next line doesn't begin with a dot, we know that more names have to follow. We can check this if we can peek at the first character of the rest of the inputstream. Therefore this stream is also passed to both functions. If there are more names on the next line, this line is splitted from the main input and the names on it are read. Then we look again to beginning of the next line.
6.13 StringArray/TermArray/OutputArray

StringArray and TermArray are respectively used to implement the StateSet and InputSet and OutputSet. OutputArray can be used in future applications to make arrays of Output objects. These three arrays are constructed in the same way. They are all derived from the Array class of the Borland Container Library. So the constructors call the standard Array constructor. All member functions of Array and its base classes can be used for the three arrays. We only redefine the [ ] operator, because we want the operator to return an object of the appropriate type and not of the Object type, as the [ ] operator in Array does.

6.14 StateTable/OutputTable

StateTable and OutputTable are used to implement the δ and β function of a Mealy machine. Both δ and β are functions of the input set and state set, so the tables are two dimensional arrays. We can take a row for each input and a column for each state as was shown in 6.6. Now we can also think of the table as an array of rows. The size of the rows is known because it is the number of states. The number of rows however isn't a fixed value. When a Mealy machine is read from an inputstream, each new input causes a new row in the δ and β function. Therefore both StateTable and OutputTable are arrays, based on the Borland Container class Array. StateTable is build of StateArrays and OutputTable is build of OutputIndexArrays. For the explanation of the member function see the previous paragraph.

6.15 StateArray/OutputIndexArray

These StateArray and OutputIndexArray are also based on the Array class. The Array class can only contain object of the type Object. Here we want to store indexes (which may be integers) in the arrays. This is not possible with Array. Therefore we have to create the classes State and OutputIndex that contain an index and are derived from Object. Because of this we have to adjust the [ ] operator. This operator should return an index. So we have to return only the data member of the State or OutputIndex object. This gives of course a problem, if there is no Object at the specified location. But naturally in this case 0 is returned which means that there is nothing specified.

Because StateTable and OutputTable are arrays of StateArray and OutputIndexArray, these objects have to be Object themselves. The fact that the arrays are derived from Array, makes them Objects. Array is indirectly derived from Object. The problem is now that the arrays are abstract classes, because the function hashValue in Object is a pure virtual function that is defined nowhere. We have to define hashValue to solve this problem. This function isn't used anywhere, so it may just return some number.

Both arrays were made streamable to make printing them easier. This printing is only used for testing.

6.16 IndexPairTable/StateListArray

In paragraphs 6.9 and 6.10 we have seen that δ⁻¹ and β⁻¹ are based on IndexPairTable. This object consists of an array of lists. For each next state and output we have a list of input/current state pairs that causes that next state or output. IndexPairTable is an array of IndexPairList objects. StateListArray is used for λ⁻¹. It is ap-
proximately the same as IndexPairTable, but is an array of another kind of list: StateList. Both tables are again based on Array.

6.17 StateList/IndexPairList

StateList and IndexPairList are based on the Borland Container class List. This means that all member function of List can be used for StateList and IndexPairList. It is again required to define the function hashValue to make StateList and IndexPairList also Objects. This is because StateListArray and IndexPairTable are arrays of them.

6.18 State/OutputIndex

As mentioned the state- and output indexes can't be stored directly in arrays of the Array type and in Lists. Only objects of the type Object can be stored in Array and List. The solution is very simple. The classes State and OutputIndex were created. They contain an index and are derived from Object. Object however is an abstract class. We don't want State and OutputIndex to be abstract classes, so we have to define the pure virtual functions of Object.

6.19 IndexPair

IndexPair contains as the name says a pair of indexes, an input-index and a state-index. IndexPair is constructed on the same principles as State and OutputIndex.

6.20 Term

The Term class and its base PackedArray existed already before the logic library was made. Therefore these classes will not be explained in detail. Only the definition of the Term class, the use of some of its public member-functions and friends will be discussed. During the design of the logic library it was necessary to add a function "subtract". This function will be discussed in 6.20.2.

6.20.1 Class definition

A term is in fact an array of the four bitvalues 0, 1, −, and −. To store this efficiently the base class PackedArray was used. This class uses just enough bits to store one value. In this application it uses two bits for one value. This is enough because there are four possibilities. With set_var and get_var we can modify and read a single bit-value in a term. minterm checks if the term, considered as an input term, is a minterm. This means it contains no don't cares. valid checks if a term is valid for further use. inputFrom and printOn are of course for reading and printing terms. The + operator gives the smallest term that covers both operands. The operator * computes the overlapping part of both operands, or the largest term that is covered by both operands. The < operator computes if the first operand is covered by the second. The function merge computes a term that covers both operands exactly, if possible. distinct returns if two terms can be merged. subtract can be used to compute such a set of terms so that the first operand covers exactly the second operand and all elements of the set. Besides we want these terms to be as large as possible. At last we have to define the pure virtual functions of Object.
6.20.2 Member function

subtract

To make the use of subtract more clear, an example follows. Suppose we describe a part of the three dimensional boolean space by term \( T_1 = -1- \). Now we want a description in terms of the part that remains after we delete term \( T_2 = 011 \) from this part. This situation is drawn in Figure 6.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6_1.png}
\caption{Graphic representation of boolean space}
\end{figure}

From this figure we can see that the remaining part can represented by the terms 11- and -10. These terms are the largest we can find. Now \( T_1 \) covers exactly the set \{011, 11-, -01\}. When we want to compute this, we have to replace the don't cares of \( T_1 \) one by one by the negation of the corresponding bit in \( T_2 \) if this bit is 0 or 1. Because the terms should be as large as possible, they should have as many don't cares as possible. Therefore we create terms that have one don't care less than \( T_1 \). All bits in \( T_1 \) that aren't don't cares, have the same value in the resulting terms. In the above case \( T_1 \) has two don't cares. The term 11- is obtained by replacing the first don't care by a 1. The first bit of \( T_2 \) is 0 after all. The rest of \( T_1 \) is copied to the result term. The second don't care is replaced by a 0, because the last bit of \( T_2 \) is a 1. Now the first two bits of \( T_1 \) are copied to the result term. The complete algorithm is the following.

\begin{verbatim}
For all bits n
    if T1[n] is don't care
        if T2[n] isn't don't care
            ResultTerm[n] = ¬T1[n]
            Copy rest of bits from T1 to ResultTerm
            Add ResultTerm to set
        else (T2[n] is don't care)
            do nothing
    else (T1[n] isn't don't care)
        do nothing
\end{verbatim}

From this algorithm we see that the number of terms in the resultset is equal to the number of don't cares in \( T_1 \) minus the number of don't cares in \( T_2 \).

The function subtract however doesn't return a set of terms but a single term. This has the advantage that we can use subtract without having a class that implements the term set. If we make a loop that calls subtract several times, we get all terms of the
resulting set nevertheless. To keep track of which don't care has to be replaced, subtract takes a counter i as argument. Counter i points out which bit has to be checked. This is n in the above algorithm.

The algorithm used in subtract is somewhat different from the one above. It has the precondition that $T_2 < T_1$. This means that $T_1[n]$ and $T_2[n]$ have the same values if $T_1[n]$ is 0 or 1. So if they haven't the same value, we know that $T_1[n]$ is a don't care. First a Term T is created. Then all bits are checked. If $T_1[n]=T_2[n]$ isn't true, so $T_1[n]$ is don't care and bit n is the next don't care to be replaced, it is replaced by the negation of $T_2[n]$. The bits before bit n have already been copied to T, because of the first and second if-statement. Counter i is now set to n+1. So the next time subtract is called, it tries to replace the don't care at position n+1. Now only the bits after bit n must be copied to T. During this it's examined if $T_1$ and $T_2$ have the same tail. If this is the case no don't cares have to be replaced and i is set to next position after the last bit of $T_1$. This is the condition to end the loop that calls subtract. If the function comes to the last two lines, this means that $T_1 = T_2$. Now i is also set larger than the last bit and a non valid term is returned. Subtracting two equal terms produces an empty set after all.
A major goal in the design of the logic library was to make very robust input functions. These functions should detect as many errors in the circuit description as possible. Because the error detection is such an important issue, a separate chapter is dedicated to it. We can distinguish several kinds of errors. First, we will look at errors that can occur in descriptions of both PLA and KISS format. Then we look at specific errors that can occur in descriptions of respectively PLA's, Mealy machines and Moore machines. The errors are discussed in the following way:

- Description of the error
- Class and Function were it is detected
- Further explanation on the error and the way of detection
- Type of error
- Message (n means linenumber, xx represents a string)

We can separate the errors also in a different way. Some errors are not really errors, they are only warnings. Some errors are so severe, that the program has to stop immediately. These are called fatal errors. The other errors don't stop the program at once. They are listed one by one. At the end of the input function it's checked if there were errors and if so the program is aborted anyhow. After all the input isn't correct, so we can't do further operations on the circuit. In this way the program doesn't detect only one error each time it is run. This type of error is just called "error". The functions that handle the three types of error are Warning, Fatal and Error. They take messages in the form of a stream, which they print on the error output stream cerr. These functions can be found in the file l_suppor.cpp.

The logic library doesn't support a function that just reads some input and then determines if this is the description of a PLA, a Mealy machine or a Moore machine. A function that does this is very difficult to design. If we want that the description is stored at the same time, we would have to create a new object that can store all three kinds of circuits. This is also difficult and not very elegant. Besides mostly the user knows what kind of circuit he is dealing with. If he has an unknown circuit he can determine if it is a PLA very simple. A PLA has only inputs and outputs on its description lines. If it isn't a PLA it should be a Moore or Mealy machine. A method to determine which of these machines has to be chosen is just to read the description as a Mealy machine. If the program detects that “void” states are used, we know that the description is a Moore machine.

7.1 Errors in PLA and KISS descriptions
• The number of input bits exceeds MAX_VARIABLES

The class Term is based on an array. This array has a fixed size which is defined in the file l_defs.h as MAX_VARIABLES. If ninputs > MAX_VARIABLES this error occurs.

Error Number of input bits is larger than MAX_VARIABLES (see l_defs.h)

• A parameter was specified twice

If the value of the parameter differs from the initial value, it has been specified already.

Warning (line n) Duplicated .xx directive

• An unknown type was specified

If the output type isn't one of "f", "fd", etc, it is unknown. It keeps the initial value "Pla_unknown"

Error (line n) Unknown type

• .phase is specified before .o

If the number of output functions isn't known phase can't be specified. If the number of outputs is 0, it is concluded that .o isn't defined yet. The phase specification isn't read now.

Error (line n) Number of outputs has to be specified before .phase

• The definition of phase was too long

If there are more 0's or 1's specified for phase than there are outputs, the definition of phase is too long. The error is detected by looking if there is still a 0 or 1 on the inputstream after noutputs bits have been read. These bits are ignored.

Warning (line n) Too many bits after .phase

• The definition of phase was too short

If there are less 0's or 1's specified for phase than there are outputs, the definition of phase is too short. The error is detected by looking if counter i, which count the number of bits that was read from the input, is smaller than noutputs.

Warning (line n) Too many bits after .phase

• Something else than 0 or 1 was used for the definition of phase

Only 0 and 1 are allowed for specifying .phase

Error (line n) Only 0 or 1 allowed for .phase
**ERRORDETECTION IN INPUT FUNCTIONS**

- **.ilb is specified before .i**
  
  If the number of input bits isn't known, the input-name array can't be initialized. If the number of inputs is 0, it is concluded that .i isn't defined yet. The .ilb specification isn't read now.

  \[ Error \ (line \ n) \ Number \ of \ inputs \ has \ to \ be \ specified \ before \ .ilb \]

- **Too few input-names were specified after .ilb**
  
  The array for the input-names is initialized for ninputs names. If the arrays isn't full, there are too few input-names specified. The standard Array function isFull is used to check this. The open places in the array are filled with the string "No_name".

  \[ Error \ (line \ n) \ Too \ few \ input \ names \]

- **Too many input-names were specified after .ilb**
  
  The array for the input-names is initialized for ninputs names. If the arrays is full while there are still names read from the input, there are too many input-names specified. The standard Array function isFull is used to check this. The extra names are ignored.

  \[ Error \ (line \ n) \ Too \ many \ input \ names \]

- **Input-names were specified twice after .ilb**
  
  The standard Array function hasMember is used to check if a name is already in the array of input-names. The duplicate name is printed in the error message. Besides it is added nevertheless to the array.

  \[ Error \ (line \ n) \ Duplicate \ input \ name \ xx \]

- **.ob is specified before .o**
  
  If the number of output functions isn't known, the output-name array can't be initialized. If the number of outputs is 0, it is concluded that .o isn't defined yet. The .ob specification isn't read now.

  \[ Error \ (line \ n) \ Number \ of \ inputs \ has \ to \ be \ specified \ before \ .phase \]

- **Too few output-names were specified after .ob**
  
  The array for the output-names is initialized for noutputs names. If the arrays isn't full, there are too few output-names specified. The standard Array function isFull is used to check this. The open places in the array are filled with the string "No_name".

  \[ Error \ (line \ n) \ Too \ few \ output \ names \]
- Too many output-names were specified after .ob
  
  The array for the output-names is initialized for n outputs names. If the array is full while there are still names read from the output, there are too many output-names specified. The standard Array function isFull is used to check this. The extra names are ignored.

  **Error**  
  (line n) Too many output names

- Output-names were specified twice after .ob

  The standard Array function hasMember is used to check if a name is already in the array of output-names. The duplicate name is printed in the errorMessage. Besides it is added nevertheless to the array.

  **Error**  
  (line n) Duplicate output name xx

- An unknown command was specified

  The string that followed the '.' at the beginning of a line, wasn't recognized as a command. The string is printed in the errorMessage.

  **Warning**  
  (line n) Unrecognized command: xx

- There was more on a line than one command

  Each line should contain only one command and the appropriate parameters. If there is still something on the stream, when everything was read that was expected, there is more on the line than one command.

  **Warning**  
  (line n) Only one command per line allowed

- Only command was read, no parameters

  After every command a number of parameters should follow. If the input stream is empty after reading the command string, no parameters were specified. Now the command is ignored.

  **Error**  
  (line n) Rest of command expected

- Number of inputbits, outputbits, states or transitions was negative.

  None of the mentioned values should be smaller than 0.

  **Error**  
  (line n) No negative numbers allowed

- No number of inputbits specified

  According the PLA and KISS formats, the number of inputbits must be specified. So it may not be 0.

  **Error**  
  Number of inputbits must be specified
ERRORDETECTION IN INPUT FUNCTIONS

- **No number of outbits specified**

  According to the PLA and KISS formats, the number of outbits must be specified. So it may not be 0.
  
  **Error**  
  Number of outbits must be specified

- **Other characters than 0, 1 and - used in inputpatterns**

  Only 0, 1 and - may be used in inputpatterns. After this error the function stops reading. So the bits that aren't read until then stay don't care and the inputbits that are still on the stream may cause other errors. See also the next error.
  
  **Error**  
  Only 0, 1, - allowed for terms

- **Too few bits were specified for inputpattern**

  This error can only occur when the inputstream suddenly ends. If there aren't enough bits on the inputstream the input function tries to read until all bits are read. Now the previous error will probably be detected in the case of sequential machines and the error too few bits in output will occur for PLA's. The term that was found is printed in the errormessage, completed with don't cares.
  
  **Error**  
  Too few bits in term xx

- **Other characters than 0, 1, - and ~ used in outputpatterns**

  Only 0, 1, - and ~ may be used in outputpatterns. After this error the function stops reading. So the bits that aren't read until then stay don't care and the inputbits that are still on the stream may cause other errors.
  
  **Error**  
  Only 0, 1, - and ~ allowed for outputs

- **Too few bits were specified for outputpattern**

  This error can only occur when the inputstream suddenly ends. The outputterm that was found is printed in the errormessage, completed with irrelevants.
  
  **Error**  
  Too few bits in output xx

- **Too much on one line**

  There are too many bits in the input- or outputpatterns or there is just something more on the line than expected.
  
  **Warning** (line n) Nothing more expected on this line

- **A command wasn't in the command block**

  All commands should be in one block. After the description lines begin, no commands are allowed anymore, except for the .e command.
  
  **Error** (line n) No commands allowed here
error detection in input functions

- The inputstream wasn't usable
  Pla :: inputFrom
  Mealy :: inputFrom
  Moore :: inputFrom

The inputstream may be a diskfile. When this file is specified incorrectly, it can't be found and the inputstream is unusable.

Fatal error InputStream not usable

7.2 Errors in PLA descriptions

- A number of states was specified
  Pla :: checklnfo
  The function checklnfo check if the information from the file header is correct for a PLA. A PLA has no states, so if nstates differs from 0 it will probably be no PLA.

Error Number of states specified for Pla

- The number for .p was specified too small
  Pla :: inputFrom
  If .p is specified, the arrays that store the inputs and outputs are initialized to that size. So if these arrays are full and still inputs and outputs are read, we have an overflow. When the use of .p is omitted, a predefined size is taken for the arrays.
  This is done in Pla :: checklnfo. The predefined size for PLAs is PLA_TABLE_SIZE. Because this may not be enough also a delta value is defined: PLA_TABLE_DELTA. These constants can be found in the file l_defs.h.

Error (line n) Overflow; .p was specified too small

- The number for .p was specified too large
  Pla :: inputFrom
  This can be used to set .p to the right value, so that the input and output arrays have exactly the correct size.

Warning .p was set too large (should be xx)

- The description of the I/O function isn't correct.
  Pla :: checklnputs
  There may be conflicts in the description of the PLAs I/O function. Consider for example the following two descriptions. 1-01 -> 00- and -01 -> 10-. This description is ambiguous. For 1-01 both 00- and 10- are possibilities outcomes. This error occur when one input covers another input, but the corresponding output doesn't cover the other.
  This is checked in checklnputs. The inputs that cause trouble are printed.

Error inputs xx and xx are not distinct

7.3 Errors in Mealy machine descriptions

- No number of states was specified
  Mealy :: checklnfo
  For Mealy machines a number of states has to be specified.

Error No number of states specified for Mealy machine

63
ERRORDETECTION IN INPUT FUNCTIONS

- "void" was used as state-name

For Mealy machines it's not allowed to use "void" as a state-name. This word is reserved for the description of Moore machines.

\textit{Error} (line n) 'void' is not allowed in Mealy machine

- Overflow of state set

An overflow of the state set can have several causes. The command .s could be specified too small. A state-name can be spelled wrong. Or an error in the preceding inputpattern causes inputbits to be read as states. If the state that has to be added to the set isn't already in the set and the state set is full, then we have overflow.

\textit{Error} (line n) State set overflow

- Overflow of input set or output set

There are more inputpatterns or outputpatterns than can fit in the sets.

\textit{Error} Term set overflow

- The description of the $\delta$ or $\beta$ functions isn't correct.

If two different next states or outputs are defined for one input/current state pair, the machine isn't described correctly. This is checked explicitly in checkFunction. The other two functions do this check also, but implicitly. It's for the user to decide which of the functions he uses. The two conflicting inputs and the state are printed in the errormessage.

\textit{Error} error in deltafunction for inputs xx and xx at state xx
\textit{Error} error in betafunction for inputs xx and xx at state xx

There are some errors that are detected, but reported as other errors. If for example a next state fails, the program reads (a part of) the outputpattern as next state. This may cause state set overflow and too few bits in the output. For missing inputs, current states and outputs, the program behaves in the same way. Also when too many bits in input or output, or too many states are specified, the program may detect state set overflow. Besides it detects that there is to much on one line.

At last a small comment should be given on Mealy :: checkInfo. For Mealy machine the .p directive may be omitted as well as for PLA's. Therefore if ntransitions is 0, it has to be given some predefined value. After all the input set, output set, $\delta$ and $\beta$ functions are initialized with this size. This predefined value is STATE_TABLE_SIZE. Also a delta value is given so that the arrays can grow if STATE_TABLE_SIZE is too small. The delta value is STATE_TABLE_DELTA. Both constants can be found in the file ldefs.h. When .p is specified however, we have to give transitionsDelta a value. The function multipleExclusive makes the input set and the $\delta$ and $\beta$ functions grow dramatically. So they must have space to grow. The variable transitionsDelta is defined in the file ldefs.h. This is also done in Moore :: checkInfo.
7.4 Errors in Moore machine descriptions

- **No number of states was specified**
  For Moore machines a number of states has to be specified.
  
  **Error** No number of states specified for Moore machine

- **"void" was used as current state name**
  For Moore machines it's not allowed to use "void" as current state-name. It's only allowed for next-states.
  
  **Error** (line n) 'void' is not allowed for current state in Moore machine

- **Overflow of state set**
  An overflow of the state set can have several causes. The command .s could be specified too small. A state-name can be spelled wrong. Or an error in the preceding inputpattern causes inputbits to be read as states. If the state that has to be added to the set isn't already in the set and the state set is full, then we have overflow.
  
  **Error** (line n) State set overflow

- **Overflow of input set or output set**
  There are more inputpatterns or outputpatterns than can fit in the sets.
  
  **Error** Term set overflow

- **Inputpattern in λ function description wasn't don't care**
  A description of the λ function in fact consists of a state and the corresponding outputpattern. The inputpattern on such a line should be don't care. If the input is something else than don't care, this is ignored.
  
  **Warning** (line n) Input should be don't care

- **The description of the δ function isn't correct.**
  If two different next states or outputs are defined for one input/current state pair, the machine isn't described correctly. This is checked explicitly in checkFunction. The other two functions do this check also, but implicitly. It's for the user to decide which of the functions he uses. The two conflicting inputs and the state are printed in the error message.
  
  **Error** error in deltafunction for inputs xx and xx at state xx

7.5 Error propagation by streams

We have seen that errors in terms are detected by Term::inputFrom. There are two problems now. First the calling function (for example Pla::inputFrom) wants to know if
the input was correct. If it wasn’t correct this has the consequence that the complete circuit description is incorrect so that the program is aborted. Second we want to print the linenumber where the error occurred. With this we can find it back in the inputfile. In Term::inputFrom the variable linenumber isn’t available. If Term::inputFrom reports an error we can print the linenumber in Pla::inputFrom. This is done by the line “Error was encountered in line n”.

The problem is now Term::inputFrom doesn’t return anything. It’s based on the virtual function inputFrom is Streamable, so we can’t change this. The stream object offers a solution here. The object contains a variable that stores the status of the stream. The status can be “good” or combinations of “bad”, “fail” and “eof”. The status is “bad” if some error occurred while reading from a stream. For example when the disk isn’t ready to read somethin from a filestream, this occurs. The status is “fail” iff some operation on the stream fails. This value is perfect to report an error from functions as Term::inputFrom. “eof” means of course the end of the stream. The status of a stream can be examined by the functions “good()”, “bad()”, “fail()” and “eof()”. A status can be set by the function “clear()”. This construction works fine for inputpatterns, but not for outputpatterns. When the stream reaches its end, not only eof is set but also fail. If we check on fail only, we would get also errors in cases that there were no errors. So we check on the combination of fail and not eof. This implies that when there is really an error, eof may not be true. Therefore we use the function putback, which puts a character on the stream, so that eof can’t be true.
8. FILESTRUCTURE

In paragraph 2.3.1 was mentioned that C++ programs can be split into many files. This was also done for the logic library. An overview of all files used in the library will be given below with their contents. Most files start with "l_" to show that they belong to the logic library. Only the files packarray.* and strmable.h don't have this prefix, because they aren't specially designed for the logic library.

Table 6 Overview of the files in the logic library

- l_pla.h Declaration of the Pla class
- l_pla.cpp Definition of the Pla class member functions
- l_seq.h Declaration of the Mealy, Moore and StateMachine class
- l_seq.cpp Definition of the Mealy, Moore and StateMachine member functions
- l_set_fu.h Declarations of the set and function classes: GenrealInformation, TermSet, InputSet, OutputSet, StateSet, DeltaFunction, BetaFunction, LambdaFunction, InverseDelta, InverseBeta and InverseLambda
- l_set_fu.cpp Definition of the members of the classes, declared in l_set_fu.h
- l_elems.h Declaration of the elementary classes State, OutputIndex, IndexPair and the array and list classes StateArray, StateList, OutputIndexArray, IndexPairList, StringArray, TermArray, OutputArray, StateTable, OutputTable, IndexPairTable and StateListArray
- l_elems.cpp Definition of the print functions of StateArray, StateList, OutputIndexArray, IndexPairList and StringArray
- l_term.h Declaration of the classes Term and Output
- l_term.cpp Definition of members and friends of Term and Output
- packarray.h Declaration of the PackedArray class
- packary.cpp Definition of the function log2, that is used by PackedArray
- l_support.h Declarations of the supporting functions Warning, Error and Fatal and the function toLine
- l_support.cpp Definition the functions Warning, Error, Fatal and toLine
- l_defs.h Definition of some important types and constants
- strmable.h Declaration of the Streamable class

The file l_defs.h contains definitions of some important types and constants. The meaning of the constants MAX_VARIABLES, PLA_TABLE_SIZE, PLA_TABLE_DELTA, STATE_TABLE_SIZE and STATE_TABLE_DELTA was explained in chapter 7. The type OutputIndex, InputIndex and StateIndex are defined here so that the user can chose if he wants to store these indexes in one or in two
ERRORDETECTION IN INPUT FUNCTIONS

bytes. For one byte the type char is used, for two bytes the type int may be used. With indexes of one byte we need less memory space to store $\delta$ and $\beta$ functions. On the other hand we can only use 256 different values. The enumerated type of classnames is required by the standard Array function isA. We have to define this function for the classes State, OutputIndex and IndexPair.

The .h files are included with the "#include" preprocessor command. The .cpp files are linked together with the application file to obtain a working program. How this linking works depends on the compiler/linker that is used. Notice that the for DOS computers the files should be compiled for a so called large memory model [3]. Otherwise we can only use several kbytes of memory.

"L_muex.cpp" is such an application file. It was used to test the input function and the multipleExclusive function of the Mealy class. When this file is linked with the files of the logic library, a "exe" file is obtained that can be executed from DOS. This program reads a Mealy machines description from a file that was specified on the command line and makes the machine multiple exclusive. Then the description of the machine is saved to a new file that has the extension ".exl".
9. CONCLUSIONS

The goal of this project was to develop and implement some basic modules for a software library for logic optimization in C++. This included developing data structures and algorithms for reading a circuit specification from an external text file into an internal data structure. The classes Pla, Mealy and Moore, that were developed and implemented in the scope of the project, are good modules for this library. The design satisfies all specifications that were mentioned in the introduction. It is possible to read input files in PLA and KISS format and to check on many errors as we have seen in chapter 7.

Moreover it appears that the data structure is transparent. The PLA class follows the structure of the PLA format and the classes for the sequential machines are based on their commonly used mathematical descriptions. Also the underlying classes have very straightforward designs.
10. RECOMMENDATIONS

As a matter of fact it is true that the design of a software library is never finished. There can always be found something to improve. At this moment I would suggest to change the following things.

In chapter 6 we have seen that there were values defined for the indexes of undefined states, don't care states and undefined outputs. To make the program more readable we could define constants for this and for the overflow code -1:

```cpp
undefined = 0
dontcareState = 1
overflow = -1
```

For the data structure for the $\delta$ and $\beta$ function a matrix was chosen. We can imagine that after reading a machine from an input file this matrix isn't very full. Therefore we could think of implementing the $\delta$ and $\beta$ functions as sparse matrices. The advantage of C++ is that we wouldn't have to change the interface of DeltaFunction and BetaFunction. These objects can be used the same, independent of the underlying data structure.

The library consists of several files as we have seen in chapter 8. Now we have to link all these files with the application file. We can make a real library of it, a .lib file, with the help of the librarian tool of Borland C++. Then we would have to link only the library file with the application.
11. BIBLIOGRAPHY

[1] Bjarne Stroustrup  
The C++ programming language, 2nd edition  
AT&T Bell Laboratories  
Addison-Wesley Publishing Company, 1993

C++ Primer, 2nd edition  
AT&T Bell Laboratories  
Addison-Wesley Publishing Company, 1991

Version 3.1  
Borland International

[4] Sequentiële machines  
H.A. Scheepmaker  
dictaatnr. 5563
Appendix A  PROOF OF ME ALGORITHM

Proof of inv1

\[ me(c, M) \land me(c, E) \land exp(\{c\} \cup M \cup A \cup E) = exp(l_{pre}) \Rightarrow \]
\{ B = A \setminus a \} 

\[ me(c, M) \land me(c, E) \land exp(\{c\} \cup M \cup B \cup \{a\} \cup E) = exp(l_{pre}) \Rightarrow \]
\{ x = c \ast a \} 

\[ me(x, M) \land me(x, E) \land exp(\{c\} \cup M \cup B \cup \{a\} \cup E) = exp(l_{pre}) \Rightarrow \]
\{ exp(M) = \exp(c)/\exp(x) \land exp(Z) = \exp(a)/\exp(x) \Rightarrow \}
\[ me(x, M) \land me(x, E) \land exp(\{c\} \cup M \cup B \cup \{a\} \cup E) = exp(l_{pre}) \Rightarrow \]
\{ E = E \cup Y \cup Z \} 

\[ me(x, M) \land me(x, E) \land exp(\{x\} \cup M \cup B \cup E) = exp(l_{pre}) \Rightarrow \]
\{ c = x, A = B \} 

\[ me(c, M) \land me(c, E) \land exp(\{c\} \cup M \cup A \cup E) = exp(l_{pre}) = inv1 \]

Proof of inv2

\[ \forall m \in M me(m, N) \land me(M) \land exp(M \cup N) = exp(l_{pre}) \Rightarrow \]
\{ A = N \setminus c \} 

\[ \forall m \in M me(m, A \cup \{c\}) \land me(M) \land exp(M \cup A \cup \{c\}) = exp(l_{pre}) \Rightarrow \]
\{ inv1 \land me(c, A) \} 

\[ \forall m \in M me(m, A \cup E) \land me(M \cup \{c\}) \land exp(M \cup \{c\} \cup A \cup E) = exp(l_{pre}) \Rightarrow \]
\{ N = A \cup E \} 

\[ \forall m \in M me(m, N) \land me(M \cup \{c\}) \land exp(M \cup \{c\} \cup N) = exp(l_{pre}) \Rightarrow \]
\{ M = M \cup \{c\} \} 

\[ \forall m \in M me(m, N) \land me(M) \land exp(M \cup N) = exp(l_{pre}) = inv2 \]
class Pla stores a programmable logic array: a logic 10 function

class Pla : public Streamable {
private:
    GeneralInformation info;       // some information
    TermArray *itable, *otable;   // the input and output table
public:
    Pla(void);                    // constructor
    ~Pla(void);                   // destructor
    void inputFrom ( istream& in ); // read Pla from input in PLA format
    void printOn ( ostream& out ) const; // print Pla in PLA format
    void establishIrrelevant(void); // change irrelevants conform type
    void removeIrrelevant(void);   // remove irrelevants
    bool valid(void) const;        // return if pla is valid
    void delVar(int var);          // delete variable at location var

private:
    void init(void);              // allocate space
    void flush(void);             // destroy data
    bool checkInfo(void);         // check info after input
    bool checkInputs(void) const; // check inputterms
    void subst(bit oldb, bit newb); // substitute old bit by new bit
};

#endif
# include <l_pla.h>
# include <l_set_fu.h>
# include <l:suppor.h>

public member functions:

Pla • Pla(void) : Streamable(), itable(NULL), otable(NULL) {} 
Pla • -Pla(void) ( flush(); } 

void Pla :: inputFrom ( istream& in ) { 

cerr « "Reading input:\n";
if (! in.goodO) ( 
 strmstream msg;
msg « "Inputstream not usable";
Fatal(msg);
)
flush(); 
in >> info;
bool ok = in.good();
info.Ctype = Platype;
checkInfoO;
init(); 
while (in.good()) { 
in >> ws;
if (in.eofO) ( 
 break;
) 
else if (in.peakO == '#') ( // skip comments
in. ignore(MAXINT, '\n');
info.linenumber++;
)
else if (in.peakO == ' '.) ( 
in. ignore(); // '.('
if (in.getO == 'e') break; // end is reached
else ( 
strstream msg;
msg « "( line " << info.linenumber
<< " ) No commands allowed here";
Error(msg);
ok = FALSE; // command after transitions
)
else ( 
strstream linestream;
toline(linestream,in);
info.linenumber++;
linestream >> ws;
if (itable->getItemsInContainer() == info.ntransitions +
info.transitiondelta) ( 
strstream msg;
msg « "( line " << info.linenumber
<< " ) Overflow; .p was specified to small";
Error(msg);
ok = FALSE;
}
else ( 
Term * I = new Term(info.ninputs);
Output * O = new Output(info.noutputs);
linestream >> *I;
if (linestream.failO) ( //error reading *I
cerr << "\tError was encountered in line " 
<< info.linenumber << endl;
ok = FALSE;
linestream.clear(ios::eofbit| ios::badbit); //reset fail
)
linestream >> *O;
if (linestream.failO & !linestream.eofO) ( // error reading *O
cerr << "\tError was encountered in line "

```
77 25  << info.linenumber << endl;
78 20  ok = FALSE;
79 20  linestream.ignore(MAXINT);
80 16  }
81 16  itable->add("1");
82 16  otable->add("0");
83 16  if (linestream.good())
84 20  linestream >> ws;  // eat whitespace
85 16  if (linestream.good())  // still something on stream
86 20  strstream msg;
87 20  msg << "\n\n" << info.linenumber
88 16  << "(\n\n\n"  Nothing more expected on this line"
89 20  Warning(msg);
90 20  linestream.ignore(MAXINT);
91 16  )
92 12  )
93 8  )
94 4  ) // while
95 4  if (ok & valid()) ok &= checkInputs();
96 4  if (!ok) {
97 8  strstream msg;
98 8  msg << "Reading input failed"
99 8  Fatal(msg);
100 4  int tableLength = itable->getItemsInContainer();
101 4  if (info.ntransitions > tableLength) {
102 8  strstream msg;
103 8  msg << ".p was set too large (should be 
104 12  << tablelength << ");
105 8  Warning(msg);
106 8  info.ntransitions = tableLength;
107 8  }
108 4  else cerr << "Reading input completed successfully\n\n"
109 0  ) // inputFrom
110 0  void Pla :: printOn ( ostream& out ) const {
111 4  if (valid()) {
112 8  info.printOn(out);
113 8  for (int i = 0; i < itable->getItemsInContainer(); i++) {
114 12  out << (*itable)[i] << " \n\n"
115 16  << (*otable)[i] << endl;
116 8  }
117 8  out << ".e" << endl;
118 4  } else out << "Pla not valid\n";
119 0  ) // printOn
120 0  void Pla :: establishIrrelevant(void) {
121 4  cerr << "Substitute irrelevants ...
\n"
122 4  if (valid())
123 8  switch (info.Otype)
124 8  {
125 8  case f : subst(dontcare, irrelevant);
126 8  case fd: subst(zero, irrelevant);  break;
127 8  case r : subst(dontcare, irrelevant);
128 8  case dr: subst(one, irrelevant);  break;
129 8  case fr: subst(dontcare, irrelevant);
130 4  } else cerr << "Ok\n\n"
131 4  )
132 4  } else cerr << "Pla not valid\n";
133 0  ) // establishIrrelevant
134 0  void Pla :: removeIrrelevant(void) {
135 4  cerr << "Removing irrelevants ...
\n"
136 4  if (valid())
137 8  switch (info.Otype)
138 8  {
139 8  case f : subst(irrelevant, zero);
140 8  case fd: subst(irrelevant, dontcare);
141 8  break;
142 8  case r : subst(irrelevant, one);
143 8  break;
144 8  case fr: subst(irrelevant, dontcare);
145 8  )
146 4  cerr << "Ok\n\n"
147 4  )
```

bool Pla::valid(void) const { return (info.valid() && itable && otable); }

void Pla::delVar(int var) {
    cerr << "Deleting variable " << var << " ...\n";
    if (valid()) {
        if (var <= 0) {
            // do nothing
        } else if (var < info.ninputs + info.noutputs) {
            itable->forEach(delete var, &var);
            info.ninputs--;
        } else if (var < info.ninputs) {
            otable->forEach(delete var, &var);
            info.noutputs--;
        }
    } else {
        cerr << "Ok\n\n";
    }
    else cerr << "Pla not valid\n\n";
} // del_var

void Pla::init(void) {
    itable = new TermArray(info.ntransitions,0,info.transitiondelta);
    otable = new TermArray(info.ntransitions,0,info.transitiondelta);
} // init

void Pla::flush(void) {
    itable->flush();
    if (itable) delete itable;
    itable = NULL;
    if (otable) delete otable;
    otable = NULL;
} // flush

bool Pla::checkInfo(void) {
    bool ok = TRUE;
    if (info.nstates) {
        stringstream msg;
        msg << "Number of states specified for Pla";
        Error(msg);
        ok = FALSE;
    }
    if (info.ntransitions) {
        stringstream msg;
        info.ntransitions = PLA_TABLE_SIZE;
        cerr << "Number of lines is set to " << info.ntransitions << endl;
        info.transitiondelta = PLA_TABLE_DELTA;
        cerr << "This number can increase with "
            << info.transitiondelta << endl;
    }
    return ok;
} // checkInfo

bool Pla::checkInputs(void) const {
    bool ok1 = TRUE;
    bool ok2 = TRUE;
    for (int i = 0; i < info.ntransitions; i++) {
        for (int j = i+1; j < info.ntransitions; j++) {
            ok2 = (((*itable)[i] < (*itable)[j]) &&
                    ((*otable)[i] < (*otable)[j])) ||
...
229 18    (**itable)[j] < (**itable)[i] &&
230 18    (**otable)[j] < (**otable)[i]);
231 12    if (!ok2) {
232 16        stringstream msg;
233 16        msg << " inputs " << (**itable)[i] << " and "
234 20        << (**itable)[j] << " are not distinct";
235 16        Error(msg);
236 12    }
237 12    ok1 &= ok2;
238 8    }
239 4    return ok1;
240 0    } // checkInfo
241 0
242 0    void Pla :: subst(bit oldb, bit newb) {
243 4    for(int n = 0; (**otable)[n] != NOOBJECT; n++) {
244 8        for (int i = 0; i < info.noutputs; i++)
245 12            if (**otable)[n].get_var(i) == oldb
246 16                (**otable)[n].set_var(i, newb);
247 4    }
248 0    } // subst
Cseq.h
File l_seq.h Created at 1:00pm on Sunday, October 9, 1994
goossens Page 1 of 2

1 0 #ifndef __L_SEQ_H
2 0 # define __L_SEQ_H
3 0
4 0 # include <iostream.h>
5 0 # include <strmable.h>
6 0 # include <l elems.h>
7 0 # include <l_set_fu.h>
8 0
9 0 class StateMachine {
10 0  protected:
11 0  InputSet input_set;
12 0  OutputSet output_set;
13 0  StateSet state_set;
14 0  DeltaFunction delta_function;
15 0  InverseDelta inverse_delta;
16 0  GeneralInformation info;
17 0
18 0  public:
19 0  StateMachine(void); // constructor
20 0  StateMachine( StateMachine& ); // copy constructor
21 0  ~StateMachine(void); // destructor
22 0
23 0  virtual void inputfrom( istream& ) = 0; // see derived classes
24 0  virtual void printOn( ostream& ) const = 0; // see derived classes
25 0
26 0  bool validStateMachine(void) const; // return if Mealy machine is valid
27 0  bool checkDeltaFunction(void) const; // check on state collision
28 0  void establishIrrelevant(void); // substitute '-' in outputs
29 0  bool fullySpecified(bool); // checks complete covering
30 0  virtual void multipleExclusive(void) = 0; // see derived classes
31 0  void establishInverseDelta(void); // creates inverse delta relation
32 0  void printlnverseDeltaOn(ostream& ) const; // print inverse delta
33 0
34 0  protected:
35 0  void initStateMachine(void); // allocate space
36 0  void flushStateMachine(void); // destroy data
37 0  void completeDeltafunction(void); // completes delta function
38 0
39 0  class Mealy : public StateMachine, public Streamable {
40 0  private:
41 0   Betafunction beta_function;
42 0   InverseBeta inverse_beta;
43 0
44 0  public:
45 0   Mealy(void); // constructor
46 0   Mealy( Mealy& ); // copy constructor
47 0   ~Mealy(void); // destructor
48 0
49 0   virtual void inputfrom( istream& ) ; // read Mealy mach in KISS format
50 0   virtual void printOn( ostream& ) const; // print Mealy mach in KISS format
51 0
52 0   bool valid(void) const; // return if Mealy machine is valid
53 0   void checkFunctions(void); // check if functions are ok
54 0   void completeFunctions(void); // completes outputfunction
55 0   void multipleExclusive(void); // make inputs multiple exclusive
56 0   void establishInverseBeta(void); // creates inverse beta
57 0   void printlnverseBetaOn(ostream& ) const; // print Inverse beta
58 0
59 0  private:
60 0   void init(void); // allocate space
61 0   void flush(void); // destroy data
62 0   bool checkInfo(void); // check info after input
63 0   int mergeFunctions(int i, int j); // merge delta and labda functions
64 0   int countTransitions(void) const; // count the number of transisions when printed

70 0  };
71 0
72 0  class Mealy : public StateMachine, public Streamable {
73 0
74 0
75 0
76 0
class Moore : publicStateMachine, publicStreamable {
private:
    LambdaFunction lambda_function;
    InverseLambda inverse_labda;

public:
    Moore(void); // constructor
    Moore(Moore&); // copy constructor
    ~Moore(void); // destructor
    void inputFrom(istream&); // read Moore machine in KISS format
    void printOn(ostream& ) const; // print Moore machine in KISS format
    bool valid(void) const; // return if Moore machine is valid
    void checkfunctions(void); // check transition function
    void completefunctions(void); // complete transition function
    void multipleExclusive(void); // make inputs multiple exclusive
    void establishInverseLambda(void); // create inverse labda
    void printlnverseLambdaOn(ostream& out) const; // print inverse labda

private:
    void init(void); // allocate space
    void flush(void); // destroy data
    bool checkinfo(void); // check info after input
    int countTransitions(void) const; // count the number of transitions when printed

}; // class Moore

#endif
```cpp
#include <l_seq.h>
#include <l_set_fu.h>
#include <l_support.h>

// Project: logic
// File: implementation of sequential machines: Mealy and Moore machines

//*****************************************************************************
// file l_seq.cpp
// Created at 10:04pm on Sunday, January 8, 1995

public member functions:

StateMachine :: StateMachine() {}

StateMachine :: StateMachine( StateMachine& S) {
    input_set = S.input_set;
    output_set = S.output_set;
    state_set = S.state_set;
    delta_function = S.delta_function;
    inverse_delta = S.inverse_delta;
    info = S.info;
}

StateMachine :: ~StateMachine() {} // copy constructor

bool StateMachine :: validStateMachine(void) const {
    return info.valid() && input_set.valid() &&
    output_set.valid() && state_set.valid() &&
    delta_function.valid();
}

StateMachine :: StateMachine( StateMachine& S) {
    input_set = S.input_set;
    output_set = S.output_set;
    state_set = S.state_set;
    delta_function = S.delta_function;
    inverse_delta = S.inverse_delta;
    info = S.info;
}

bool StateMachine :: checkDeltaFunction(void) const {
    if (validStateMachine()) {
        int errorstate;
        bool error = FALSE;
        for (InputIndex i = 0; i < input_set.nTerms(); i++)
            for (InputIndex j = i+1; j < input_set.nTerms(); j++)
                if (input_set[j] * input_set[i] && i) {
                    errorstate = delta_function.distinct(j, i);
                    if (errorstate) {
                        stringstream msg;
                        msg << "collision in delta function for inputs ".
                          << input_set[i] << " and " << input_set[j] << 
                          " at state " << state_set.StateName(errorstate-1);
                        Error(msg);
                    }
                }
        return error | errorstate;
    }
    else cerr << "Machine not valid\n"
    return TRUE;
}

StateMachine :: completeDeltaFunction(void) {
    if (validStateMachine()) {
        int errorstate;
        for (InputIndex i = 0; i < input_set.nTerms(); i++)
            for (InputIndex j = 0; j < input_set.nTerms(); j++)
                if (input_set[j] < input_set[i] && i != j) {
                    errorstate = delta_function.merge(j, i);
                    if (errorstate) {
                        stringstream msg;
                        msg << "collision in delta function for inputs ".
                          << input_set[i] << " and " << input_set[j] << 
                          " at state " << state_set.StateName(errorstate-1);
                        Error(msg);
                    }
                }
        else cerr << "Machine not valid\n"
    }
}
```

---

Project: logic

File: implementation of sequential machines: Mealy and Moore machines

---

StateMachine :: StateMachine() {} // copy constructor

StateMachine :: ~StateMachine() {} // copy constructor

bool StateMachine :: validStateMachine(void) const {
    return info.valid() && input_set.valid() &&
    output_set.valid() && state_set.valid() &&
    delta_function.valid();
}

bool StateMachine :: checkDeltaFunction(void) const {
    if (validStateMachine()) {
        int errorstate;
        bool error = FALSE;
        for (InputIndex i = 0; i < input_set.nTerms(); i++)
            for (InputIndex j = i+1; j < input_set.nTerms(); j++)
                if (input_set[j] * input_set[i] && i) {
                    errorstate = delta_function.distinct(j, i);
                    if (errorstate) {
                        stringstream msg;
                        msg << "collision in delta function for inputs ".
                          << input_set[i] << " and " << input_set[j] << 
                          " at state " << state_set.StateName(errorstate-1);
                        Error(msg);
                    }
                }
        return error | errorstate;
    }
    else cerr << "Machine not valid\n"
    return TRUE;
}

StateMachine :: completeDeltaFunction(void) {
    if (validStateMachine()) {
        int errorstate;
        for (InputIndex i = 0; i < input_set.nTerms(); i++)
            for (InputIndex j = 0; j < input_set.nTerms(); j++)
                if (input_set[j] < input_set[i] && i != j) {
                    errorstate = delta_function.merge(j, i);
                    if (errorstate) {
                        stringstream msg;
                        msg << "collision in delta function for inputs ".
                          << input_set[i] << " and " << input_set[j] << 
                          " at state " << state_set.StateName(errorstate-1);
                        Error(msg);
                    }
                }
        else cerr << "Machine not valid\n"
    }
}
bool StateMachine::fullySpecified(bool DeleteOnError) {
  cerr << "Checking covering of boolean space:\n";
  if (validStateMachine()) {
    TermSet Ta(info.ninputs,8*info.ninputs,8*info.ninputs);
    Term * t = new Term(info.ninputs);
    Ta.addTerm(*t);
    for (int i = 0; i < input_set.nTerms(); i++) {
      Term * Ti;
      Ti = &(input_set[i]);
      int limit = Ta.nTerms();
      for (int Ie = 0; Ie < limit; Ie++) {
        Term TIe(Ta[Ie]);
        Term T2(info.ninputs);
        Term T1(*Ti * TIe);
        if (TIe < *T1) {
          Ta.deleteTerm(Ie);
          limit--;
          int j = 0;
          Term * T2;
          while (j < info.ninputs+1) {
            T2 = new Term(subtract(TIe, T1, j));
            Ta.merge_addTerm(*T2);
          }
        } else {
          int ok = (Ta.nTerms() == 0);
          if (ok) cerr << "\vResult: Complete boolean space is covered by inputs\n\n";
          else {
            cerr << "\vResult: The following term(s) should be added:\n";
            for (int k = 0; k < Ta.nTerms(); k++) {
              cerr << '\t' << Ta[k] << endl;
            }
          }
        }
      }
      int ole = (Ta.nTerms() == 0);
      if (ole) cerr << "\vResult: Complete boolean space is covered by inputs\n\n";
      else {
        cerr << "\vResult: Machine not valid\n\n";
        return FALSE;
      }
    } elsecerr << "\vResult: Machine not valid\n\n";
    return FALSE;
  }
  cerr << "Creating inverse delta table:\n";
  if (validStateMachine()) {
    inverse_delta.init(info.nstates);
    for (int i = 0; i < input_set.nTerms(); i++)
      for (int s = 0; s < info.nstates; s++) {
        inverse_delta.set(delta_function.get(i,s),i,s);
      }
  } else cerr << "\vResult: Machine not valid\n\n";
  return FALSE;
}

void StateMachine::establishInverseDelta(void) {
  cerr << "Creating inverse delta table:\n";
  if (validStateMachine()) {
    inverse_delta.init(info.nstates);
    for (int i = 0; i < input_set.nTerms(); i++)
      for (int s = 0; s < info.nstates; s++) {
        inverse_delta.set(delta_function.get(i,s),i,s);
      }
  } else cerr << "\vResult: Machine not valid\n\n";
  return FALSE;
}

void StateMachine::printInverseDeltaOn(ostream& out) const {
  cerr << "Printing inverse delta table:\n";
  if (inverse_delta.valid()) {
    InputIndex i;
    StateIndex cs;
    bool EndOfList;
    for (int s = 0; s < info.nstates; s++) {
      ListIterator &it = inverse_delta.initIterator(s);
      out << state_set.StateName(s) << " is next-state for following input-state pairs: " <<
endl;
    EndOfList = inverse_delta.get(it,i,cs);
    while (!EndOfList) {
        out << "< " << input_set[i] << "," << state_set.StateName(cs) << " > " ;
        EndOfList = inverse_delta.get(it,i,cs);
    }
    out << endl;
    delete &it;
}
else cerr << "Create inverse delta first\n\n";
};
}  // establishInverseDelta

protected member functions:

void StateMachine :: initStateMachine(void) {
    input_set.init(info.ninputs, info.ntransitions, info.transitiondelta);
    output_set.init( info.noutputs, info.ntransitions, 
                    info.transitiondelta, info.Otype);
    state_set.init(info.nstates);
    delta_function.init(info.ntransitions, info.nstates, info.transitiondelta);
};
}  // initStateMachine

void StateMachine :: flushStateMachine(void) {
    info.flush();
    input_set.flush();
    output_set.flush();
    state_set.flush();
    delta_function.flush();
    inverse_delta.flush();
};
}  // flushStateMachine

public member functions:

Mealy :: Mealy() : StateMachine(), Streamable() {
    info.Ctype = Mealytype;
};
}  // constructor

Mealy :: Mealy(Mealy& M) : StateMachine( (StateMachine&) M), Streamable() {}

Mealy :: ~Mealy() (flush();)

void Mealy :: inputFrom( istream& in) {
    char CS[80], NS[80];
    String * str;
    int i, cs, ns, 0;
    strstream msg;  // stream for error messages
    cerr << "Reading input:\n";
    if (!in.good()) {
        msg << "Inputstream not usable";
        Fatal(msg);
    }
    flush();
    in >> info;
    bool ok = in.good();
    info.Ctype = Mealytype;
    checkInfo();
    init();
    while (in.good()) {
        if (in.get() == 'e') break;
    }
    while (in.eof()) {
        break;
    }
    else if (in.peek() == '#' || in.peek() == '!' )  // skip comments
        in.ignore(MAXINT, '\n');
    else if (in.peek() == '.') {
        in.ignore();  // "."
        if (in.get() == 'e') break;  // end is reached
    }
    else {
        msg << "(line " << info.linenumber
              << ") No commands allowed here";
        }
else ns = state_set.addNextState(*str);

for (int s = 0; s < info.nstates; s++) {
    delta_function.set(i, s, ns);
    beta_function.set(i, s, 0);
}
else cs = state_set.addState(*str);
str = new String(NS);
if (state_set.isFull() && !state_set.hasMember(*str)) {
    msg << "(line " << info.linenumber << "); State set overflow"
    Error(msg);
    ok = FALSE;
    goto skip_line;
}

if (state_set.isDontcare(String(NS)) {  // error reading "I",
    str = new String(NS);
    if (state_set.isFull() && !state_set.hasMember(*str) &&
        !state_set.isDontcare(String(NS)) {  // error reading "O",
        msg << "(line " << info.linenumber << "); State set overflow"
        Error(msg);
        ok = FALSE;
        goto skip_line;
    }
}

o = output_set.addOutputTerm(*O);
if (o == -1) {
    cerr << "Error was encountered for output in line " << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}

if (state_set.isDontcare(String(CS))) {
    str = new String(NS);
    if (state_set.isFull() && !state_set.hasMember(*str) &&
        !state_set.isDontcare(String(NS))) {
        msg << "(line " << info.linenumber << "); State set overflow"
        Error(msg);
        ok = FALSE;
        goto skip_line;
    }
    else ns = state_set.addNextState(*str);
    for (int s = 0; s < info.nstates; s++) {
        delta_function.set(i, s, ns);
        beta_function.set(i, s, o);
    }
}
else {  // error reading "I"
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}
linestream >> *I;
if (linestream.fail()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}
linestream >> CS >> NS;
if (state_set.isVoid(CS) || state_set.isVoid(NS)) {
    msg << "(line " << info.linenumber << "); \'void\' is not allowed in Mealy machine"
    Error(msg);
    ok = FALSE;
    goto skip_line;
}
linestream >> *O;
if (linestream.fail() && !linestream.eof()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}

linestream >> *1;
if (linestream.fail()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}
linestream >> *0;
if (linestream.fail() && !linestream.eof()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}

linestream >> ws;
linestream >> ws;
toLine(linestream, in);
info.linenumber++;
linestream >> ws;
Term * I = new Term(info.ninputs);
Output * O = new Output(info.noutputs);

linestream >> *1;
if (linestream.fail()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}
linestream >> *0;
if (linestream.fail() && !linestream.eof()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}

linestream >> *O;
if (linestream.fail() && !linestream.eof()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}

linestream >> *I;
if (linestream.fail()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}
linestream >> *0;
if (linestream.fail() && !linestream.eof()) {
    cerr << "\tError was encountered in line "
    << info.linenumber << endl;
    ok = FALSE;
    goto skip_line;
}
```cpp
304  24     Error(msg);
305  24     ok = FALSE;
306  24     goto skip_line;    // rest of input line is skipped
307  20     }
308  20     else {
309  24     ns = state_set.addNextState(*str);
310  24     delta_function.set(i, cs, ns);
311  24     beta_function.set(i, cs, o);
312  20     }
313  16     }  
314  12     if (linestream.good())
315  16     linestream >> ws;    // eat whitespace
316  12     if (linestream.good()) ( // still something on stream     
317  16     msg << "(line " << info.linenum    
318  20     << ") Nothing more expected on this line"
319  16     Warning(msg);
320  16     skip_line;
321  16     linestream.ignore(MAXINT);
322  12     }
323  8     }
324  4     }    // while
325  0     }
326  4     if (!tok) {
327  8     stringstream msg;
328  8     msg << "Reading input failed";
329  8     Fatal(msg);
330  4     }
331  4     else cerr << "Reading input completed successfully\n\n";
332  0     }    // inputFrom
333  0     }
334  0     void Mealy::printOn(ostream& out) const {
335  4     cerr << "Printing Mealy machine:\n";
336  4     if (valid()) {
337  8     GeneralInformation infoCopy(info);
338  8     infoCopy.ntransitions = countTransitions();
339  8     out << infoCopy;
340  0     }
341  8     for (int s = 0; s < info.nstates; s++)
342  12     for (int i = 0; i < input_set.nTerms(); i++) {
343  16     StateIndex ns = delta_function.get(i, s);
344  16     if (ns) ( // not undefined    
345  20     out << input_set[i] << " "    
346  24     << state_set.StateName(s) << " "    
347  24     << state_set.NextStateName(ns) << " "    
348  20     OutputIndex o = beta_function.get(i, s);
349  20     if (o) out << output_set[o] << endl;    // output is specified    
350  20     else ( Term T(info.noutputs); out << T << endl; )    // output is don't care    
351  16     }
352  12     out << ".e" << endl;
353  4     }
354  4     else cerr << "Mealy machine not valid\n";
355  0     }    // printOn
356  0     }
357  0     bool Mealy::valid(void) const {
358  4     return validStateMachine() && beta_function.valid();
359  0     }    // valid
360  0     }
361  0     void Mealy::checkFunctions(void) {
362  0     cerr << "Checking delta and beta function: \n";
363  4     if (valid()) {
364  8     int errorstate = 0;
365  8     bool error = FALSE;
366  8     for (int i = 0; i < input_set.nTerms(); i++) {
367  12     for (int j = i+1; j < input_set.nTerms(); j++)
368  16     if (input_set[i] * input_set[j] <= 0)    
369  16     errorstate = delta_function.distinct(i, j);
370  20     if (errorstate) {
371  24     stringstream msg;
372  24     msg << "error in delta function for inputs " << 
373  24     input_set[i] << " and " << input_set[j] << " at state " << 
374  24     state_set.StateName(errorstate-1);
375  24     Error(msg);
376  24     }
377  20     error |= errorstate;
378  20     errorstate = beta_function.distinct(i, j);
379  20     
```

if (errorstate) {
    stream msg;
    msg << "error in beta function for inputs " << 
        input_set[] << " and " << input_set[] << " at state " << 
        state.set.StateName(errorstate-1);
    Error(msg);
}
    }
    error |= errorstate;
}
}
}
}
} else {
}
}
else cerr << "Result: Mealy machine not valid\n\n";
}
else cerr << "Result: Mealy machine not valid\n\n";
}
// checkFunctions
}
void Mealy:: completeFunction(void) {
    cerr << "Completing delta and beta function:\n";
    if (valid()) {
        bool error = FALSE;
        for (int i = 0; i < input.set.nTerms(); i++) {
            for (int j = 0; j < input.set.nTerms(); j++) {
                if (input_set[j] < input_set[i] && i == j) {
                    error |= mergeFunctions(i,j);
                }
            }
        }
    } else cerr << "Result: Mealy machine not valid\n\n";
}
// completeLambdaFunction
}
void Mealy:: multipleExclusive(void) {
    cerr << "Making circuit multiple exclusive ";
    if (valid()) {
        int error = 0;
        Term *T2;
        for (int i = 0; i < input.set.nTerms(); i++) {
            int limit = input.set.nTerms();
            for (int j = i+1; j < limit; j++) {
                Term *Common = new Term(input_set wondered_1 * Term_j);
                if (*Common == 0) delete Common;
                else if (*Common != 0; not multiple exclusive
                    int n-= 0; delete Common;
                    while (n < info.ninputs) {
                        Term *T2 = new Term(subtract(input_set wondered_1,Term_j),n); 
                        int t2 = input.set.addTerm(*T2);
                        error |= mergeFunctions(t2,i);
                    }
                input.set.setTerm(i,Term_j);          // replace input[i] by common term
                error |= mergeFunctions(1,i);
            } // incomplete lambda function
    } else if (Term_j < Term_i) {
        int n = 0;
        delete Common;
        while (n < info.ninputs) {
            Term *T2 = new Term(subtract(input_set wondered_1,Term_i),n);
            int t2 = input.set.addTerm(*T2);
            error |= mergeFunctions(t2,i);
        }
    } else {
        int n = 0;
        while (n < info.ninputs) {
            add non-overlapping terms
        }
    }
}
void Mealy:: completeFunctions(void) {
    cerr << "Completing delta and beta function:\n";
    if (valid()) {
        bool error = FALSE;
        for (int i = 0; i < input.set.nTerms(); i++) {
            for (int j = 0; j < input.set.nTerms(); j++) {
                if (input_set[j] < input_set[] && i == j) {
                    error |= mergeFunctions(i,j);
                }
            }
        }
        }
    } else cerr << "Result: Machine description is incorrect";
    Fatal(msg);
   _FAR_
void Mealy:: establishInverseBeta(void) {
    if (valid) {
        inverse_beta.init(output_set.nTerms());
        for (int i = 0; i < output_set.nTerms(); i++) {
            for (int s = 0; s < info.nstates; s++)
                inverse_beta.set(beta_function.get(i,s),i,s);
        }
    } else cerr << "Mealy machine not valid\n";
}

void Mealy:: printlnverseBetaOn(ostream& out) const {
    if (inverse_beta.valid()) {
        ListIterator &it = inverse_beta.initIterator(o);
        out << output_set[0] << " is output for following input-state pairs: " << endl;
        EndOfList = inverse_beta.get(it,i,cs);
        while (!EndOfList) {
            out << "( " << input_set[i] << " , " << state_set.StateName(cs) << " ) " << endl;
            EndOfList = inverse_beta.get(it,i,cs);
        }
    } else cerr << "Create inverse beta first\n";
}

void Mealy:: flush(void) {
    ...
beta_function.flush();
inverse_beta.flush();

bool Mealy:: checklnfo(void) {
    bool ok = TRUE;
    if (!info.nstates) {
        stringstream msg;
        msg << "No number of states specified for Mealy machine";
        Fatal(msg);
        ok = FALSE;
    }
    if (!info.ntransitions) info.ntransitions = STATE_TABLE_SIZE;
    info.transitiondelta = STATE_TABLE_DELTA;
    return ok;
}

int Mealy:: mergeFunctions(int i, int j) {
    int errorstate = 0;
    error = 0;
    errorstate = delta_function.merge(i, j);
    error = errorstate;
    if (errorstate) {
        stringstream msg;
        msg << "collision in deltafunction for inputs "
        << input_set[i] << " and " << input_set[j]
        " at state " << state_set.StateName(errorstate-1);
        Error(msg);
    }
    errorstate = beta_function.merge(i, j);
    error = errorstate;
    if (errorstate) {
        stringstream msg;
        msg << "collision in betafunction for inputs "
        << input_set[i] << " and " << input_set[j]
        " at state " << state_set.StateName(errorstate-1);
        Error(msg);
    }
    return error;
}

int Mealy:: countTransitions(void) const {
    int count = 0;
    if (valid()) {
        for (int s = 0; s < info.nstates; s++)
            for (int i = 0; i < input_set.nTerms(); i++) {
                StateIndex ns = delta_function.get(i, s);
                if (ns) count++;
            }
    }
    return count;
}

11***************************
Moore *************************************
11
11 // public member functions:
11
Moore:: Moore(void) : StateMachine(), Streamable() {
    info.Ctype = Mooretype;
}

Moore:: Moore(Moore& M) : StateMachine( (StateMachine&) M), Streamable() {
    info.Ctype = Mooretype;
}

Moore:: Moore(type M) : StateMachine( StateMachine& M), Streamable() {
}

Moore:: "Moore(void) ( flush(); )

void Moore:: inputFrom( istream& in) {
    char CS[80], NS[80];
    String * sr;
    int i, cs, ns, 0;
    stringstream msg; //stream for error messages
    cerr << "Reading input:\n";
    if (!in.good()) {
        msg << "Inputstream not usable";

else ( 
  cs = state_set.addState(*str); 
  o = output_set.addOutputTerm(*O); 
  if (0 == -1) { 
  cerr << "Error was encountered for output in line" << info.linenum << endl; 
  ok = FALSE; 
  goto skip_line; 
  } 
  in >> ws; 
) 
else if (in.peek() ==='#' || in.peek() == '"') // skip comments 
  in.ignore(MAXINT, '\n'); 
else if (in.peek() == '.') (
  in.ignore(); // '.' 
  if (in.get() == 'e') break; // end is reached 
  else { 
  msg << "(line " << info.linenum << ")
  No commands allowed here;
  Error(msg); 
  ok = FALSE; // command after transitions 
  goto skip_line; 
  } 
} 
else { 
  stringstream linestream; 
  toLine(linestream, in); 
  info.linenum++; 
  linestream >> *0; 
  Term * 1 = new Term(info.ninputs); 
  Output * 0 = new Output(info.noutputs); 
  if (in.eof()) {
  break;
  } 
  else if (in.peek() == '#') // test if input is don't care 
  msg << '"("line " << info.linenum << 
  Input should be don't care at line" << info.linenum << ".
  State set overflow";
  Error(msg); 
  ok = FALSE; 
  goto skip_line; 
  } 
  else if (in.peek() == '!') { 
  if (in.peek() == '!') { 
  cerr << "Error was encountered in line" << info.linenum << endl;
  ok = FALSE; 
  goto skip_line; 
  } 
  else if (state_set.isVoid(CS)) { 
  msg << "(line " << info.linenum << 
  'void' is not allowed for current state in Moore machine;"
  Error(msg);
  ok = FALSE;
  goto skip_line; 
  } 
  else if (state_set.isVoid(NS)) { 
  str = new String(CS); 
  if (state_set.isFull() && state_set.hasMember("str")) { 
  msg << "(line " << info.linenum << 
  State set overflow";
  Error(msg);
  ok = FALSE;
  goto skip_line; 
  } 
  else { 
  c = state_set.addState(*str); 
  o = output_set.addOutputTerm(*O); 
  if (o == -1) { 
  cerr << "Error was encountered for output in line" << info.linenum << endl;
  ok = FALSE;
  goto skip_line; 
  } 
  Term DC(info.ninputs); // create a don't care term 
  if ('!' == DC) { // test if input is don't care 
  msg << "(line " << info.linenum << 
  input should be don't care;"; 
  }
Warning(msg);

str = new String(NS);
if (state_set.isFull() && !state_set.hasMember(*str)) {
  msg << "(line " << info.linenumber << ") State set overflow";
  Error(msg);
  ok = FALSE;
  goto skip_line;
}

for (int s = 0; s < info.nstates; s++) {
  delta_function.set(i, s, ns);
}

strstream msg;
msg << "Reading input failed";
Fatal(msg);
else cerr << "Reading input completed successfully\n\n";

void Moore :: printOn( ostream& out) const {
  cerr << "Printing Moore machine:\n";
  if (validO) {
    Generallnformation infoCopy(info);
  }
infoCopy.ntransitions = countTransitions();
out << infoCopy;
for(int s = 0; s < info.nstates; s++)
    for (int i = 0; i < input_set.nTerms(); i++) {
        StateIndex ns = delta_function.get(i,s);
        if (ns) { // not undefined
            out << input_set[i] << " ";
            out << state_set.StateName(s) << " ";
            out << state_set.NextStateName(ns) << " ";
            out << output_set.zeroOutputTerm() << endl;
        }
    }
for (s = 0; s < info.nstates; s++) {
    OutputIndex o = labda_function.get(s);
    if (o) // output is specified
        out << "e" << endl;
else cerr << "Moore machine not valid\n\n";
}
bool Moore :: valid(void) const {
    return validStateMachine() && labda_function.valid();
}
void Moore :: checkFunctions(void) {
    cerr << "Checking definition of delta function:\n";
    if (valid()) {
        bool error = checkDeltaFunction();
        if (error) {
            cerr << "Result: Data is deleted\n\n";
            flush();
        }
    } else cerr << "Result: Delta function is ok\n\n";
else cerr << "Result: Moore machine not valid\n\n";
}
void Moore :: completeFunctions(void) {
    if (valid())
        completeDeltaFunction();
else cerr << "Moore machine not valid\n\n";
}
void Moore :: multipleExclusive(void) {
    cerr << "Making circuit multiple exclusive ";
    if (valid()) {
        int error = 0;
        Term *T2;
        for (int i = 0; i < input_set.nTerms(); i++) {
            int limit = input_set.nTerms();
            for (int j = i+1; j < limit; j++) {
                Term Term_i(input_set[i]);
                Term Term_j(input_set[j]);
                Term *Common = new Term(Term_i * Term_j);
                if (*Common == 0) delete Common;
                else {
                    int n = 0;
                    while (n < info.ninputs) { // add non-overlapping terms
                        T2 = new Term(subtract(Term_i,Term_j,n));
                        int t2 = input_set.addTerm(T2);
                        error |= delta_function.merge(t2,i);
                        T2 = new Term(subtract(Term_j,Term_i,n));
                        int t1 = input_set.addTerm(T1);
                        error |= delta_function.merge(t1,j);
                        if (*Common == 0) delete Common;
                    } else { // *Common != 0; multiple exclusive
                        if (Term_j < Term_i) {
                            int n = 0;
                            delete Common;
                            while (n < info.ninputs) { // add non-overlapping terms
                                T2 = new Term(subtract(Term_i,Term_j,n));
                                T2 = new Term(subtract(Term_j,Term_i,n));
                            } else if (Term_i < Term_j) {
                                int n = 0;
                                delete Common;
                            }
                        }
                    }
                }
            }
        }
    }
}
```cpp
int t2 = input_set.addTerm(T2);
error |= delta_function.merge(t2,j);
} else {
    int n = 0;
    while (n < info.ninputs) { // add non-overlapping terms
        T2 = new Term(subtract(Term_i,*Common,n));
        int t2 = input_set.addTerm(T2);
        error |= delta_function.merge(t2,j);
    }
    n = 0;
    while (n < info.ninputs) { // add non-overlapping terms
        T2 = new Term(subtract(Term_j,*Common,n));
        int t2 = input_set.addTerm(T2);
        error |= delta_function.merge(t2,j);
    }
    input_set.setTerm(i,*Common); // replace Input[i] by common term
    error |= delta_function.merge(i,j);
    int x = input_set.findMember(*Common);
    input_set.setTerm(i,*Common); // replace Input[i] by common term
    error |= delta_function.merge(i,j);
    if (x > 0) {
        error |= delta_function.merge(i,x-1);
        input_set.deleteTerm(x-1);
        delta_function.deleteForInput(x-1);
        if ((x-1) < j) j--;
        if ((x-1) < (limit) limit--;
        if ((x-1) < i) i--;
    } else cerr << ".",
    input_set.deleteTerm(j);
    delta_function.deleteForInput(j);
    limit--;
    j--;
    cerr << ".",
} if (!error) cerr << "Result: Circuit is multiple exclusive\n\n";
else cerr << "Result: Circuit is ambiguous\n\n";
else cerr << "Result: Moore machine not valid\n\n";
} // multipleExclusive
void Moore :: establishInverseLabda(void) {
    if (valid()) {
        inverse_labda.init(output_set.nTerms());
        for (int s = 0; s < info.nstates; s++)
            inverse_labda.set(labda_function.get(s),s);
        else cerr << "Moore machine not valid\n\n";
    } // establishInverseLabda

    void Moore :: printlnverseBetaOnCostream& out) const {
        if (inverse_labda.validC) {
            StateIndex cs;
            bool EndOfList;
            for (int o = 0; o < output_set.nTerms(); o++) {
                ListIterator &it = inverse_labda.initIterator(o);
                out << output_set[o] << " is output for following states: " << endl;
                EndOfList = inverse_labda.get(it,cs);
                while (!EndOfList) {
                    out << "\n" " <<state_set.StateName(cs) << "\n" "
                    EndOfList = inverse_labda.get(it,cs);
                } out << endl;
                delete &it;
            } else cerr << "Create inverse beta first\n\n";
        } // printlnverseBetaOn
    } // printlnverseBetaOn
```
void Moore :: init(void) {
  info.Ctype = Mooretype;
  initStateMachine();
  labda_function.init(info.nstates);
} // initMoore

void Moore :: flush(void) {
  labda_function.flush();
  inverse_labda.flush();
} // flushMoore

bool Moore :: checkInfo(void) {
  bool ok = TRUE;
  if (!info.nstates) {
    stringstream msg;
    msg << "No number of states specified for Moore machine";
    Fatal(msg);
    ok = FALSE;
  }
  if (Iinfo.ntransitions) info.ntransitions = STATE_TABLE_SIZE;
  info.transitiondelta = STATE_TABLE_DELTA;
  return ok;
} // checkInfo

int Moore :: countTransitions(void) const {
  int count = 0;
  if (validO) {
    for(int s = 0; s < info.nstates; s++)
      for (int i = 0; i < input_set.nTerms(); i++) {
        StateIndex ns = delta_function.get(i,s);
        if (ns) count++;
      }
    for (s = 0; s < info.nstates; s++)
      OutputIndex o = labda_function.get(s);
    if (o) count++;
  }
  return count;
} // countTransitions
In General Information some information is stored, that is useful for the program and the programmer.

Project: logic
File: implementation of three logic circuits: Pla, Mealy- and Moore-machine

By newbits

Index Loc

Oldbits In All Terms

A Don't Care Term

In Set

Constructor

Copy Constructor

Destructor

Destroy Data

Return If Valid

Return Term At Substitute All

Return If T Is

Return Number Of Terms In Set

Number Of Bits In Terms

Default Constructor

Read File Header In KIS/PLA Format

Print File Header In KIS/PLA Format

General Information (void);
General Information (General Information&);
GeneralInformation (void);
void flush (void);
bool valid (void) const;
void inputFrom (istream&);
void printOn (ostream& ) const;

Private:

void readOutputTypeFrom (istream&);
void printOutputTypeOn (ostream& out) const;
void printCircuitTypeOn (ostream& out) const;
void readPhaseFrom (istream&);
void readInputNamesFrom (istream&, istream&);
void readOutputNamesFrom (istream&);

Term zeroTerm() const;
int addTerm (Term & T);
int merge_addTerm (Term& T);
void deleteTerm (int loc);
void setTerm (int loc, const Term&);
int nTerms(void) const;
Term& operator () (int loc) const;
void subst(bit oldb, bit newb);
bool isDontcare(Term&) const;
bool hasMember(Term&) const;

General Information (void);
General Information( GeneralInformation& );
'GeneralInformation(void);
void flush(void);
bool valid(void) const;
void inputFrom (istream&);
void printOn (ostream& ) const;

Functions

End
```cpp
int findMember(Term& t) const; // return pos+1 if t is in set, else 0
void set( InputIndex i, StateIndex curs, StateIndex nexts);

InputSet is derived from TermSet and represents the set of input-term of a
sequential machine.

InputSet(void); // constructor
InputSet( InputSet& ); // constructor that copies
InputSet( void); // destructor
void flush();
bool valid() const;

OutputType type;

OutputSet(void); // constructor
OutputSet( OutputSet& ); // constructor that copies
OutputSet( void); // destructor
void init(int nb, int nt, int delta, OutputType); // allocates space for nt outputs of nb bits

InputSet( void); // constructor
InputSet( InputSet& ); // copy constructor
-InputSet( void); // destructor

StateSet( void); // constructor
StateSet( StateSet& ); // copy constructor
StateSet( void); // destructor
void flush();
bool valid() const;

OutputType type;

OutputSet(void); // constructor
OutputSet( OutputSet& ); // copy constructor
OutputSet( void); // destructor
void init(int nb, int nt, int delta, OutputType); // allocates space for nt outputs of nb bits

string addOutputTerm( Term& T);
Output operator [] (int o) const;
Output zeroOutputTerm() const { return (Output) zeroTerm(); }

void establishIrrelevant(void);
II substitutes irrelevant bits

OutputSet is also derived from TermSet and represents the set
of output terms of a sequential machine.

OutputSet(void); // constructor
OutputSet( OutputSet& ); // copy constructor
OutputSet( void); // destructor
void init(int nb, int nt, int delta, OutputType); // allocates space for nt outputs of nb bits

void isDontcare( String& ) const;
bool isVoid( String ) const;
bool isFull(void) const;
bool hasMember( String str) const;

void printOn(ostream& out) const { out

StateSet represents the set of states of a sequential machine.

class StateSet { // class StateSet
private:
    StringArray* State; // array of states
    int ns; // number of states

public:
    StateSet(void); // constructor
    StateSet( StateSet& ); // copy constructor
    StateSet( void); // destructor
    void flush();
    bool valid() const;

    int addState( String& ); // add a state to the set of states,
    int addNextState( String& ); // add a next-state to the set of states
    int findMember(Term& t) const; // return pos+1 if t is in set, else 0

    String& StateName( int loc) const; // returns the index
    StateName(int ns) const; // returns state at index loc
    StateName(int ns-2 or "*" if not in set

    String& NextStateName( int ns) const; // returns state at index loc
    NextStateName(int ns-2 or "**" if not in set

    void printOn(ostream& out) const ( out << *State << endl; }

}; // class StateSet

// The class DeltaFunction stores the IxS -> S (state transition) function

class DeltaFunction { // class DeltaFunction
private:
    StateTable* delta; // table of delta function
    int ns; // number of states

public:
    DeltaFunction(void); // constructor
    DeltaFunction( DeltaFunction& ); // copy constructor
    DeltaFunction( void); // destructor
    void init(int size, int nst, int sizedelta); // allocate space
    void flush(); // destroy data
    bool valid(void) const; // return if function is valid

    void set( InputIndex i, StateIndex curs, StateIndex nexts);}
```
The class LabdaFunction stores the S -> \delta (output) function of a Moore machine.
class LabdaFunction {
  private:
    OutputIndexArray * labda;
  public:
    LabdaFunction(void);          // constructor
    LabdaFunction( LabdaFunction& ); // copy constructor
    "LabdaFunction(void);         // destructor
    void init(int size);          // allocate space
    void flush(void);             // destroy data
    bool valid(void) const;       // return if function is valid

  void set(input index i, StateIndex cs, OutputIndex o);
  // set output for i and cs to o
  OutputIndex get(input index i, StateIndex cs) const;
  // return output for i and cs
  int merge(input index i, input index j);
  // merge function for inputs i and j
  int distinct(input index i, input index j) const;
  // return if there is no collision in outputs for input i and j
  void deleteForInput(input index i);
  // delete function for input i
}; // class LabdaFunction

The class BetaFunction stores the IxS -> I (output) function of a Mealy machine.
class BetaFunction {
  private:
    OutputTable * beta;
  public:
    BetaFunction(void);          // constructor
    BetaFunction( BetaFunction& ); // copy constructor
    ~BetaFunction(void);         // destructor
    void init(int size, int nstates, int delta); // allocate space
    void flush(void);             // destroy data
    bool valid(void) const;       // return if function is valid

  void init(int size);
  void flush(void);
  bool valid(void) const;
}; // class BetaFunction

The class DeltaFunction stores the IxS -> O (output) function of a Mealy machine.
class DeltaFunction {
  private:
    IndexPairTable * inverse_delta;
  public:
    DeltaFunction(void);          // constructor
    DeltaFunction( DeltaFunction& ); // copy constructor
    ~DeltaFunction(void);         // destructor
    void init(int size);
  // allocate space
}; // class DeltaFunction

InverseDelta also stores the delta function. With InverseDelta one has fast access from a given nextstate. (inverse delta is a S -> IxS relation)
class InverseDelta {
  private:
    IndexPairTable * inverse_delta;
  public:
    InverseDelta(void);          // constructor
    InverseDelta( InverseDelta& ); // copy constructor
    ~InverseDelta(void);         // destructor
    void init(int size);          // allocate space

  int merge(input index i, input index j);
  // merge function for inputs i and j
  return error state+1
  int distinct(input index i, input index j) const;
  // return if there is no collision in transitions for input i and j
  void deleteForInput(input index i);
  // delete function for input i
}; // class InverseDelta

// set index of nextstate
StateIndex get(input index i, StateIndex cs) const;
// return index of nextstate
OutputIndex get(input index i, StateIndex cs) const;
// merge function for inputs i and j
int merge(input index i, input index j);
// put result in i, return error state+1
int distinct(input index i, input index j) const;
// return if there is no collision in transitions for input i and j
bool isundefined(input index i, StateIndex cs) const;
// return if nextstate for i and cs is not defined

void deleteForInput(input index i);
// delete function for input i
void merge(input index i, input index j);
// merge beta function for input i and j to i
return error state+1
bool distinct(input index i, input index j) const;
// return if there is no collision in outputs for input i and j
void deleteForInput(input index i);
// delete function for input i

void flush(void); // destroy data
bool valid(void) const; // return if function is valid

void set(StateIndex ns, InputIndex i, StateIndex cs);
// add ns -> (i,cs) to table
ListIterator& initIterator(StateIndex s) const;
// return a ListIterator for the list of state s
bool get(ListIterator& i, StateIndex& cs) const;
// return next (i,cs) pair for nextstate ns and if list is ended
void printOn(ostream& ) const;
// print inverse delta table

II

void set(OutputIndex 0, InputIndex i, StateIndex cs);
// add 0 -> (i,cs) to table
ListIterator& initIterator(OutputIndex 0) const;
// return a ListIterator for the list of output 0
bool get(ListIterator&, InputIndex& i, StateIndex& cs) const;
// return next (i,cs) pair for output 0 and if list is ended

II

bool valid(void) const;
// return if function is valid

II

is valid
II
// destroy data
II
return if function is valid
II
the table
II
number of outputs
II
constructor
II
copy constructor
II
destructor
II
allocate space
II
destroy data
II
return if function is valid
II
add o -> (i,cs) to table
II
return a ListIterator for the list of output o
II
return next (i,cs) pair for output o and if list is ended
II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II

II
Generallnformation::Generallnformation() { flush(); }
linestream >> command;

if (linestream.good()) {
    if (strcmpi(command, "i") == 0)
        if (ninputs == 0)
            linestream >> ninputs;
        else {
            stringstream msg;
            msg << "\(\text{line } " << linenumber << "\) Number of inputs " << "has to be specified before .i phase";
            Error(msg);
        }
    else if (strcmpi(command, "o") == 0)
        if (noutputs == 0)
            linestream >> noutputs;
        else {
            stringstream msg;
            msg << "\(\text{line } " << linenumber << "\) Number of outputs " << "has to be specified before .o phase";
            Error(msg);
        }
    else if (strcmpi(command, "p") == 0)
        if (ntransitions == 0)
            linestream >> ntransitions;
        else {
            stringstream msg;
            msg << "\(\text{line } " << linenumber << "\) Number of inputs " << "has to be specified before .p phase";
            Error(msg);
        }
    else if (strcmpi(command, "s") == 0)
        if (nstates == 0)
            linestream >> nstates;
        else {
            stringstream msg;
            msg << "\(\text{line } " << linenumber << "\) Number of states " << "has to be specified before .s phase";
            Error(msg);
        }
    else if (strcmpi(command, "phase") == 0) {
        if (phase == NULL)
            if (noutputs == 0) {
                stringstream msg;
                msg << "\(\text{line } " << linenumber << "\) Number of outputs " << "has to be specified before .phase";
                Error(msg);
            }
            else {
                readPhaseFrom(linestream);
                if (linestream.fail() && !linestream.eof())
                    linestream.ignore(MAXINT);
            }
        else {
            stringstream msg;
            msg << "\(\text{line } " << linenumber << "\) Number of inputs " << "has to be specified before .ilb";
            Error(msg);
            linestream.ignore(MAXINT);
        }
    }
}
else if (strcmpi(command, "ob") == 0) {
    if (OutputNames == NULL)
        if (noutputs == 0) {
            strstream IIISg;
            msg << "(line " << linenumber << ") Number of outputs"
                << "has to be specified before .ob":
            Error(msg);
            in.clear(ios::failbit);
        } else readOutputNamesFrom(linestream, in);
    else {
        strstream msg;
        msg << "(line " << linenumber << ") Duplicated .ob directive";
        Warning(msg);
        }
    } else if (strcmpi(command, "uh") == 0) {
        if (ninputs < 0 || noutputs < 0 || nstates < 0 || ntransitions < 0) {
            strstream IIISg;
            msg << "(line " << linenumber << ") Remaining parameters allowed";
            Warning(msg);
            ninputs = 0;
        } else if (linestream.good())
            linestream >> ws; // eat whitespace
        if (linestream.good()) { // still something on stream
            strstream msg;
            msg << "(line " << linenumber << ") Only one command per line allowed"
                << "command: ";
            Warning(msg);
            linestream.ignore(MAXINT);
        } else 
            strstream msg;
            msg << "(line " << linenumber << ") Rest of command expected";
            Error(msg);
        } // if dot command
        else break; // end of dot directives
    } // while
}

if (!Otype) {
    cerr << "No type was specified; Type was made unknown\n";
    Otype = Pla_Unknown;
}

if (ninputs > MAX_VARIABLES) {
    strstream msg;
    msg << "Number of input bits is larger than MAX_VARIABLES (see l_defs.h)";
    Error(msg);
    in.clear(ios::failbit);
}

if (noutputs > MAX_VARIABLES) {
    strstream msg;
    msg << "Number of output bits is larger than MAX_VARIABLES (see l_defs.h)";
    Error(msg);
    in.clear(ios::failbit);
}

if (ninputs == 0) {
    strstream msg;
    msg << "Number of input bits must be specified";
    Error(msg);
}
in.clear(ios::failbit);
if (noutputs == 0) {
    stringstream msg;
    msg << "Number of outputbits must be specified";
    Error(msg);
}
in.clear(ios::failbit);
}

void GeneralInformation :: printHeaderInformation(ostream& out) const {
    out << "#
printCircuitTypeOn(out);
out << endl;
if (ninputs) out << ".i " << ninputs << endl;
if (noutputs) out << ".o " << noutputs << endl;
if (Otype) {
    if ((Ctype == Mooretype || Ctype == Mealytype) && (Otype == fr)) {
        out << ".type ";
printOutputTypeOn(out);
        out << endl;
    }
if (ntransitions) out << ".t " << ntransitions << endl;
if (nstates) out << ".s " << nstates << endl;
if (phase) out << ".phase " << *phase << endl;
if (InputNames) out << ".iib " << *InputNames << endl;
if (OutputNames) out << ".ob " << *OutputNames << endl;
}
}

// private member functions:
void GeneralInformation :: readOutputTypeFrom(istream& in) {
    char atype[5];
in >> setw(5) >> atype;
if (strcmp(atype, "f") == 0) Otype = f;
else if (strcmp(atype, "r") == 0) Otype = r;
else if (strcmp(atype, "fd") == 0) Otype = fd;
else if (strcmp(atype, "dr") == 0) Otype = dr;
else if (strcmp(atype, "fr") == 0) Otype = fr;
else if (strcmp(atype, "fdr") == 0) Otype = fdr;
else {
    stringstream msg;
    msg << "(line " << linenumber << ") Unknown type";
    Error(msg);
}
}

void GeneralInformation :: printOutputTypeOn(ostream& out) const {
    if (Otype == f) out << "f";
    else if (Otype == r) out << "r";
    else if (Otype == fd) out << "fd";
    else if (Otype == dr) out << "dr";
    else if (Otype == fr) out << "fr";
    else if (Otype == fdr) out << "fdr";
}

void GeneralInformation :: printCircuitTypeOn(ostream& out) const {
    if (Ctype == Unknown) out << "Unknown machine type";
    else if (Ctype == Platype) out << "Pla";
    else if (Ctype == Sequential) out << "Sequential Machine";
    else if (Ctype == Mooretype) out << "Moore Machine";
    else if (Ctype == Mealytype) out << "Mealy Machine";
}

void GeneralInformation :: readPhaseFrom(istream& in) {
    phase = new Term(noutputs);
    for (int i = 0; (in.good()) && (i < noutputs); i++) {
        char c;
in >> ws;
in >> c;
switch (c) {
    case '0': (*phase).set_var(i,zero);
    break;
    case '1': (*phase).set_var(i,one);
}
File l_set_fu.cpp Created at 10:16pm on Sunday, January 8, 1995 goossens Page 5 of 16

```cpp
void GeneralInformation::readInputNamesFrom(istream& in, istream& main) {
    InputNames = new StringArray(ninputs-1);
    char s[80];
    String * str;
    while (in.good()) {
        in >> setw(80) >> s;
        if (InputNames->isFull()) {
            stringstream msg;
            msg << "(line " << linenumber << ") Too many input names;"
            Error(msg);
            return;
        } else {  
            str = new String(s);
            if (InputNames->hasMember(*str)) {
                stringstream msg;
                msg << "(line " << linenumber << ") Duplicate input name " << *str << ";"
                Error(msg);
            } else {
                InputNames->add(*str);
            }
        }
    }
    main >> ws;
}
```

```cpp
// readPhaseFrom

if (i < noutputs) {
    in.clear(ios::failbit);
    stringstream msg;
    msg << "(line " << linenumber << ") Only 0 or 1 allowed for .phase;"
    Error(msg);
} else {
    strstream linestream;
    toLine(linestream,main);
    linenumber++;
    while (linestream.good()) {
        linestream >> setw(80) >> s;
        if (InputNames->isFull()) {
            stringstream msg;
            msg << "(line " << linenumber << ") Too many bits after .phase;"
            Warning(msg);
        } else {
            str = new String(s);
            if (InputNames->hasMember(*str)) {
                stringstream msg;
                msg << "(line " << linenumber << ") Duplicate input name " << *str << ";"
                Warning(msg);
            } else {
                InputNames->add(*str);
            }
        }
    }
    main >> ws;
    while (main.peek() != ' ' & main.peek() != '#') { // names also on next line
        strstream linestream;
        toLine(linestream,main); // next line also input names
        linenumber++;
        while (linestream.good()) {
            linestream >> setw(80) >> s;
            if (InputNames->isFull()) {
                stringstream msg;
                msg << "(line " << linenumber << ") Too many input names;"
                Error(msg);
            } else {
                str = new String(s);
                if (InputNames->hasMember(*str)) {
                    stringstream msg;
                    msg << "(line " << linenumber << ") Duplicate input name " << *str << ";"
                    Warning(msg);
                } else {
                    InputNames->add(*str);
                }
            }
        }
        main >> ws;
    }{
```

```cpp
if (!InputNames->isFull()) {
    stringstream msg;
    msg << "(line " << linenumber << ") Too few input names;"
    Error(msg);
}
```
for (int i = InputNames->getItemsInContainer(); i < InputNames->arraySize(); i++) {
    str = new String("No_name");
    InputNames->add(*str);
}

}  // readInputNamesFrom

void GeneralInformation::readOutputNamesFrom(istream & in, istream & main) {
    OutputNames = new StringArray(noutputs-1);
    char s[80];
    string * str;
    while (in.good()) {
        in >> setw(80) >> s;
        if (OutputNames->isFull()) {
            stringstream msg;
            msg << "(line " << linenumber << ") Too many output names;"
            Error(msg);
        } else {
            str = new String(s);
            if (OutputNames->hasMember(*str)) {
                stringstream msg;
                msg << "(line " << linenumber << ") Duplicate output name " << *str << ";"
                Error(msg);
            } else {
                OutputNames->add(*str);
                in >> ws;
            }
        }
    }

    while (main.peek() != '.' && main.peek() != '#') {
        stringstream linestream;
        toLine(linestream,main);
        linenumber++;
        while (linestream.good()) {
            linestream >> setw(80) >> s;
            if (OutputNames->isFull()) {
                stringstream msg;
                msg << "(line " << linenumber << ") Too many output names;"
                Error(msg);
            } else {
                str = new String(s);
                if (OutputNames->hasMember(*str)) {
                    stringstream msg;
                    msg << "(line " << linenumber << ") Duplicate output name " << *str << ";"
                    Error(msg);
                } else {
                    OutputNames->add(*str);
                    linestream >> ws;
                }
            }
        }
    }

    for (int i = InputNames->getItemsInContainer(); i < InputNames->arraySize(); i++) {
        str = new String("No_name");
        InputNames->add(*str);
    }

}  // readOutputNamesFrom

TermSet::TermSet() : Terms(NULL), nb(0) {}
457 4 Terms = new TermArray(nt-1,0,deLta);
458 4 nb = nbits;
459 0 // constructor
460 0
461 0 TermSet :: TermSet() { flush(); }
462 0
463 0 void TermSet :: init(int nbits, int nt, int deLta) {
464 4 Terms = new TermArray(nt-1,0,deLta);
465 4 nb = nbits;
466 0 } // init
467 0
468 0 void TermSet :: flush() {
469 4 nb = 0;
470 0 if (Terms) delete Terms;
471 0 Terms = NULL;
472 0 } // flush
473 0
474 0 bool TermSet :: valid() const {
475 4 return (bool) Terms;
476 0 } // valid
477 0
478 0 Term TermSet :: zeroTerm() const {
479 4 Term T(nb);
480 0 for (int n = 0; n < nb; n++)
481 8 T.set_var(n,zero);
482 0 return T;
483 0 } // zeroTerm
484 0
485 0 int TermSet :: addTerm( Term & T) {
486 4 if ((Terms->arraySize() == Terms->getItemsInContainer()) && Terms->deLta()==0) {
487 8 stream msg;
488 8 msg << "Term set overfLow";
489 8 Error(msg);
490 8 return -1;
491 0 }
492 4 for (int t = 0; t < Terms->getItemsInContainer(); t++)
493 8 if (T.isEqual(*Terms)[t]) return t;
494 0 Terms->add(T);
495 0 return Terms->getItemsInContainer()-1;
496 0 } // addTerm
497 0
498 0 int TermSet :: merge_addTerm( Term & T) {
499 0 if ((Terms->arraySize() == Terms->getItemsInContainer()) && Terms->deLta()==0) {
500 8 stream msg;
501 8 msg << "Term set overfLow";
502 8 Error(msg);
503 8 return -1;
504 0 }
505 4 }
506 4 if (added = FALSE;)
507 4 for (int t = 0; t < Terms->getItemsInContainer(); t++)
508 8 Term m(Term::merge(T,(*Terms)[t]));
509 0 if (m) {
510 12 Terms->addAt(m,t); // add bigger (merged) term
511 12 added = t+1;
512 0 }
513 8 else if (T < (*Terms)[t]) added = t+1;
514 0 }
515 0 if (!added) {
516 0 Terms->add(T);
517 0 return Terms->getItemsInContainer()-1;
518 0 } // addTerm
519 0
520 0 void TermSet :: deleteTerm( int loc) {
521 4 Terms->destroy(loc);
522 0 } // deleteTerm
523 0
524 0 void TermSet :: setTerm( int loc, const Term & copy) {
525 4 Term * T = new Term(copy);
526 0 Terms->addAt(*T,loc);
527 0 } // setTerm
528 0
529 0 int TermSet :: nTerms(void) const {
void OutputSet::operator[] (int loc) const
{
    return (Term&) (Terms)[loc];
}

void OutputSet::substitute(bit oldb, bit newb)
{
    for (int n = 0; n < (int)Terms.size(); n++)
    {
        Output& out = (Output&) (Terms)[n];
        for (int i = 0; i < nb; i++)
        {
            if (out.get_var(i) == oldb)
                out.set_var(i, newb);
        }
    }
}

bool OutputSet::isDontcare(Term& T) const
{
    Term DC(nb);
    Term test(1);
    return (test == DC);
}

void OutputSet::init(int nb, int nt, int delta, OutputType t)
{
    TermSet::init(nb, nt, delta);
    type = t;
}

void OutputSet::addOutputTerm(Term& T)
{
    if (isDontcare(T)) return 0;
    else
    {
        int t = addTerm(T);
        if (t == -1) return -1; // overflow of OutputSet
        else return t+1;
    }
}

Output& OutputSet::operator[] (int 0) const
{
    if (0) return (Output&) TermSet::operator[] (0-1);
    else return 0;
}

OutputSet::OutputSet() : TermSet() {}
 establishedIrrelevant

/******************************* StateSet ****************************

StateSet :: StateSet() : State(NULL) {
    ns = 0;
    // copy constructor
} 

StateSet :: StateSet( StateSet& s) :
    State(s.State) {
    ns = s.ns;
} 

StateSet:: -StateSet() ( flush(); }

void StateSet :: init(int size) {
    State = new StringArray(size-1);
    ns = size;
} 

void StateSet :: flush() {
    if (State) delete State;
    State = NULL;
    ns = 0;
} 

bool StateSet :: valid() const {
    return ns && State;
} 

int StateSet :: addState(String& str) {
    for (int s = 0; s < State->getItemsInContainer(); s++) {
        if (str == (*State)[s]) return s; // found
    }
    State->add(str);
    return State->getItemsInContainer()-1;
} 

int StateSet :: addNextState(String& str) {
    if (isDontcare(str)) return 1;
    for (int s = 0; s < State->getItemsInContainer(); s++) {
        if (str == (*State)[s]) return s+2; // found
    }
    State->add(str);
    return State->getItemsInContainer()+1;
} 

String& StateSet :: StateName(int loc) const {
    return (*State)[loc];
} 

String StateSet :: NextStateName(int ns) const {
    if (ns == 0) return String("ud");
    if (ns == 1) return String("*");
    return (*State)[ns-2];
} 

bool StateSet :: isDontcare(String& str) const {
    return str == String("*") || str == String("-"));
} 

/* StateSet */

DeltaFunction:: DeltaFunction() : delta(NULL) {
    delta = NULL;
} 

DeltaFunction :: DeltaFunction() : delta(NULL) {
}
DeltaFunction :: DeltaFunction( DeltaFunction& df ) :
    delta(df.delta) {
    ns = df.ns;
    }
    } // copy constructor

DeltaFunction :: ~DeltaFunction() ( flush(); )

void DeltaFunction :: init(int size, int nst, int sizedelta) {
    delta = new StateTable(size-1,0,sizedelta);
    ns = nst;
    } // initDeltaFunction

void DeltaFunction :: flush(void) {
    if (delta) delete delta;
    delta = NULL;
    } // flushDeltaFunction

bool DeltaFunction :: valid(void) const {
    return (ns && delta);
    } // valid

void DeltaFunction :: set( InputIndex i, Statelndex cs, StateIndex nxs) {
    State *nexts = new State(nxs);
    if (i < delta->getItemsInContainer()) ( // old input
        (*delta)[i].addAt(*nexts, cs);
    ) else {
        StateArray * a = new StateArray(ns-1);
        a->addAt(*nexts, cs);
        delta->add(*a);
    }
    return 0;
    } // set

StateIndex DeltaFunction :: get( InputIndex i, Statelndex cs) const {
    if (i < delta->getItemsInContainer()) {
        return (*delta)[i][cs];
    } else {
        stringstream msg;
        msg << "Input not in range (DeltaFunction :: get)";
        Error(msg);
        return 0;
    }
    } // get

int DeltaFunction :: merge(InputIndex i, InputIndex j) {
    if (j < delta->getItemsInContainer()) {
        if (i < delta->getItemsInContainer()) ( // merge to old input
            int nsi, nsj;
            for (int t = 0; t < ns; t++) {
                nsi = (int) (*delta)[i][t];
                nsj = (int) (*delta)[j][t];
                if (nsi > 1 && nsj > 1 && nsi != nsj) { // report Error-state+1, 0 means ok
                    return t+1;
                } else {
                    State * s = new State(nsi);
                    (*delta)[i].addAt(*s, t);
                    return 0;
                }
            }
        )
        else ( // merge to new input
            StateArray *a = new StateArray(ns);
            int nsj;
            for (int t = 0; t < ns; t++) {
                nsj = (*delta)[j][t];
                if (nsj) {
                    State * s = new State(nsj);
                    a->addAt(*s, t);
                }
            }
        )
    )
    } // merge
```cpp
761 12  delta->add(*a);
762 12  return 0;
763 8  }
764 4  else {
765 8  stringstream msg;
766 8  msg << "Input not in range (DeltaFunction :: merge)";
767 8  Error(msg);
768 8  cout << j << " << delta->getItemsInContainer() << endl;
769 1  cin.get();
770 8  return 1;
771 4  }
772 0  } // mergeDeltaFunction
773 5  int DeltaFunction :: distinct(InputIndex i, InputIndex j) const {
774 4  if (j < delta->getItemsInContainer())
775 8  if (i < delta->getItemsInContainer())
776 12  int nsi, nsj;
777 12  for (int t = 0; t < ns; t++)
778 16  nsi = (int) (*delta)[i][t];
779 16  nsj = (int) (*delta)[j][t];
780 16  if (nsi > 1 && nsj > 1 && nsi != nsj)
781 0  return t+1;  // report error-state+1, 0 means ok
782 16  }
783 20  return 0;  // report ok
784 8  else {
785 12  stringstream msg;
786 12  msg << "Input not in range (DeltaFunction :: distinct)";
787 8  Error(msg);
788 8  return 1;
789 8  }
790 4  else {
791 8  stringstream msg;
792 8  msg << "Input not in range (DeltaFunction :: distinct)";
793 8  Error(msg);
794 8  return 1;
795 4  }
796 8  }
797 8  else {
798 8  stringstream msg;
799 8  msg << "Input not in range (DeltaFunction :: distinct)";
800 8  Error(msg);
801 8  return 1;
802 4  }
803 4  void DeltaFunction :: deleteForInput(InputIndex i) {
804 4  if (i < delta->getItemsInContainer())
805 8  delta->destroy(i);
806 4  }
807 4  else {
808 8  stringstream msg;
809 8  msg << "Input not in range (DeltaFunction :: deleteForInput)";
810 8  Error(msg);
811 4  }
812 0  } // deleteDeltaFuncForInput
813 0  }
814 0  BetaFunction ****************************
815 0  BetaFunction :: BetaFunction() : beta(NULL) {}
816 0  BetaFunction :: BetaFunction( BetaFunction& Lf ) : beta(Lf.beta) {
817 0  ns = Lf.ns;
818 0  } // copy constructor
819 0  void BetaFunction :: init(int size, int nst, int delta) {
820 4  beta = new OutputTable(size-1,0,delta);
821 4  ns = nst;
822 0  } // initBetaFunction
823 0  void BetaFunction :: flush(void) {
824 0  beta = NULL;
825 4  if (beta) delete beta;
826 4  ns = 0;
827 4  BetaFunction :: flush();
828 0  } // flush
829 0  BetaFunction :: ~BetaFunction() { flush(); }
bool BetaFunction :: valid(void) const {
    return (ns && beta);
}

void BetaFunction :: set(InputIndex i, StateIndex cs, OutputIndex o) {
    OutputIndexType *oi = new OutputIndexType(o);
    if (i < beta->getItemsInContainer()) { // old input
        (*beta)[i].addAt(*oi, cs);
    } else { // new input
        OutputIndexArray * a = new OutputIndexArray(ns);
        a->addAt(*oi, cs);
        beta->add(*a);
    }
}

OutputIndex BetaFunction :: get(InputIndex i, StateIndex cs) const {
    if (i < beta->getItemsInContainer()) {
        return (*beta)[i][cs];
    } else {
        stringstream msg;
        msg << "Input not in range (BetaFunction :: get)"; Error(msg);
        return 0;
    }
}

int BetaFunction :: merge(InputIndex i, InputIndex j) {
    if (j < beta->getItemsInContainer()) {
        if (i < beta->getItemsInContainer()) { // merge to old input
            int oi, oj;
            for (int t = 0; t < ns; t++) {
                oi = (int) (*betaHi)[t];
                oj = (int) (*betaHj)[t];
                if (oi > 0 && oj > 0 && oi != oj) {
                    return t+1; // report error-state+1, 0 means ok
                }
                if (oi < 1 && oj > 0) {
                    OutputIndexType * o = new OutputIndexType(oj);
                    (*beta)[i].addAt(*o, t);
                }
            }
            beta->add(*a);
            return 0;
        } else { // merge to new input
            OutputIndexArray * a = new OutputIndexArray(ns);
            int oj;
            for (int t = 0; t < ns; t++) {
                oj = (*beta)[j][t];
                if (oj) {
                    OutputIndexType * o = new OutputIndexType(oj);
                    a->addAt(*o, t);
                }
            }
            beta->add(*a);
            return 0;
        }
    } else {
        stringstream msg;
        msg << "Input not in range (BetaFunction :: merge)"; Error(msg);
        return 1;
    }
}

bool BetaFunction :: distinct(InputIndex i, InputIndex j) const {
    if (j < beta->getItemsInContainer()) {
        if (i < beta->getItemsInContainer()) { // merge to old input
            int oi, oj;
            for (int t = 0; t < ns; t++) {
                oi = (int) (*betaHi)[t];
                oj = (int) (*betaHj)[t];
                if (oi > 0 && oj > 0 && oi != oj) {
                    return t+1; // report error-state+1, 0 means ok
                }
                if (oi < 1 && oj > 0) {
                    OutputIndexType * o = new OutputIndexType(oj);
                    (*beta)[i].addAt(*o, t);
                }
            }
            beta->add(*a);
            return 0;
        } else { // merge to new input
            OutputIndexArray * a = new OutputIndexArray(ns);
            int oj;
            for (int t = 0; t < ns; t++) {
                oj = (*beta)[j][t];
                if (oj) {
                    OutputIndexType * o = new OutputIndexType(oj);
                    a->addAt(*o, t);
                }
            }
            beta->add(*a);
            return 0;
        }
    } else {
        stringstream msg;
        msg << "Input not in range (BetaFunction :: merge)"; Error(msg);
        return 1;
    }
}
```cpp
if (oi > 0 && oj > 0 && oi != oj) {
    return t+1; // report error-state=1, 0 means ok
} else {
    stringstream msg;
    msg << "Input not in range (BetaFunction :: distinct)";
    Error(msg);
    return 1;
} else {
    stringstream msg;
    msg << "Input not in range (BetaFunction :: distinct)";
    Error(msg);
    return 1;
} return 0;
else {
    stringstream msg;
    msg << "Input not in range (BetaFunction :: distinct)";
    Error(msg);
    return 1;
}
}

void BetaFunction :: deleteForInput(InputIndex i) {
if (i < beta->getItemsInContainer()) {
    beta->destroy(i);
} else {
    stringstream msg;
    msg << "Input not in range (BetaFunction :: deleteForInput)";
    Error(msg);
    return 1;
}
}

110
```
```cpp
void InverseDelta::init(int size) {
    inverse_delta = new IndexPairTable(size-1);
    ns = size;
    IndexPairList * l;
    for (int s = 0; s < ns; s++) {
        l = new IndexPairList();
        inverse_delta->add(*l);
    }
}
```

```cpp
bool InverseDelta::valid() const {
    return ns && inverse_delta;
}
```
void InverseBeta::init(int size) {
  inverse_beta = new IndexPairTable(size-1);
  no = size;
  IndexPairList * l;
  for (int o = 0; o < no; o++) {
    l = new IndexPairList();
    inverse_beta->add(*l);
  }
}

bool InverseBeta::valid() const {
  return no && inverse_beta;
}

void InverseBeta::flush(void) {
  no = 0;
  if (inverse_beta) delete inverse_beta;
  inverse_beta = NULL;
}

bool InverseBeta::set(OutputIndex o, InputIndex i, StateIndex cs) {
  if ((o-1) < no && (o==0)) {
    IndexPair *P = new IndexPair(i,cs);
    (*inverse_beta)[o-1].add(*P);
  }
}

ListIterator& InverseBeta::initIterator(OutputIndex o) const {
  if (o < no)
    return (ListIterator&)(*inverse_beta)[o].initIterator();
  else {
    stringstream msg;
    msg << "Output not in range (InverseBeta :: initIterator)";
    Error(msg);
    return (ListIterator&)(*inverse_beta)[no-1].initIterator();
  }
}

bool InverseBeta::get(ListIterator& it, InputIndex & i, StateIndex & cs) const {
  if (!it) // end of list
    return 1;
  else {
    ((IndexPair&)it.current()).getIndexPair(i,cs);
    it++;
    return 0;
  }
}


InverseLabda InverseLabda::InverseLabda() {
  return inverse_labda(NULL);
}

InverseLabda InverseLabda::InverseLabda(INVERSELABDA il) {
  inverse_labda(il.inverse_labda) {
    no = il.no;
    return 1;
  }
}

InverseLabda InverseLabda::InverseLabda(void) (flush(); )

void InverseLabda::init(int size) {
  inverse_labda = new StateListArray(size-1);
  no = size;
  StateList * l;
  for (int o = 0; o < no; o++) {
    l = new StateList();
    inverse_labda->add(*l);
  }
}

void InverseLabda::flush(void) {
  no = 0;
  if (inverse_labda) delete inverse_labda;
  inverse_labda = NULL;
}

} // initInverseLabda
bool Inverselabda :: valid() const {
    return no && inverse_labda;
}

void Inverselabda :: set(OutputIndex o, StateIndex cs) {
    if ((o-1) < no) && (o<0)) {
        State *s = new State(cs);
        (*inverse_labda)[o-1].add(*s);
    }
}

ListIterator& Inverselabda :: initIterator(OutputIndex o) const {
    if (o < no)
        return (ListIterator&)(*inverse_labda)[o].initIterator();
    else {
        stringstream msg;
        msg << "Output not in range (Inverselabda :: initIterator)";
        Error(msg);
        return (ListIterator&)(*inverse_labda)[no-1].initIterator();
    }
}

bool Inverselabda :: get( ListIterator& it, StateIndex& cs) const {
    if (!it) // end of list
        return 1;
    else {
        cs = ((State&)it.current()).StateIndex();
        it++;
        return 0;
    }
}

// get
#ifndef _ELEMS_H
#define _ELEMS_H

#include <string.h>
#include <list.h>
#include <l_term.h>

//****************************************************************************
Element classes  
//****************************************************************************

These are the smallest elements in a logic circuit. These classes are derived from the Object class (Borland Container Library) so that they can be stored in arrays or lists. Term is also a element class, but because of its large implementation it is in a separate file (l_term.h).

The State class is an Object-like implementation of StateIndex. Now the deltafunction of a state-machine can be build of arrays of States.

class State: public Object {  
private:
  StateIndex st;
public:
  State(int sti = 0) : Object() { st = sti; }  // constructor  
  StateIndex StateIndex(void) const { return st; }  // returns the value  
  // Required by Object class from Container Library  
  virtual void printOn(ostream& out) const { out << (int) st; }
  virtual hashValueType hashValue() const { return 0; }
  virtual classType isA() const { return stateClass; }
  virtual char *nameOf() const { return "State"; }
  virtual int isEqual(const Object& testObject) const {
    State& s = (State&) testObject;
    return st == s.st;
  }
};  //class State

The OutputIndexType class is an Object-like implementation of OutputIndex. Now the labdafunction of a Mealy-machine can be build of arrays of OutputIndex.

class OutputIndexType: public Object {  
private:
  OutputIndex oi;
public:
  OutputIndexType(int o = 0) : Object() { o1 = o; }  //constructor  
  OutputIndex outputIndex(void) const { return oi; }  // returns value  
  // Required by Object class from Container Library  
  virtual void printOn(ostream& out) const { out << (int) oi; }
  virtual hashValueType hashValue() const { return 0; }
  virtual classType isA() const { return outputIndexClass; }
  virtual char *nameOf() const { return "OutputIndex"; }
  virtual int isEqual(const Object& testObject) const {
    OutputIndexType& o = (OutputIndexType&) testObject;
    return oi == o.oi;
  }
};  // class OutputIndexType

The IndexPair class is an Object-like implementation of a pair of indices. Now the inverse delta relation of a state-machine can be build of lists of IndexPairs.
class IndexPair : public Object {
  private:
  InputIndex inp;
  Statelndex cst;
  public:
    IndexPair(InputIndex inp = 0, Statelndex cst = 0) : Object() // constructor
    ( i = inp; cs = cst; ) // destructor
  void setIndexPair(InputIndex inp, Statelndex cst) {
    i = inp; cs = cst;
  }
  void getlndexPair(lnputlndex& inp, Statelndex& cst) {
    inp = i; cst = cs;
  }
  // Required by Object class from Container Library
  void printOn(ostream& out) const {
    out << "(" << (int) i << ", " << (int) cs << ") " << "IndexPair"");
  }
  virtual classType isA() const { return indexPairClass; }
  virtual char *nameOfO const { return "lndexPair"; }
  virtual int isEqual(const Object& testObject) const {
    IndexPair& ip = (IndexPair&) testObject;
    return (i == ip.i) && (cs == ip.cs);
  }
};

class StateArray : public Array, public Streamable {
  private:
    int ns;
  public:
    StateArray(int size) : Array(size, 0, 0), Streamable()
      ( ns = size;) // constructor
    StateIndex operator[](int loc) const {
      Object& 0 = Array:: operator[](loc);
      if (O == NOOBJECT) return 0;
      else return ((State&) O).StateIndex();
    }
    void printOn(ostream& ) const;
    void inputFrom(istream &) { cerr << "Can't read StateArray from input\n";)
  // Required by Object class from Container Library
    if (i == ip.i) return;
  virtual hashValueType hashValue() const (return 0;)
};

class StateList : public List {
  private:
    StateList(void) : List() {
      "StateList(void) ()
    void printOn(ostream& ) const;
    void inputFrom(istream &) { cerr << "Can't read OutputIndexArray from input\n";)
  // Required by Object class from Container Library
    if (i == ip.i) return;
  virtual hashValueType hashValue() const (return 0;)
};

class OutputIndexArray : public Array, public Streamable {
  private:
    int ns;
  public:
    OutputIndexArray(int size) : Array(size, 0, 0) (ns = size;)
};
OutputIndex operator[](int loc) const {
    Object& 0 = Array::operator[](loc);
    if (0 == NOOBJECT) return 0;
    else return ((OutputIndexType&) 0).outputIndex();
}
void printOn(ostream& ) const; // Required by Object class from Container Library
virtual hashValueType hashValue() const ( return 0; )

class IndexPairList : public List {
public:
    IndexPairList(void) : List() {}
    void printOn(ostream& ) const; // Required by Object class from Container Library
    virtual hashValueType hashValue() const ( return 0; )

}; // class IndexPairList

class StringArray : public Array {
public:
    StringArray(int size) : Array(size,0,0) ()
    String& operator[](int loc) const
    ( return (String&) Array::operator[](loc); )
    bool isFull(void) const ( return getItemsInContainer() == arraySize(); )
}; // class StringArray

class TernArray : public Array {
public:
    TernArray(int size, int lower = 0, int delta = 0) :
    Array(size, lower, delta) ()
    Term& operator[](int loc) const
    ( return (Term&) Array::operator[](loc); )
    sizeType delta() (return Array::delta; )
}; // class TernArray

class OutputArray : public Array {
public:
    OutputArray(int size, int lower = 0, int delta = 0) :
    Array(size, lower, delta) ()
    Output& operator[](int loc) const
    ( return (Output&) Array::operator[](loc); )
}; // class OutputArray

class StateTable : public Array {
public:
    StateTable(int size, int lower = 0, int delta = 0) :
    Array(size, lower, delta) ()
    StateArray& operator[](int loc) const
    ( return (StateArray&) Array::operator[](loc); )
}; // class StateTable

class OutputTable : public Array {
public:
    OutputTable(int size, int lower = 0, int delta = 0) :
    Array(size, lower, delta) ()
    OutputIndexArray& operator[](int loc) const
    ( return (OutputIndexArray&) Array::operator[](loc); )
}; // class OutputTable
class IndexPairTable : public Array {
  IndexPairTable(int size, int lower = 0, int delta = 0) :
      Array(size, lower, delta) {}  
  IndexPairList& operator[](int loc) const 
    { return (IndexPairList&) Array::operator[](loc); } 
};

class StateListArray : public Array {
  StateListArray(int size, int lower = 0, int delta = 0) :
      Array(size, lower, delta) {} 
  StateList& operator[](int loc) const 
     { return (StateList&) Array::operator[](loc); } 
};

#if 0
#endif
```cpp
#include <l elems.h>
#include <array.h>
#include <iomanip.h>

#include <l elems.cpp Created at 1:23pm on Wednesday, September 7, 1994 goossens Page 1 of 1

void StateArray::printOn(ostream& out) const {
    out << "[ ";
    for (int s = 0; s < arraySize(); s++)
        out << setw(2) << (int) (State&) objectAt(s).StateIndex() << " ";
    out << " ]";
}

void OutputIndexArray::printOn(ostream& out) const {
    out << "[ ";
    for (int s = 0; s < arraySize(); s++)
        out << setw(2) << (int) (OutputIndexType&) objectAt(s).outputIndex() << " ";
    out << " ]" << endl;
}

void IndexPairList::printOn(ostream& out) const {
    ListIterator it(*this);
    it.restart();
    while (it) {
        out << (IndexPair&) it.current() << " ";
        it++;
    }
}

void StateList::printOn(ostream& out) const {
    ListIterator it(*this);
    it.restart();
    while (it) {
        out << (State&) it.current() << " ";
        it++;
    }
}

void StringArray::printOn(ostream& out) const {
    for (int s = 0; s < getItemsInContainer(); s++)
        out << (String&) objectAt(s) << " ";
}
```

```cpp
#include <array.h>
#include <packarray.h>
#include <strmable.h>
#include <l_defs.h>

#ifndef L_TERM_H
#define L_TERM_H

*************************** ************** •••••• ************** ••••••

class Term : public Object, public Streamable, public BitVector {

protected:
    static const bitword one_mask;
    static const bitword dontcare_mask;
    static const bitword zero_mask;
    static const bitword irrelevant_mask;

    friend void delete_var(Object& o, void* v); // auxiliary for Pla:del_var

public:
    Term(int nvars = 0); // constructor that creates a don't care term
    Term(const Term& copy); // constructor that copies an other term
    Term(char *str); // constructor that reads the term from the string

    void set_var(int var, bit value) {
        put_elem(var, value);
    }

    bit get_var(int var) const {
        register res = get_elem(var);
        return ((res == irrelevant + 1) ? irrelevant : res);
    }

    bool minterm(void) const;
    bool valid(void) const ( return (nelems > 0); )
    virtual void printOn( ostream& outputStream ) const;
    virtual void inputFrom( istream& inputStream );

    virtual int isEqual( const Object& testObject ) const;
    // for implementing operator ==
    operator int(void) const ( return valid(); )

friend Term operator + (const Term& T1, const Term& T2);
```

---

```
Object of the class Term represents one product term in the function specification (of the form: x1 and x2 and ... and xn). Value "one" for a given variable means that the variable is the term in the positive form, value "zero" means that the variable is in the term but negated, value "don'tcare" means that the variable is not in the term. Value "irrelevant" is not used for Terms.
```

---

```
(C) Pawel Konieczny 1994
Remarks to: pawel@eb.ele.tue.nl
The class Term is derived from PackedArray. In objects of the class Term, terms of a combinatorial circuit can be stored. The class Output is derived from Term. Some special properties of output patterns are provided by this class.
```

---

```
typedef PackedArray<MAX_VARIABLES,2> BitVector;
```

---

...
friend Term operator *(const Term& T1, const Term& T2);
// gives the smallest term that covers the both operands
friend int operator < (const Term& T1, const Term& T2);
// returns 1 only if the first argument is completely covered
// or a not valid term if the operands do not overlap
friend Term subtract(const Term& T1, const Term& T2, int& i);
// returns the don't care term that
static Term merge(const Term& T1, const Term& T2);
// returns a term that covers exactly T1 and T2,
// a not valid term if it is not possible
bool Term::distinct(const Term& T) const;
// returns true if the terms cannot be merged
// (they have zero-one collision on the same variable)

// Required by Object class from Container Library
virtual hashValueType hashValue() const;
virtual classType isA() const { return termClass; }
virtual char *nameOf() const { return "Term"; }
virtual void printOn(ostream& outputStream) const;
virtual void inputFrom(istream& inputStream);

public:
Output(int nvars = 0);
Output(const Term& copy) : Term(copy) {
Output(char *str);
virtual void printOn(ostream& outputStream) const;
virtual void inputFrom(istream& inputStream);
bool distinct(const Output& T) const;
// the same as Term::distinct() but faster and assumes
// that none of the outputs contains a don't care value

// Required by Object
virtual classType isA() const { return outputClass; }
virtual char *nameOf() const { return "Output"; }

}; // Output

#endif
# include <l_term.h>
# include <l_support.h>
# include <string.h>
# include <iomanip.h>
# include <checks.h>

/**************************** BitVector *****************************/

II
Definitions of static constants for the most interesting instance
II
of PackedArray: BitVector.
II
For other instances the definitions would be exactly the same
II
number of bits needed to address an elem within a bitword
const int BitVector::offset_bits = log2(BITSPERBITWORD/elem_len);
II
mask for offset address of an elem within a bitword
const unsigned BitVector::offset_mask = BITSPERBITWORD/elem_len - 1;
II
mask for an admissible elem
const bitword BitVector::elem_mask = (1 « elem_len) - 1;
II

/**************************** Term *****************************/
II
one mask and dontcare mask represent all bits in a bitword to be one or don't care,
respectively.
II
The bit patterns are consistent with the enum bit type definition.
II
static bitword generate_one_mask()
II
{
    bitword res = 0x55; // "one"s in one byte (01010101)
    for (int i = BITSPERBYTE; i < BITSPERBITWORD; i *= 2)
        res = res « i;
    return res;
}
const bitword Term: :one_mask = generate_one_mask();
const bitword Term: :dontcare_mask = "one_mask;";
const bitword Term: :zero_mask = 0;
const bitword Term: :irrelevant_mask = "zero_mask;";
II
Term: :Term(int nnelems) : Object(), Streamable(), PackedArray<MAX_VARIABLES,2>()
II
PRECONDITION(nnelems <= MAX_VARIABLES);
nelems = nnelems;
for (register int n = 0; n < bitwords_used(); n++)
    elems[n] = dontcare_mask; // all variables don't care
II
Term: :Term(const Term& copy)
II
nelems = copy.nelems;
for (register int n = 0; n < bitwords_used(); n++)
    elems[n] = copy elems[n];
II
Term: :Term(char *str)
II
i strstream in(str);
nelems = 0;
inputFrom(in);
II
Term: :Term(char *)

int OK = 1;

Bit B;

return OK;

if (nelems < BITSPERBITWORD/2)
  value &= 0xFFFF >> (BITSPERBITWORD/2-nelems)*2;  // clear not used bits
return value;

// Term::hashValue

int Term::isEqual( const Object& testObject ) const
{
  const Term& second = (const Term&) testObject;
  int OK = (nelems == second.nelems);
  int limit = bitwords_used() - 1;
  for (int n = 0; OK && (n < limit); n++)
    OK = OK && (elems[n] == second.elems[n]);
  for(int i = limit*(BITSPERBITWORD/2); OK && (i < nelems); i++)
    OK = OK && (get_var(i) == second.get_var(i));
  return OK;
}

// Term::isEqual

bool Term::minterm(void) const
{
  int OK = 1;
  for (int i = 0; OK && i < nelems; i++)
    OK = OK && (get_var(i) != dontcare);
  return OK;
}

// Term::minterm

void Term::printOn( ostream& outputStream ) const
{
  if (valid())
  {
    bit B;
    for (int i = 0; i < nelems; i++)
      if ((B = get_var(i)) != dontcare)
        outputStream << (int) B;
      else
        outputStream << '-' << C;
    else
      outputStream << "<not valid>";
  }
}

// Term::printOn

void Term::inputFrom( istream& inputStream ) {
if (nelems!=0) {// normal situation
  for (int i = 0; inputStream.good() && (i < nelems);) {
    char c;
    inputStream >> c;
    switch (c) {
      case '0': set_var(i, zero); i++; break;
      case '1': set_var(i, one); i++; break;
      case '-': set_var(i, dontcare); i++; break;
      default : strstream msg;
        msg << "Only 0, 1, - allowed for terms";
    }
152 26   Error(msg);
153 26   inputStream.clear(ios::failbit);
154 26   return;
155 16   }
156 12   inputStream >> ws;
157 8   }
158 8   if (i < nelems) {
159 12   stringstream msg;
160 12   msg << "Too few bits in term " << *this;
161 12   Error(msg);
162 12   inputStream.putback(' '); // makes eof false
163 8   }
164 4   }
165 4   else {// inputFrom called from Term(char*)
166 8   int i = 0;
167 8   inputStream >> ws;
168 8   while(inputStream.good()) {
169 12   char c;
170 16   inputStream >> c;
171 16   switch (c) {
172 20   case '0': set_var(i, zero); i++; break;
173 20   case '1': set_var(i, one); i++; break;
174 20   case '-': set_var(i, dontcare); i++; break;
175 20   default: stringstream msg;
176 26   msg << "Only 0, 1, - allowed for terms";
177 26   Error(msg);
178 26   inputStream.clear(ios::failbit);
179 26   return;
180 16   }
181 12   inputStream >> ws;
182 8   }
183 8   nelems = i;
184 4   }
185 0   } // Term::inputFrom
186 0
187 0
188 0
189 0
190 0 // Static members of Term class
191 0
192 0
193 0   Term Term :: merge(const Term& T1, const Term& T2)
194 0   {
195 0   
196 8   if (T1 < T2) return T2;
197 8   if (T2 < T1) return T1;
198 0   
199 0   Term T(T1);
200 4
201 0
202 4   if (T2.nelems < T1.nelems)
203 7   T.nelems = T2.nelems;
204 0
205 0
206 4   // find a bit of difference
207 0
208 4
209 4   int i;
210 4   bit b1, b2;
211 8   for (i = 0; i < T.nelems; i++)
212 10   if ( (b1 = T1.get_var(i)) != (b2 = T2.get_var(i)) )
213 4   break;
214 0
215 4   if (i == T.nelems) return T; // T1 == T2
216 8   
217 0
218 4   if ( (b1 == dontcare) || (b2 == dontcare) )
219 8   return 0; // cannot merge
220 0
221 0
222 4   // is only one such a bit?
223 0
224 4
225 4
226 0
227 0   } // Term::merge(Term, Term)
const int limit = bitwords_used();

for (int n = 0; n < limit; n++)
{
    // we use the fact that all not used bits
    // in the last word are set don't care or irrelevant
    for (int n = 0; n < limit; n++)
    {
        # if VERSION == 1
            #ifdef VERSION 2
                slower
            #endif
        #endif

        bitword test = (elems[n] | one_mask ^ T.elems[n])
            &
            (elems[n] ^ one_mask | T.elems[n]);
        // test has 00 (only) in the place where
        // elems and T.elems have 00 and 01 or 01 and 00
        # if VERSION == 2
            #ifdef VERSION 2
                faster
            #endif
        #endif

        register bitword test = (elems[n] | one_mask ^ T.elems[n])
            &
            (elems[n] ^ one_mask | T.elems[n]);
        // test has 00 (only) in the place where
        // elems and T.elems have 00 and 01 or 01 and 00
        test |= (test & dontcare_mask) ^ 1; // all 10 become 11
        // if there is no 00-01 collision, test should have all odd bits set
        if (-(test & dontcare_mask))
        {
            return 1;
        }

        # if VERSION == 2
            #ifdef VERSION 2
                faster
            #endif
        #endif

        if (test & mask)
        {
            return 0; // not distinct
        }

        # endif
    }
}

Term operator + (const Term& T1, const Term& T2)
{
    if (!T1)
    {
        return T2;
    }

    if (!T2)
    {
        return T1;
    }

    Term P(T1.nelems);
    if (T2.nelems < P.nelems)
    {
        P.nelems = T2.nelems;
    }

    for (int i = 0; i < P.nelems; i++)
    {
        bit var1 = T1.get_var(i);
        bit var2 = T2.get_var(i);
        CHECK(var1 != irrelevant);
        CHECK(var2 != irrelevant);
        if (var1 == var2)
        {
            P.set_var(i, var1);
        }
        else
for (int i = 0; i < P.nelems; i++)
    (bit var1 = T1.get_var(i);
     bit var2 = T2.get_var(i);
     if (var1 -- irrelevant)
        P.set_var(i, var2);
     else
         if (var2 -- irrelevant)
             P.set_var(i, var1);
         else
             if (var1 == dontcare)
                 P.set_var(i, var2);
             else
                 if (var2 -- dontcare)
                     P.set_var(i, var1);
                 else
                     if (var1 == var2)
                         P.set_var(i, var1);
                     else
                         return 0;
    II not valid)

Term operator *(const Term& T1, const Term& T2)

    if ( !T1 || !T2) // not valid terms
        return 0;
    Term P(T1.nelems);
    if (T2.nelems < P.nelems)
        P.nelems = T2.nelems;
    for (int i = 0; i < P.nelems; i++)
        (bit var1 = T1.get_var(i);
         bit var2 = T2.get_var(i);
         if (var1 == irrelevant)
             P.set_var(i, var2);
         else
             if (var2 == irrelevant)
                 P.set_var(i, var1);
             else
                 if (var1 == dontcare)
                     P.set_var(i, var2);
                 else
                     if (var2 == dontcare)
                         P.set_var(i, var1);
                     else
                         if (var1 == var2)
                             P.set_var(i, var1);
                         else
                             return 0; // not valid
        )
    return P;

Term operator < (const Term& T1, const Term& T2)

    true if T2 covers T1

    int OK = T1.nelems <= T2.nelems;
    for (int i = 0; OK && i < T1.nelems; i++)
        (bit b2 = T2.get_var(i);
         if (b2 != dontcare)
             OK = T1.get_var(i) == b2;
        )
    return OK;

Term subtract(const Term& T1, const Term& T2, int& i) {
    Term T(T1.nelems);
    bit b;
    for (int n = 0; n < T1.nelems; n++)
        (if ((b = T1.get_var(n)) == (T2.get_var(n)))
            T.set_var(n,b);
        else
            if (n < i) {
                T.set_var(n,dontcare);
            } else {
                // t1[n] = '1' and T2[n] = 0 or 1

125
switch(T2.get_var(n)) {
    case zero: T.set_var(n, zero); break;
    case one: T.set_var(n, one); break;
    i = n+1;
    int same_tail = 1;
    for (int k = i; k < T1.nelems; k++)
        if ((b = T1.get_var(k)) == T2.get_var(k))
            T.set_var(k, b);
        else {
            T.set_var(k, dontcare);
            same_tail = 0;
        }
    if (same_tail) i = T1.nelems+1;
    return T;
}

if (n == T1.nelems) i = n+1;
return 0; // not valid;
} // subtract

Output::Output(int nelems) : Term(nelems)
{
    for (int n = 0; n < bitwords_used(); n++)
        elems[n] = irrelevant_mask; // all variables irrelevant
}

Output::Output(char *str) : Term(0)
{
    if (str) {
        istrstream in(str);
        nelems = 0;
        inputFrom(in);
    }
}

void Output::printOn( ostream& outputStream ) const
{
    for (int i = 0; i < nelems; i++)
        switch (get_var(i))
            case zero: outputStream << '0'; break;
            case one: outputStream << '1'; break;
            case dontcare: outputStream << '-'; break;
            case irrelevant: outputStream << '-'; break;
    return 0; // not valid;
}

void Output::inputFrom( istream& inputStream ) {
    if (nelems!=O) {
        II normal situation
        for (int i = O; inputStream.good() && (i < nelems);) {
            char c;
            inputStream >> c;
            switch (c) {
                case '0': set_var(i, zero); i++; break;
                case '1': set_var(i, one); i++; break;
                case '-': set_var(i, dontcare); i++; break;
                case '-': set_var(i, irrelevant); i++; break;
                default : stringstream msg;
                    msg << "Only 0, 1, - and - allowed for outputs";
                    Error(msg);
                    inputStream.clear(ios::failbit);
                    return;
            }
            inputStream >> ws;
        }
    }
}

if (i < nelems) {
    strstream msg;
    msg << "Too few bits in output " << *this;
    Error(msg);
    inputStream.putback(' '); // makes eof false
    inputStream.clear(ios::failbit);
}
else { // inputFrom called from Output(char*)
    int i = 0;
    inputStream >> ws;
    while(inputStream.good()) {
        char c;
        inputStream >> c;
        switch (c) {
            case '0': set_var(i, zero); i++; break;
            case '1': set_var(i, one); i++; break;
            case '-': set_var(i, dontcare); i++; break;
            default: strstream msg;
                msg << "Only 0, 1, - and - allowed for outputs";
                Error(msg);
                inputStream.clear(ios::failbit);
                return;
        }
        inputStream >> ws;
    }
    nelems = i;
}
bool Output::distinct(const Output& T) const {
    PRECONDITION(nelems == T.nelems);
    // we use the fact that all not used bits in the last word are set irrelevant
    int limit = bitwords_used();
    for (int n = 0; n < limit; n++) {
        // the following version works correctly only if
        // there are no don't care bits in the both outputs
        // this is always true for the case of the fd pla type
        // if this assumption is not true, Term::distinct() must be used
        bitword test = elems[n] - T.elems[n];
        // test has 01 (only) in the place where
        // elems and T.elems have 00 and 01 or 01 and 00
        // if one of bits were 10, it would have 01 also for 10 and 11
        // but we assume it never happens
        test = (test & dontcare_mask) >> 1; // all 10 become 11
        test = (test & dontcare_mask) >> 1; // all 11 (and prev. 10) become 10
        // if there is no 00-01 collision, test should have all odd bits clear
        if (test & one_mask)
            return 1;
    }
    return 0; // not distinct
}
// Output::distinct
# include <values.h>

/* CONSTANT DEFINITIONS */

/* TYPE DEFINITIONS */

// returns max possible elem's value + 1

// constructor

// Below, elem is the elem's index. Elem's value is of type bitword,
// as it is the largest possible type for elem.

// if wrong index

// returns max possible elem's value + 1
template <int maxelems, int elem_len>
inline void PackedArray<maxelems, elem_len>::put_elem(int elem, bitword value)
{
    if ((elem < nelems) && (elem >= 0))
    {
        register word = elem >> offset_bits; // bitword where elem is stored
        register bits_addr = ((elem & offset_mask) * elem_len); // offset in bitword
        // clear prev value
        elems[word] &= ~(elem_mask << bits_addr);
        // set new value
        elems[word] |= (value & elem_mask) << bits_addr;
    }
}

template <int maxelems, int elem_len>
inline bitword PackedArray<maxelems, elem_len>::get_elem(int elem) const
{
    if ((elem < nelems) && (elem >= 0))
    {
        register word = elem >> offset_bits;
        register bits_addr = ((elem & offset_mask) * elem_len);
        return (elems[word] >> bits_addr) & elem_mask;
    }
    else
    return (1 << elem_len); // error value: can never occur
}

#endif
// PackedArray.cpp

#include "packarray.h"

// auxiliary functions

int log2(unsigned long n) // if n is not a power of 2, log2 rounds down
{
    int res = -1;
    while (n != 0) {
        n >>= 1;
        res++;
    }
    return res;
}
# ifndef __l_SUPPOR_H
# define __l_SUPPOR_H
#include <sstream>

void Error(ostream&);
void Warning(ostream&);
void Fatal(ostream&);
void toler(ostream&, istream&);

# endif
# include <l_suppor.h>
# include <process.h>

void Error(strstream& msg) {
    char c;
    cerr << "ERROR: ";
    while (msg.good()) {
        c = msg.get();
        cerr << c;
    }
    cerr << endl;
}

void Warning(strstream& msg) {
    char c;
    cerr << "WARNING: ";
    while (msg.good()) {
        c = msg.get();
        cerr << c;
    }
    cerr << endl;
}

void Fatal(strstream& msg) {
    char c;
    cerr << "FATAL ERROR: ";
    while (msg.good()) {
        c = msg.get();
        cerr << c;
    }
    cerr << endl;
    cerr << "PROGRAM ABORTED" << endl << endl;
    exit(0);
}

void toLine(strstream& lineStream, istream& in) {
    char line[80];
    char c;
    while (in.good() && in.peek() != '
') {
        in.get(line,80);
        lineStream << line;
    }
    if (in.peek() == '\n') in.get(c);
    // toline
#ifndef L_DEFS_H
#define L_DEFS_H

/**
 * User Definitions
 */

#define MAX_VARIABLES 20

#define PLA_TABLE_SIZE 100
#define PLA_TABLE_DELTA 10
#define STATE_TABLE_SIZE 500
#define STATE_TABLE_DELTA 10

typedef int OutputIndex;
typedef int InputIndex;
typedef unsigned char StateIndex;

typedef char bool;

enum OutputType {Pla_Unknown, f, r, fd, dr, fr, fdr};
enum CircuitType {Unknown, Platype, Sequential, Mooretype, Mealytype};

enum bit {zero, one, dontcare, irrelevant};

enum {FALSE, TRUE};

#define zero '00', one = '01', dontcare = '10', irrelevant = '11'

for better understanding
typedef char bool;

// for better understanding
typedef char bool;

// Definition of bit-values
enum ( FALSE, TRUE );

// Definitions of some types of logic circuits
enum OutputType (Pla_Unknown, f, r, fd, dr, fr, fdr);
enum CircuitType (Unknown, Platype, Sequential, Mooretype, Mealytype);

# endif
strmable.h

File strmable.h Created at 1:01am on Wednesday, August 31, 1994
goossens Page 1 of 1

1 0 // strmable.h
2 0 // Makes objects streamable to output and from input
3 0 // (C) Pawel Konieczny 1994
4 0 // Remarks to: pawel@eb.ele.tue.nl
5 0
6 0 #ifndef STRMABLE_H
7 0 # define __STRMABLE_H
8 0 # include <strstream>
9 0
t10 0 # define __STRMABLE_H
11 0
t12 0 class Streamable
13 4 {
14 4  virtual void inputFrom ( istream& in ) = 0;
15 4  virtual void printOn ( ostream& out ) const = 0;
16 0
17 4  friend istream& operator >> ( istream& in, Streamable& obj )
18 8  {
19 12  obj.inputFrom ( in );
20 12  return in;
21 8  }
22 4  friend ostream& operator << ( ostream& out, Streamable& obj )
23 8  {
24 12  obj.printOn ( out );
25 12  return out;
26 8  }
27 0 ) ; // Streamable
28 0
29 0
30 0 #endif
```cpp
#include <iostream.h>
#include <string.h>
#include <fstream.h>
#include <l_seq.h>
#include <new.h>

void mem_warnC) {
    cerr << "OUT OF MEMORY!
";
    exit(1);
}

int main(int argc, char*argv[]) {
    char outfile[80] = "";
    cout << "This program makes input file in KIS format multiple exclusive

    switch (argc) {
    case 1 : cerr << "Specify input file name.
";
             break;
    case 2 : case 2:
             strncatCoutfile,argv[1],strlenCargv[1])-3);
             strcatCoutfile,"ex1");
             ifstream inCargv[1]);
             ofstream outCoutfile);
             Mealy M;
             in >> M;
             M.multipleExclusive();
             out << M;
             cerr << to " << outfile << endl;
             break;
    default: cerr << "Too many arguments. Only processing first file!

         goto case 2;
         break;
    }
    set_new_handlercO);
    cout << "Press enter...." << endl;
    cin.get();
    return 0;
}
```