Eindhoven University of Technology

MASTER

Implementing CTL model checking within the BSN framework

Vaassen, J.H.M.

Award date:
1996

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section (ES)

Implementing CTL Model Checking within the BSN framework.

by J.H.M. Vaassen

Master Thesis

performed: May 1995 - February 1996
by order of prof.Dr.-Ing. J.A.G. Jess
supervised by ir. G.L.J.M. Janssen
and ir. C.A.J. van Eijk

The Eindhoven University of Technology is not responsible for the contents of training and thesis reports.

Abstract

Design of electronic circuits is still a growing business. Because the complexity of circuits grows, there is need for better verification. There are several methods to verify a circuits actions. One of these methods is symbolic model checking.

Finite State Machines are for example used for controllers. As they grow bigger, the number of states grows very fast. Therefore it is not possible to enumerate states and sets of states. To be able to check large FSMs, Binary Decision Diagrams are used to represent these sets of states. These sets are represented by their characteristic function. This way, the state explosion problem can be avoided.

For model checking, a large number of states can be lethal, because sets or even BDDs representing these sets get to big. To avoid this, a reachability analysis is done first to reduce the number of states. From an initial state, all reachable states are calculated. The set of reachable states is then used to do model checking.

Symbolic Model Checking is used to check behaviour for Finite State Machines. It is a technique that uses a hardware language (BSN used by IBM) to describe the operation of a circuit and a logic (in this case CTL) to describe the desired properties. An implementation was already written by McMillan for his SMV-system, but for the BSN-language a similar program should be implemented.

The goal of this project was to build a model checker for BSN. This goal was not completely reached. The reachability algorithm was implemented, and all CTL-formulas to. Only a parser to process CTL-input files has not yet been implemented. Some tests were done to check if the algorithms worked. The results of those tests were good, the algorithms functioned properly.

Contents

Chapter 1

Introduction

The design of electronic circuits has assumed enormous proportions the last two decades, from the first microprocessors to the VLSI-designs with millions of transistors. When the complexity of design grows, it becomes more difficult to have insight in the exact functioning of a design. And because a production company cannot afford to distribute hundreds of thousands of chips with severe errors, their design must be checked before production.

There are several methods to check circuits. Methods like simulation or testing using patterns are possibilities, but only if these patterns are constructed very smartly. Otherwise they might not cover the full functionality to be checked, or take to much time to be feasible. For sequential circuits, checking is even harder. Because inside the circuit unknown data could be stored, it is even much more difficult to detect all errors. Symbolic Model Checking offers a solution for this problem.

Symbolic model checking uses a description of the circuit as its basis. A transition function is calculated for all inputs, outputs and internal latches. On the other hand we use CTL (Computational Tree Logic) to describe a requested time-dependent behaviour of the circuit. If these two sets of information are put into the model-checker, it checks if the circuit fulfills the desired behaviour. If not, it generates counterexamples, with which a designer should be able to correct the circuit.

1.1 BSN

In this master thesis the process of implementing a model checking algorithm will be described. The hardware to be checked is described in the hardware description language BSN (Boolean Specification Networks), a language developed at IBM T.J. Watson Research Center, New York.
With the BSN language it is possible to describe a digital system hierarchically at various abstraction levels. There are two types of modules, called boxes that are used: The Bbox (behavioural box) and Cbox (connection box). Cboxes define the interconnection between inputs, outputs and lower level boxes. They are called just like subroutines in a program. Bboxes are built just like procedures in a program. They may have inputs, outputs and latches, local variables and tables.

Example Registers are an essential part of FSMs. The next piece of BSN code represents a n-bit register (D-flipflop). Between double square brackets the parameters are passed. The latch can be initialized to any value within 0 to $2^n - 1$ through `Initval`. The only statement `Q:=D` states that the next-state output will become equal to the current-state input.

```
/* n-bit register */
bbox register[[n, Initval]] (input D[0:n-1];
                             latch Q[0:n-1] init Initval)
{
  Q := D;
}
```

The first column of a table defines the variables of the columns, and separates the inputs from the outputs using a `->` mark. The input-variables may be expressions. An example to calculate an minimum for a variable `inf` for some series of inputs called `IN` is implemented as follows:

```
table tab
(IN < MIN ->inf)
{
    0     ->MIN;
    1     ->IN;
}
```

This table can be read as follows: If `IN<MIN = 0` then infinum will be MIN, that is the previous minimum. If `IN<MIN = 1` then the infinum will be `IN`, a new minimum.
Further there are some operators that use a vector in the calculation. These are implemented as follows: `sum := (`*op* `(j:0..n-1) vec[j])`. This action *ops* the bits in the `vec`-vector. The top_instance is the box that is directly visible to the world, for which outputs and inputs can be used in the rest of the program. The operators for BSN with their meaning can be found in appendix C.

Chapter 2

Binary Decision Diagrams

2.1 What are binary decision diagrams?

In binary logic there are several methods to describe a boolean function or value. Some well known methods are the sum of cubes notation, the truth-table, and the Karnaugh-diagram. Another method to represent a boolean function is the Binary Decision Diagram. It was first really implemented in a usable form by Bryant [Bry85], [Bry86]. A Binary Decision Diagram (further referred to as BDD) is a Directed Acyclic Graph (DAG), a graph representation. Each node has a variable attached to it, and has exactly two children. The leaf nodes are always zero or one nodes. The two edges coming from one node are different, they are referred to as the THEN-edge, and the ELSE-edge. This way we can represent a BDD-node as follows:

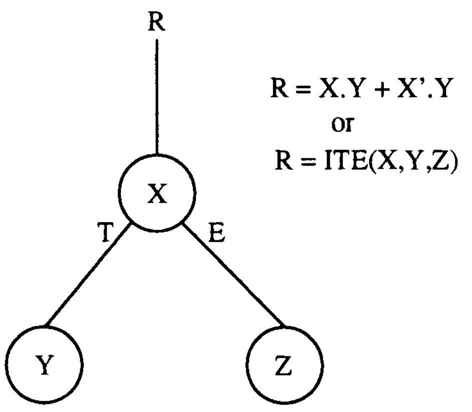

Figure 2.1: A BDD-node

Each node with its children can be represented as an IF-THEN-ELSE (ITE) function. If X is true then Y else Z. The nodes Y and Z themselves again can be BDD-nodes with children, and this way we can build a graph. An example is shown in figure 2.2. The boolean function corresponding to this BDD is:

$$N = y_1(x_1 + \overline{x}_0) + \overline{y}_1\overline{x}_1 x_0$$

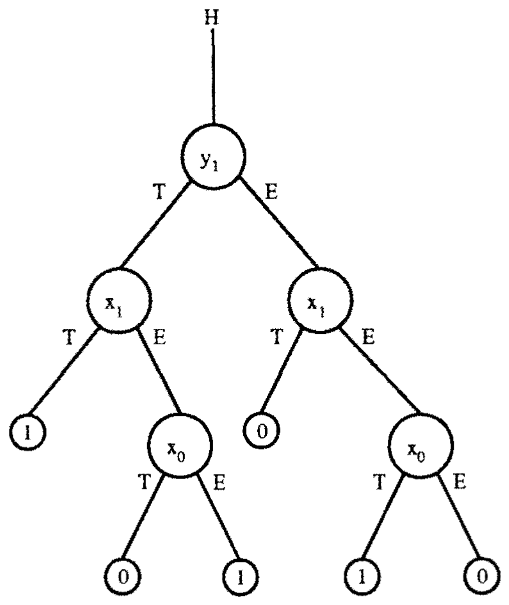

Figure 2.2: A simple BDD

Why do we use BDDs? In computing, one of the known problems is the growth of processors, memory etc. When we want to represent some function as a truth-table, this will cost enormous amounts of memory. It is a very straightforward method, but it (more or less automatically) also means that it is not really efficient. Comparing two truth-tables is really simple, because it is a canonical notation. This means that given some variable-ordering in the table, there is only one way to represent a function. A sum of cubes might be much smarter to denote some binary function, because we select maximal 50 % of the data. But comparing two sums of cubes is much more work. There are various ways to denote two identical functions, so we need to flatten all terms and order them to check if two sums are identical.

BDDs are, given some variable ordering, a canonical representation of a function. So if we want to compare two BDDs, we only search (DFS or BFS doesn't matter) through the graphs, and compare all nodes. This can be done in linear time. The space needed to represent some function in a BDD varies from function to function. Most functions can be represented really compact with BDDs, if a good variable ordering is used. For some functions (notably multiplications) the BDDs grow exponentially in size, no matter what variable ordering is used.

There are some methods to shrink a BDDs size [Met95]. One of these methods is by using so called complemented edges. This means that the value of the BDD to which this edge is pointing is complemented. Graphical it is denoted with a dot on the edge. There are some rules when using negative edges, to make sure that the representation stays canonical. Some configurations of negative edges are not allowed and must be changed to other equivalent configurations. These are the rules to make sure canonicity is guaranteed:

1. The THEN-link of every node must be a non-complemented edge

2. The 1-function is represented by a non complemented edge to the only terminal node (a one-node). The 0-function is a complemented edge to the terminal node.

Of the 8 (= 2^3) possible configurations for a node, only 4 are accepted in BDDs. The other 4 configurations can be directly translated to one of the first 4. In figure 2.3 all possibilities are shown, with the note that the left ones are always chosen.

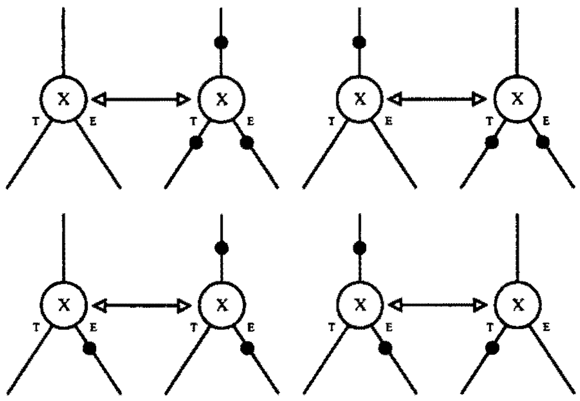

Figure 2.3: Four pairs of equivalent BDDs

Another algorithm to reduce a BDD is the Reduce algorithm [Bry86]. It contains two rules:

1. If the THEN-edge and the ELSE-edge of a node point to the same subgraph, this node may be removed from the graph. The edge(s) which pointed to the deleted node will now point to the subgraph.

2. If two subgraphs are the same, the pointer to one of them will be directed to the other, and the subgraph that has no pointer pointing to it, will be deleted.

If we apply this algorithm on the graph repeatedly until no changes are made, then the graph is canonical and minimal, given the variable ordering.

2.2 Using the BDD-package

The BDD-package is a set of files to be included and linked with your own source code, in which an extensive set of functions is described and implemented in C. The package includes all logical functions (AND, OR, etc), functions for comparing, building, ordering and showing BDDs. In this package, Dynamic Variable Ordering is also implemented. At first, BDDs were build and their variable ordering didn't change during calculations. As a result, the graphs grew very big, and were not manageable anymore. By ordering variables during execution of the package, it is in most cases possible to prevent the BDDs from exploding. Since the ordering is not static anymore, it is called Dynamic Variable Ordering. It is done by some heuristic algorithm, and is almost completely invisible for the user. However there are options in the program to switch DVO on and off, because for some actions, the ordering should not change while they are being executed.

2.3 Expressing a set of states as a BDD

It is worth while, to look a little bit closer at the representation of a set in the BDD-package. It is not necessary to represent each element of the set as a BDD. It is possible to represent a complete set as a function. This is called the characteristic function of a set. It is defined as follows:

▷ **Definition 2.1.** *[Characteristic function of a set]*
Let C be a set and let $A \subseteq C$. The characteristic function *of A is the function $\chi_A : C \rightarrow \mathbb{B}$ defined by:*

$$\chi_A(a) = \begin{cases} 1 & \text{if } a \in A \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

□

Each characteristic function can be represented as a BDD. So it is possible to represent a set of states as a BDD. That will be necessary to efficiently use large sets without enumerating them explicitly.

Example Lets look at a set of transitions. It is not necessary to understand what this means yet, it is used to show the connection between a set of states and its BDD. In equation 2.2 a set of transitions is given. It is defined on four variables (x_0, x_1, y_0, y_1). In figure 2.4 five steps are given for building the BDD for this set of states.

$$H = \{(\overline{x}_0\overline{x}_1\overline{y}_0y_1), (\overline{x}_0x_1y_0y_1), (x_0x_1y_0y_1), (x_0\overline{x}_0\overline{y}_0\overline{x}_1)\} \tag{2.2}$$

The chosen variable ordering is x_0, x_1, y_0, y_1. At each step one variable is chosen to be removed. At the first level this is x_0. The elements in the set that have a negated x_0 variable will be passed through to the *else*-edge, after the x_0 variable has been removed. Those elements for which the variable is not negated are passed through to the *then*-edge after the variable has been removed. This process is repeated recursively until all variables have been removed. The result graph can be simplified using negated edges and by applying the *reduce*-algorithm. BDD 5 in figure 2.4 is reduced, canonical and optimal given the ordering.

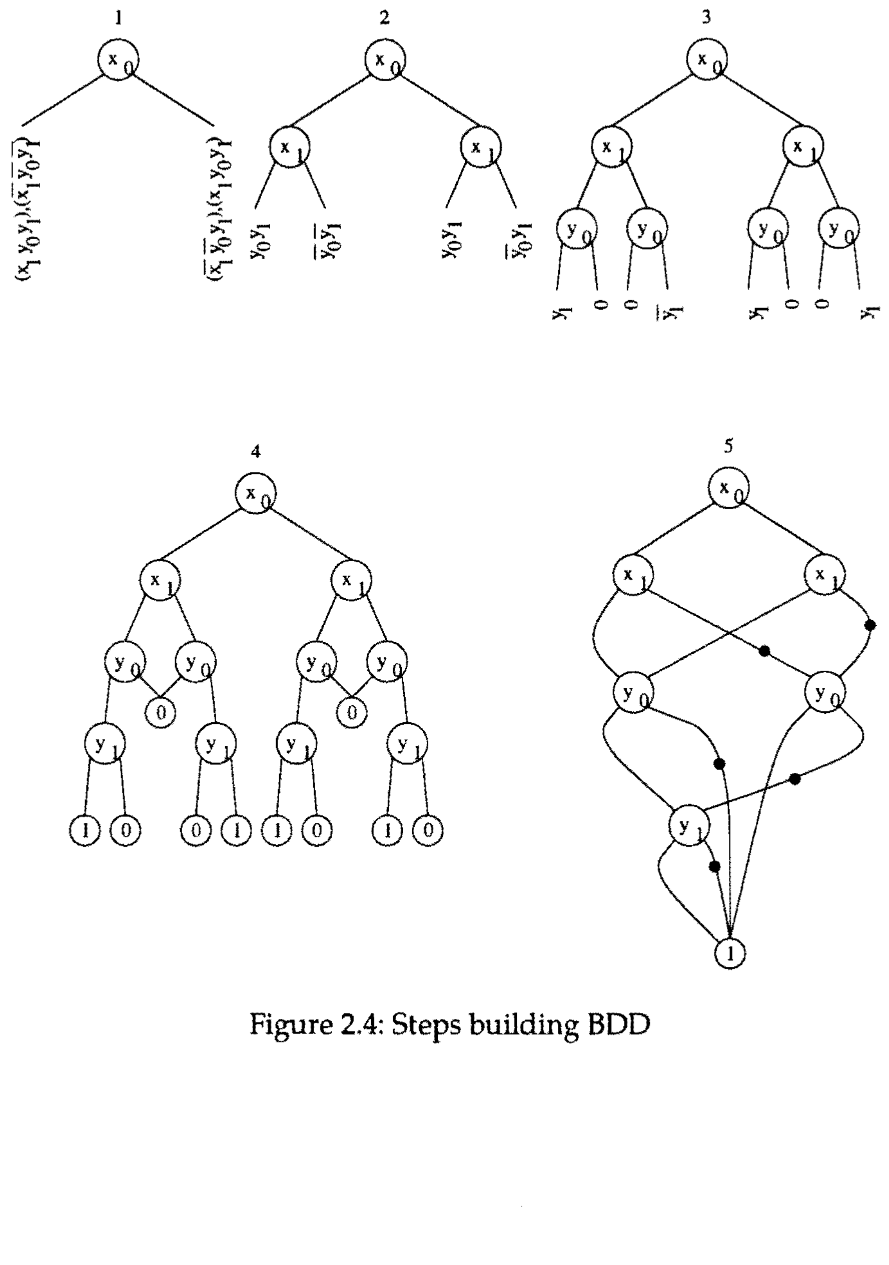

Figure 2.4: Steps building BDD

Chapter 3

Reachability analysis on Finite State Machines

3.1 Introduction

In this chapter, first some definitions will be given about Finite State Machines (to be referred to as FSM), and about a particular model for FSM, the Mealy machine. This model is used for describing synchronous state machines. Further on in this chapter the reachability calculation is explained.

3.2 Finite State Machines

Looking at all electronic circuits, it is possible to make some rough partitions. Analog circuits are one of them for example. One important partition of electronic circuits is that of the FSMs. They represent all digital circuits with memory elements. Since each memory element can only have a discrete number of values (in most cases 2), there is a finite number of states. In the subsequent discussion only circuits with one clock for all latches will be dealt with. The circuit can be (conceptually) split up in two parts. The first part contains all the combinational electronics, with inputs and outputs, while the second part exists only of latches (memory elements). This can be seen in figure 3.1.
Formally we can define a Mealy machine $\mathcal{M}$ as follows:

$$
\begin{aligned}
\mathcal{M} &= (X, Y, S, S_0, \Delta, \Lambda) \qquad (3.1)\\
\text{with} & \\
X &: \text{input alphabet, } X \subseteq \mathbb{B}^m\\
Y &: \text{output alphabet, } Y \subseteq \mathbb{B}^p\\
S &: \text{a finite set of states, } S \subseteq \mathbb{B}^n\\
S_0 &: \text{the set of initial states, } S_0 \subseteq S\\
\Delta &: \text{the next state relation, } \Delta : S \times X \times S\\
\Lambda &: \text{the output function, } \Lambda : S \times X \to Y
\end{aligned}
$$

When we don't look at the combinational logic, the number of possible states only depends

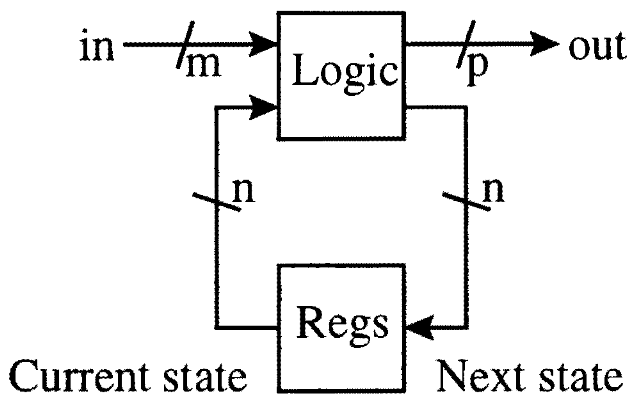

Figure 3.1: Mealy machine

on the number of latches. For the remaining part, the assumption will be made that only binary latches are used. Then, the maximum number of states can be easily calculated as 2^n, with n latches in the circuit. Depending on the logic, the number of states is only a part of this maximum, but for instance for a counter, the set of all possible states, called *state space* can grow very large. Because this number grows very rapidly with the number of latches, taking into account all states when doing model checking gets impossible. This problem is known as the *state explosion problem*. To simplify the model checking, or rather to make it faster or use less memory, only states that can be reached are taken into account. So when an initial state (or set of states) is defined, the set of reachable states can be calculated from it. This process of calculating the set of reachable states is called exploration. The rest of this chapter will treat this process.

▷ **Definition 3.2.** *[Set of reachable states.]*
If we look at a finite state machine

$$\mathcal{M} = (N, S, \mathrm{S}_0, \Delta, \Lambda) \tag{3.2}$$

with symbols like in formula 3.1 then the set of reachable states is the set of states that can be reached in zero or more steps from S_0. *If we denote a transition* $x \to y$ *for* $x, y \in S$, *with* $E(x, y)$, *then the next formula calculates the set of reachable states* y *from state* x:

$$Z(x, y) = E(x, y) \vee \exists_t (Z(x, t) \wedge Z(t, y)) \tag{3.3}$$

The set of reachable states is $\{y | \exists_{x \in \mathrm{S}_0} Z(x, y)\}$.

□

3.3 The initial state

As clearly can be seen from the algorithms to calculate the set of reachable states, two things are necessary. First the transition function N, and second the initial state S_0. There are some options to get these functions. Most of the time it is assumed that all latches are set to zero at initialization of a circuit, through a master reset. This is also the standard option for circuits described in BSN. There is an option to define some other initial state for the latches.

3.4 Transition relation

As written before, to calculate a set of reachable states, is it necessary to have the transition function. This must be derived from the specification of the circuit. The definition for transition function is as follows:

▷ **Definition 3.3.** *[Transition function]*
Let F be the Boolean function vector of next-state functions F: $\mathbb{B}^m \times \mathbb{B}^n \to \mathbb{B}^n$. Let $X = \{ x_1, \ldots, x_m \}$ be the set of input variables, $S = \{s_1, \ldots, s_n\}$ be the set of current state variables and $T = \{t_1, \ldots, t_n\}$ be the set of next-state variables. The characteristic function *of F, denoted by $\mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$, is defined as*

$$N(X, V, V') = \prod_{1 \le i \le n} (t_i \equiv f_i(\underline{x}, \underline{s}))$$

where f_i represents the next-state function for each state bit. N is a functional representation of the following set:

$$\{(\underline{x}, \underline{s}, \underline{t}) \in \mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^n \mid \underline{t} = F(\underline{x}, \underline{s})\}$$

□

In this definition the input is one of the parameters. But to get all reachable states, all possible input combinations are to be included. Because trying them all is impossible, the most simple and correct solution is to quantify the inputs. So all input sequences are included in the process. Just later when doing model checking, the input sequence is needed to trace counterexamples. So before the exploration process is started, the inputs are quantified from the transition function.

Example The automaton in figure 3.2 is a simple FSM. It has no inputs, and the outputs are the state variables themselves.

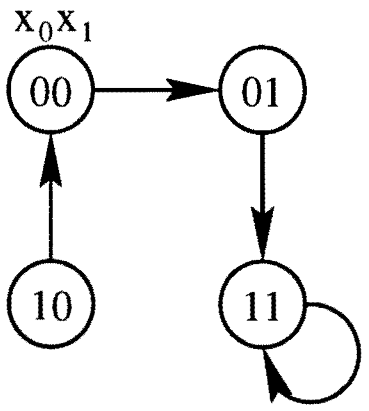

Figure 3.2: Sample FSM

The transition function belonging to this FSM is given in equation 3.4. The BDD for this example can be found in figure 2.4. Each of the four terms stands for a transition in the graph. The x-variables denote current states, and the y-variables denote next-states. It would be possible to simplify the equation because second and third term can be fit together to one term. As can be seen from these two terms, only the current state is different, the next state part of these two terms are equal. This means that these two transitions are pointing to the same next-state. From a graphical representation it is very straightforward to calculate a transition function, but lets look how is it done using only the known equations.

$$N(\underline{x},\underline{y}) = \overline{x}_0\overline{x}_1\overline{y}_0y_1 + \overline{x}_0x_1y_0y_1 + x_0x_1y_0y_1 + x_0\overline{x}_0\overline{y}_0\overline{y}_1 \quad (3.4)$$

3.4.1 Calculation of the transition function.

First thing to know is what information is present after the circuit has been read from file. There are three BDD-vectors constructed from the data in the file. In the first vector the input-variables are given. In the second vector the BDDs that express the output as a function of inputs and latches are given. The third vector contains the BDDs belonging to the latches. For each latch there are two BDDs. The first BDD represents the function of the latch: $f_i(\underline{x},\underline{s})$, and the second BDD represents the variable s_i.
To build a transition function, not only these BDDs are needed, but also a variable identifier for the next state t_i. These variable identifiers will be created, their BDDs will be created, and they will be stored in a separate array. After this has been done, the real calculation can start.

As can be seen in Definition 3.2 the *transition function* is defined as a product over all state-variables, of an *equivalence*-function. This is exactly how it is implemented. First for all state variables the *equiv*-function is executed upon next-state-variable and next-state-bdd. Next the result for all state-variables is *and*-ed into one transition function.

The code for calculating the transition function can be written as is shown in the next procedure.

```
BDD Calculate_Trans_Rel ($F$)
{
  N = 1;
  for (i=0;i<|$F$|;i++) {
    R = bdd_equiv ($t[i]$, $F[i]$);
    N = bdd_and (N, R);
  }
  return N;
}
```

Of course the $t[i]$ and $F[i]$ in the procedure refer to t_i and f_i in the definitions.

Example Again the example in figure 3.2 will be used to clarify the calculation. First we look at the separate functions for current-state to next-state transitions. I will write down the machines transitions as an array in table 3.1. In this table, x_0 and x_1 are the current state (CS)-variables, while y_0 and y_1 represent the next-state (NS)-variables.

Table 3.1: Transitions for sample automaton

CS	NS	
x_0x_1	y_0	y_1
00	0	1
01	1	1
10	0	0
11	1	1

From this table the current-state to next-state transitions for each bit separately can be derived:

$$y_0 \equiv x_1 \tag{3.5}$$
$$y_1 \equiv \overline{x}_0 + x_1 \tag{3.6}$$

Now we will expand these two relations using the equivalence operator. When we use the expansion: $(a \equiv b) \Leftrightarrow (ab + \overline{a}\overline{b})$ we can transform previous equations. From equation 3.5, equation 3.7 is made. From equation 3.6, equation 3.8 is made. By *and*-ing equations 3.7 and 3.8, the transition function is obtained. Because there are no inputs, there is no need to quantify them.

$$y_0 \equiv x_1 = y_0x_1 + \overline{y}_0\overline{x}_1 \tag{3.7}$$
$$y_1 \equiv \overline{x}_0 + x_1 = y_1(\overline{x}_0 + x_1) + \overline{y}_1x_0\overline{x}_1 \tag{3.8}$$
$$N = \overline{x}_0\overline{x}_1\overline{y}_0y_1 + \overline{x}_0x_1y_0y_1 + x_0x_1y_0y_1 + x_0\overline{x}_1\overline{y}_0\overline{y}_1 \tag{3.9}$$

3.4.2 Alternative implementation

Instead of the linear *and*-function, we wanted to try to build a tree-like structure, without changing the order of the variables. To implement this, the calculation procedure was split up in two parts. The first part calculates all temporary results from the equiv's and puts them in an array. The second part recursively retrieves them from this array and puts the transition relation together. The code is given in the next procedures. Commands not necessary for the function (like functions to free memory) are removed from the code to make it more clear.

```
BDD Calculate_Trans_Rel_2 (F)
{
  for (i=0;i<|F|;i++)
    R[i++] = bdd_equiv (t[i], F[i]);
  N = Trans_Calc_Tree (R, i) ;
  return N;
}

BDD Trans_Calc_Tree (BDD equiv_vec[], long len)
```

```
{
  if (len < 2)
    R = equiv_vec[0];
  else
  if (equiv_vec_len == 2)
    R = bdd_and (equiv_vec[0], equiv_vec[1]);
  else {
    long mid_point = len / 2 ;
    R1 = Trans_Calc_Tree (equiv_vec, mid_point);
    R2 = Trans_Calc_Tree (equiv_vec + mid_point,
                 len - mid_point);
    R = bdd_and (R1, R2);
  }
  return R;
}
```

In this way, the calculation was faster and bigger examples could be calculated. In a later stage, we wanted to calculate the transition function again using the linear list and-function. Instead of restoring the old situation by putting the and-function back in the loop, a small function called Trans_Calc_Lin to *and* a vector of BDDs was written, and Trans_Calc_Tree was replaced with it. To my great pleasure the program accepted much bigger examples then before. The only way to explain this, is to say that the results of the calculation were worse because the variable ordering was constantly being disturbed. The *and*-ing builds the transition function but when the *equiv*-function is executed when the transition function is not completely calculated, its variable ordering might be disturbed resulting in a far but optimal solution. It is amazing that such a little change in implementation made such a big difference. The new algorithm is the following.

```
BDD Trans_Calc_Lin (BDD equiv_vec[], long len)
{
  R = bdd_1 ();

  while (len--)
    R = bdd_and (R, equiv_vec[len]);
  return R;
}
```

After testing the different examples with both options, it appeared that it depends on the chosen example, which method works best. We made a command line option of this choice, so the user can chose which method to use.

3.5 The exploration algorithm

From definition 3.3 we can quite easily derive an algorithm to calculate the set of reachable states. As seen in this definition we start with S_0. If this is an empty set we don't need to look any further since the set of reachable states will also be empty. If this is not the case, the set of reachable states can be calculated from S_0 and the transition function. In each step of the algorithm we use a part of the already found partial set, and calculate for all states in this partial set the next states. So each step this partial set will grow until all reachable states are found. If this is the case, the search will stop. There are some alternatives in calculating the reachable state space. I will show two of the most common ones.

3.5.1 Using a Front-set

```
Set Reach (S0, H)
{
  R = S0;
  Front = S0;
  while (Front) {
    Next_states = H (Front);
    Front = Next_states \ R;
    R = R U Next_states;
  }
  return R;
}
```

In this pseudo code H is the set of transitions, and $H(X)$ calculates the set of next states for the set X. What is done in this algorithm can be very easy explained. In each step the next states are calculated from a front-set. This front set contains all next-states that are calculated for the first time. A graphical view can be seen in figure 3.3. After a few steps a set of states already found is created, including S_0. After calling $H(x)$ on the front part of this set, some new states are found (the vertical ellipse). A part of that was already part of the set of states already found. The rest will act as the new front set, which will be used to calculate the next states.

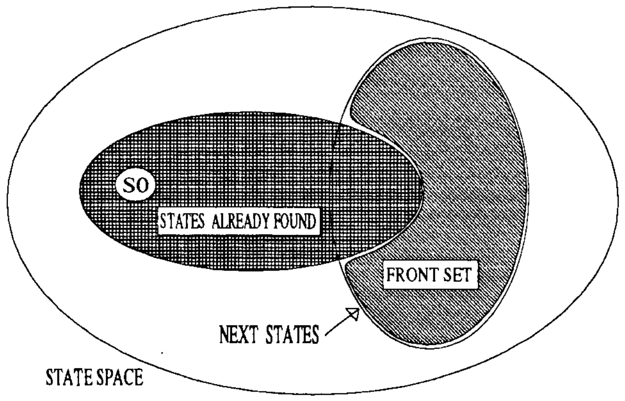

Figure 3.3: Exploration using Front set

This is the minimal set of states to calculate a set of next-states from. Though this looks quite smart it actually is not that smart at all. The size of a BDD is not necessarily related to number of states of the set it represents. So it is not necessary to calculate next states from a small set of states, it just doesn't matter that much. For the join-operator that joins a set of already reached states with the new states, the same thing counts.

There is another point in the algorithm that is not optimal when using BDDs. That is the exclusion-operator that is implemented as an *and not*-operation to exclude all already known states from the front set. The *and*-operation is an expensive BDD-operation, so if it can be avoided, it should be. Keeping in mind these two things the algorithm can be improved. We will see this in the next section.

3.5.2 A more efficient implementation

```
Set Reach ($S_0$, $H$)
{
  $R_k$ = 0;
  $R_{k+1}$ = $S_0$;
  while ($R_k \neq R_{k+1}$) {
    $R_k$ = $R_{k+1}$;
    $R_{k+1}$ = $S_0 \cup H$ ($R_k$);
  }
  return $R_k$;
}
```

As we can see in this implementation, both disadvantages of the previous algorithm are removed. In each step the next states are calculated for all reachable states calculated until now. So set R_k will be monotonous non-shrinking. The only state that is not sure to be found in the next-state calculation is S_0. So in each step S_0 has to be included into the set of next-states. If the newly calculated set is equal to the set it is calculated from, a fixed-point is found. All reachable states have been found. A graphical example of one step can be seen in figure 3.4.

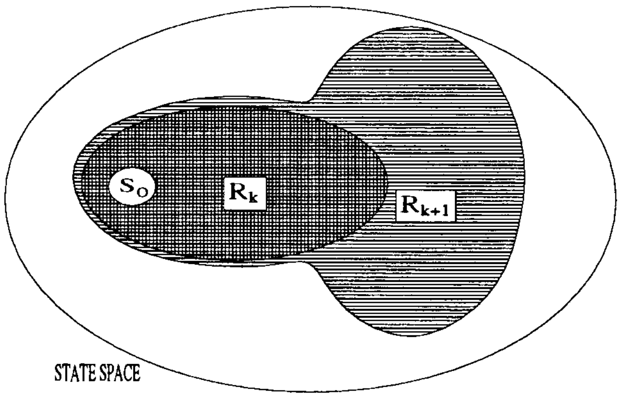

Figure 3.4: Exploration using efficient implementation

3.5.3 A closer look at the algorithm

The transition function can be seen as a set of transitions, as stated in section 2.3. To be able to make a distinction between current-state variables and next-state variables, they have other identifiers. The main action of the reachability calculation, calculating $H(R_k)$, is nothing else but an *and* of N (representing the BDD of the characteristic function of H) and R_k, or in BDD-terms the *and*-function applied to N and R_k. The result is a BDD with current-state and next-state variables in it. The current state variables are not needed anymore, so they will be quantified existentially. The next-state variables will now be exchanged with their corresponding current state variables, that can be used in the next loop. If these steps are carried out consecutively and repeatedly until there is no change in the reachable set graph, the calculation is done.

Example For the given sample automaton in figure 3.2 the transition function is the following:

$$N = \overline{x}_0\overline{x}_1\overline{y}_0y_1 + \overline{x}_0x_1y_0y_1 + x_0x_1y_0y_1 + x_0\overline{x}_1\overline{y}_0\overline{y}_1 \tag{3.10}$$

In this equation x_0 and x_1 are the current state variables, and y_0 and y_1 are the next-state variables. Each term stands for a transition in the graph. If the state 00 is defined as the initial state, then the calculation can done as follows.
First we multiply N and S_0. The result is $\overline{x}_0\overline{x}_1\overline{y}_0y_1$. This is the transition from 00 to 01. Since the start state is not needed it is quantified from the result. This leaves $\overline{y}_0y_1$. But the result is wanted as an expression of current-state variables, so the next-state variables y_n are simply substituted by current-state variables x_n. After taking the union with S_0 the first step in the reachable calculation is done. The result is a set with characteristic function $\overline{x}_0$. In this example this step has to be done three more times to calculate the complete set of reachable states. The temporary values for R_k, the set of next-states (NS) and R_{k+1} can be seen in table 3.2. The last step doesn't add any states to the set, so it functions as a stop-criterion. The set will be $x_1 + \overline{x}_0$, which is equal to the states 00 and 01 and 11.

Table 3.2: Exploration of sample FSM

R_k	NS	R_{k+1}
{(00)}	{(01)}	{(00),(01)}
{(00),(01)}	{(01),(11)}	{(00),(01),(11)}
{(00),(01),(11)}	{(01),(11)}	{(00),(01),(11)}

3.6 Number of reachable states

For all examples tested I will try to give a calculation how to get the number of reachable states. This was used to check if the algorithm works properly. At some point this seemed so and when comparing calculated number of states with theoretical values, the difference between calculated values and theoretical values grew to big. This was result of an error in the procedure for calculation of the number of minterms for a BDD. This has been corrected in the BDD-package.

3.6.1 The arbiter

The number of reachable states for the arbiter can be calculated very easily. A brief explanation and some schematics for the arbiter are given in appendix A. There are two latches in each cell of the arbiter. The number of cells will be n. So the maximum of reachable states is $(2 \cdot 2)^n$. The first latch in each cell contains the wait-value, which checks if a request is holding more than one time unit and the token has passed. Hence each cell can be zero or one at any time independently. So these latches can have maximal 2^n states. The other latches are used to pass a token. This token gives a cell priority to cells with higher fixed priority when its request is persistent. For the arbiter to work correct, there must only be one token. This means that only one cell has the token, and all the others cells don't. So here exactly n states are possible. Since both latches hold their values independently, the maximum number of states is $n \cdot 2^n$.

3.6.2 The Minmax signal processor

The Minmax signal processor is a simple cell which calculates for a defined number of bits, the maximum and minimum for a series of numbers fed to it. It also outputs the last input value. The specific definition for Minmax is given in appendix B. As can be seen in this file, there are three latch arrays, each having n bits.
When calculating the number of states, there is a number of possibilities to take as starting-point. Let us take the *last*-register to calculate the result from. The *last* register holds the last input value, and *min* and *max* registers will contain the minimum and maximum value in the series. There is one exception to this last rule, when reset is true, the *max*-register will be filled with zeros, and the *min*-register will be filled with one's. In this specific case, *last* can hold any value, so that accounts for 2^n states. In all other cases there are restrictions on registers. For all inputs, the *min* register must contain a value smaller then the *last* register, and the value contained in the *max* register must be larger then the *last* value. So for a certain input value x, the *min*-value is restricted to $x+1$ values. The *max*-value is restricted to $2^n - x$ values, all values larger then or equal to x. So the total number of states is given in equation 3.11.

$$\#states = 2^n + \sum_{i=0}^{2^n-1} (i+1) \cdot (2^n - i) = 2^n + \sum_{i=1}^{2^n} i \cdot (2^n - i + 1) \tag{3.11}$$

To get a notion of the growth of this number of states, in table 3.3 some values are shown.

3.7 Results

Since reachability is a first important step on the way to symbolic model checking it seemed obvious to do some testing on sample circuits. For the tested circuits the number of states were known, and there were some results from an earlier built model checker by McMillan. These will be compared with the results we got when running these examples. Also, when testing the programs, the BDD for the reachability function was looked at. With a little

Table 3.3: Number of reachable states

#bits / #cells	Minmax	Arbiter
1	6	2
10	179482624	10240
20	$1.922 \cdot 10^{17}$	20971520
30	$2.063 \cdot 10^{26}$	$3.221 \cdot 10^{10}$
40	$2.215 \cdot 10^{35}$	$4.398 \cdot 10^{13}$
50	$2.397 \cdot 10^{44}$	$5.629 \cdot 10^{16}$
60	$2.554 \cdot 10^{53}$	$6.917 \cdot 10^{19}$
70	$2.743 \cdot 10^{62}$	$8.264 \cdot 10^{22}$
80	$2.945 \cdot 10^{71}$	$9.671 \cdot 10^{25}$

analysis, it is possible to compare the BDD with the expected function and conclude that they are the same.

3.7.1 The arbiter

For the arbiter the tests as given in the previous section are done. The use of time grows steadily, but at some point, where the maximum allowable memory of the computer is reached, the time grows exponentially (±65 Mb). This can be seen in figure 3.5. In figure 3.6 the same graph is given, without a logarithmic y-axis. This graph really shows how fast the used time grows. In figure 3.7 the peak memory use is shown. In this graph, the y-scale is again logarithmic, to show the steps in the area from 0 to 70 cells. If displayed without log-axis, it would look like the time-use in figure 3.6.
To support the proposition that the efficient implementation is indeed more efficient then the implementation using a front set, a reachability analysis has been done, using the different algorithms. The data are given in table 3.4.

3.7.2 The minmax signal processor

In figure 3.8 the time use for reachability calculation of the minmax signal processor is shown. It shows three lines, first the time used to calculate the sequential model, this includes getting the inputs, calculating the initial state, calculating the domain of all latch-bits, and then calculating the transition function. The second line in the graph shows the time for exploration, and because (no matter how many bits) the search is always done in four steps, this time is regularly very low, compared to the time to calculate the sequential model. The third line is an equation that approximates the total calculation time. The equation is: Time (ms) = $3700 + 9.5 \cdot b^{2.2}$ (b is #bits). So in this area the time use is still polynomial with a reasonable order. But as the program is using BDDs and heuristics, nothing can be said about the cases not calculated. The enormous peak in the graph proves that there is no guarantee of time/memory use whatever. The memory use is given in figure 3.9. The peak seen in the time-use graph can also be seen here, but is much smaller compared to the peak in the time graph. This graph can also be read as a graph for the maximum number of BDD-nodes, as there is a strong connection between used memory

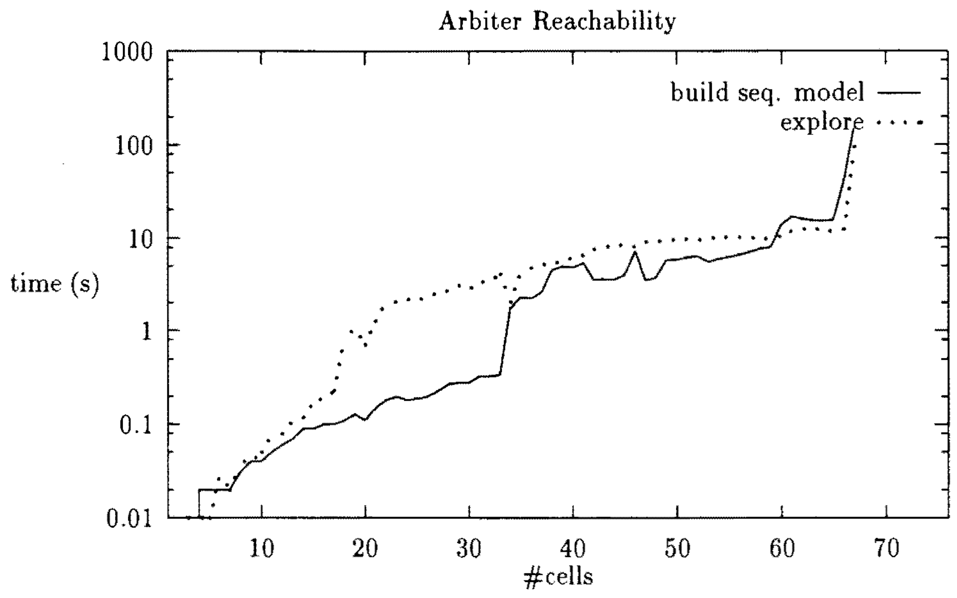

Figure 3.5: Reachability time use

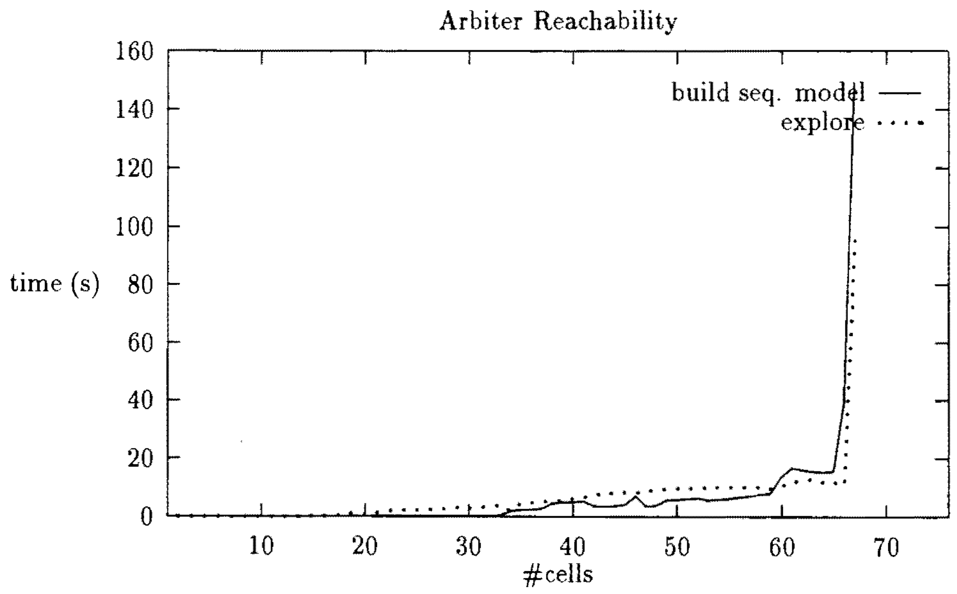

Figure 3.6: Reachability time use

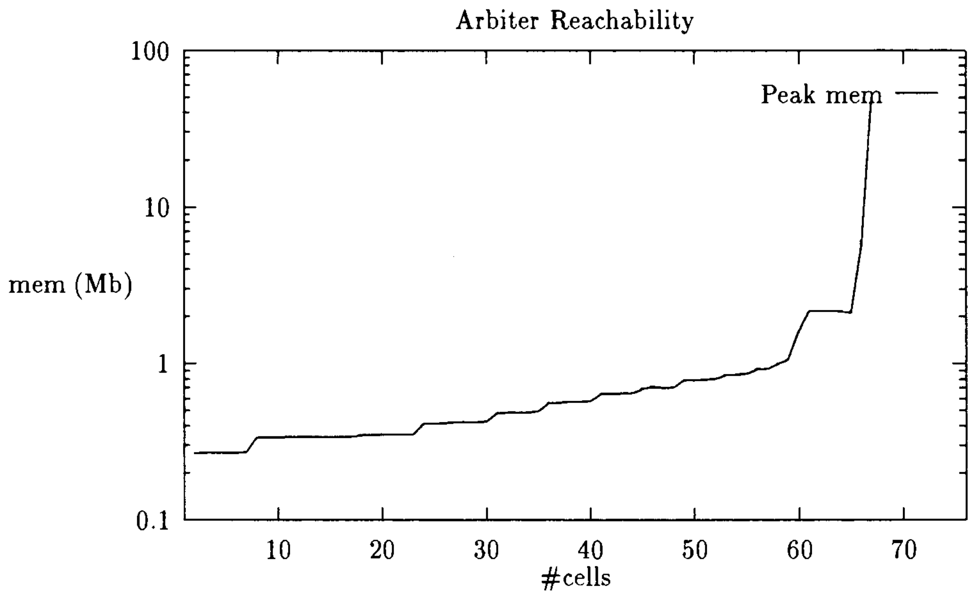

Figure 3.7: Reachability peak memory

Table 3.4: Implementation differences

#cells	Memory-use (kb)		Time-use (s)	
	Front	Eff.	Front	Eff.
2	267	267	0.09	0.16
10	337	337	0.40	0.33
20	417	351	4.69	1.60
30	503	425	11.61	5.50
40	650	580	35.69	16.15
50	1004	790	67.68	29.61
60	2964	1635	211.61	46.04
61	4116	2191	264.66	51.43
62	-	2183	-	53.37
67	-	48898	-	282.49

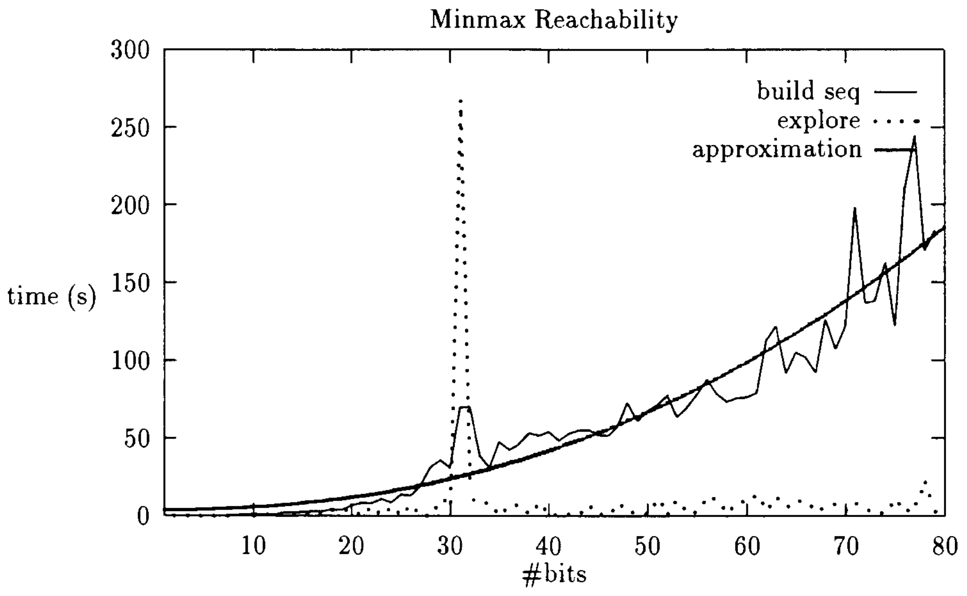

Figure 3.8: Reachability time use

and number of nodes. Also a relation can be seen between the time use and the number of nodes. This is plot in figure 3.10. The little diamonds are the actual number of nodes used, the dotted line is a moving average over 7 points. A linear approximation results in the next function:

$$\#\text{nodes} \cong 981 \times \text{time(s)}$$

So about 1000 nodes a second are calculated / used.

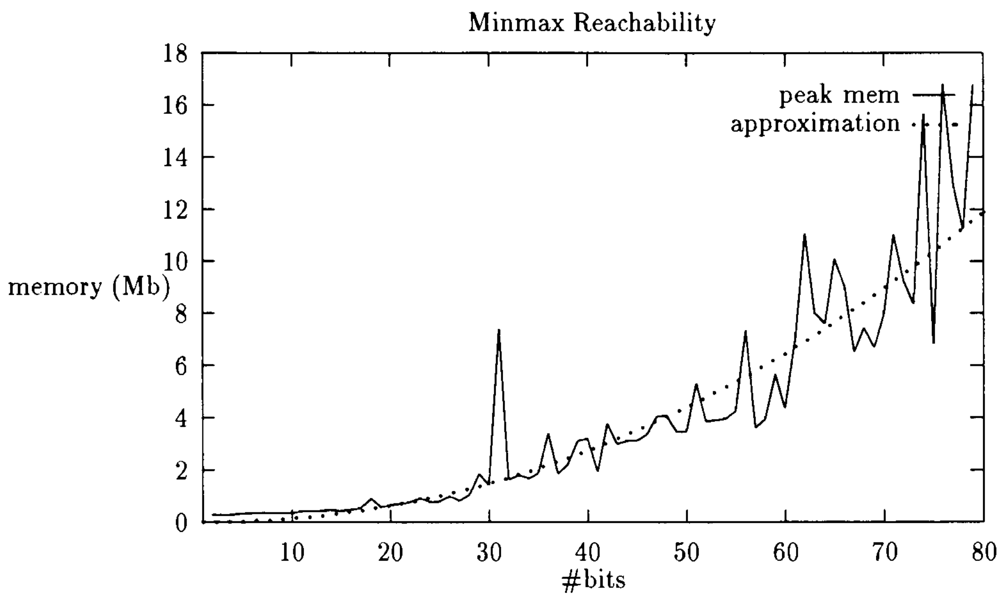

Figure 3.9: Reachability memory use

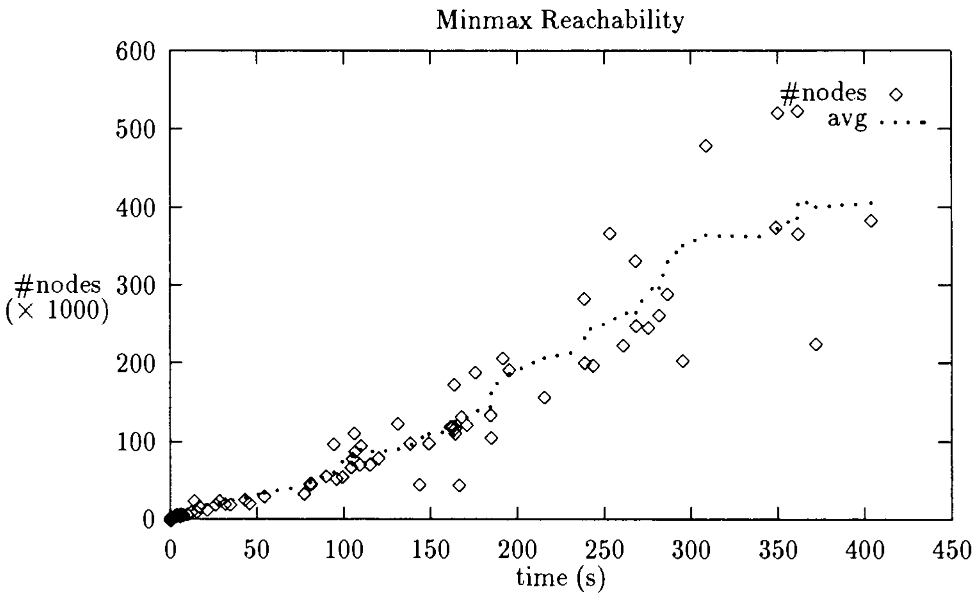

Figure 3.10: Reachability memory versus time

Chapter 4

Symbolic Model Checking

There is a number of techniques to verify the operation of an electronic circuit. Simulation is, looking at the complexity of it, a simple technique. But for larger circuits, the number of possible inputs or input sequences grows that big, that simulation cannot be done in acceptable time, without missing to much errors. It is a fact that simulation, when not done exhaustively, might not find all errors in a circuit. So there is a need for another possibility for checking electronic circuits.
Symbolic Model Checking is a technique to verify properties of an electronic circuit. First a model is needed to describe the circuit. This can be a behavioral description, or a hardware description on gate-level. On the other hand a language is necessary to describe properties one wants to verify. This chapter will discuss a language to describe the latter one. For sequential circuits a time-dependent description is needed. So it is necessary to include some time-conception in this model. The class of time-dependent models for verification of electronic circuits is called temporal logics. One of the temporal logics is CTL.

4.1 Computation Tree Logic

CTL is one of the most common model checking languages. It knows some operators to describe time-dependent behavior. They look alike the PTL operators, with one difference. Because for each state multiple successors are allowed, there is an adaption made. Each of the PTL operators is extended with a path descriptor. This says that some predicate must hold for at least one path, or that something must hold for all paths. The set of operators is summarized in the table 4.1. Of course the simple boolean operators like *and*, *or*, *not* and combinations of these three are also allowed. The predicates p and q used in table 4.1 must of course have boolean values. The result of a CTL formula is always a boolean value. Nesting of operators is also allowed.
The semantics of a CTL formula is defined with respect to a labeled directed graph, called a CTL-structure. Formally a CTL structure is a quadruple $M = (S, R, A, L)$, where

1. S is a finite set of states
2. R is a binary relation on S. For $s, t \in S$ sRt means that t is a immediate successor of s.
3. A is a set of atomic formulas.

Table 4.1: CTL operators

Operator	description
AG p	for all paths p holds in every state
EG p	for at least one path p holds in every state
AF p	for all paths, eventually p holds
EF p	for at least one path, eventually p holds
AX p	for all paths, p holds in the next state
EX p	for at least one path, p holds in the next state
p AU q	for all paths p holds until q holds
p EU q	for at least one path p holds until q holds

4. $L : A \rightarrow 2^S$ is a function that maps each atomic formula into the set of states in which the formula holds.

Let now s be a state in the CTL structure $M = (S, R, A, L)$. With M and s we associate a computation tree, rooted as s and with an edge from node t to u or tRu. Given a CTL formula f, we write ($s \models f$ for M) to state that the formula f holds in the computation tree derived from M and rooted at s.

The next semantics for CTL will clarify this:

$$\begin{array}{rcl}
s \models f & \Longleftrightarrow & s \text{ in } L(f), f \text{ an atomic formula} \\
s \models \neg f & \Longleftrightarrow & s \not\models f \\
s \models f \wedge g & \Longleftrightarrow & s \models f \text{ and } s \models g \\
s \models \mathrm{AX}f & \Longleftrightarrow & \forall_{paths}(s_0, s_1, \ldots), s_1 \models f \\
s \models \mathrm{EX}f & \Longleftrightarrow & \exists_{path}(s_0, s_1, \ldots), s_1 \models f \\
s \models \mathrm{AG}f & \Longleftrightarrow & \forall_{paths}\forall_{n\geq 0}(s_0, s_1, \ldots), s_n \models f \\
s \models \mathrm{EG}f & \Longleftrightarrow & \exists_{path}\forall_{n\geq 0}(s_0, s_1, \ldots), s_n \models f \\
s \models \mathrm{AF}f & \Longleftrightarrow & \forall_{paths}\exists_{n\geq 0}(s_0, s_1, \ldots), s_n \models f \\
s \models \mathrm{EF}f & \Longleftrightarrow & \exists_{path}\exists_{n\geq 0}(s_0, s_1, \ldots), s_n \models f \\
s \models f\mathrm{AU}g & \Longleftrightarrow & \forall_{paths}\ (\exists_{n>0}(s_0, s_1, \ldots), s_n \models g \text{ and } \forall_{0\leq j<n} s_j \models f) \\
s \models f\mathrm{EU}g & \Longleftrightarrow & \exists_{path}(\exists_{n>0}(s_0, s_1, \ldots), s_n \models g \text{ and } \forall_{0\leq j<n} s_j \models f)
\end{array}$$

There is a clear correspondence between PTL and CTL. The CTL operator is in fact a PTL-operator with prefix A or E. And the choices for A as prefix meaning *for all*,$\forall$ and E meaning *exists*, $\exists$ are not coincidental at all.

Example Let us look at a simple traffic-light. There are two roads, one running from north (N) to south (S), and one from east (E) to west (W). At the intersection there are four lights, one for each direction. The lights for traffic coming from the north are coupled to those coming from the south. Lights for east and west are also identical. The input variables are *traffic*$_{\text{dir}}$ that indicate traffic coming from the given direction. The output variables are *Color*$_{\text{dirs}}$ indicating the color of the light for the given directions. A safety CTL-statement could be:

$$\text{AG } \neg(\text{Green}_{\text{EW}} \wedge \text{Green}_{\text{NS}}) \tag{4.1}$$

This CTL-formula states that at no time all lights may be green. To state that if traffic arrives from some direction, it will eventually get a green light, the next CTL statement could be constructed:

$$\text{AG AF } ((\text{traffic}_{\text{E}} \vee \text{traffic}_{\text{W}}) \Rightarrow \text{Green}_{\text{EW}}) \wedge \tag{4.2}$$
$$\text{AG AF } ((\text{traffic}_{\text{N}} \vee \text{traffic}_{\text{S}}) \Rightarrow \text{Green}_{\text{NS}}) \tag{4.3}$$

It is possible to express some CTL operators in others. This means that only a few operators have to be implemented.
Table 4.2 gives the conversions from CTL-formulas to other CTL-formulas. When some loop references are excluded, a small set of formulas can be chosen to be implemented. The rest then can be expressed as a function of these expressions.

Table 4.2: Some conversion for CTL operators

formula	abbreviates
$p \vee q$	$\neg(\neg p \wedge \neg q)$
AX p	$\neg$ EX $\neg p$
AF p	$\neg$ EG $\neg p$
EF p	$True$ EU p
AG p	$\neg(True$ EU $\neg p)$
p AU q	$\neg((\neg q$ EU $\neg(p \vee q)) \vee$ EG $\neg q)$

So this leaves just a few formulas to implement. Those are *EX* p, *EG* p and p *EU* q.

4.2 Implementation

Next the implementations for the last three CTL-operators will be given. These calculations were shown by Emerson and Clarke [ECl81] to be characterizable as fixed point calculations. Let's first introduce the functional representation: $\tau = \lambda y.f$ is the function f, with each occurrence of y in it replaced by the parameter of τ. For example if $\tau = \lambda y.(x \vee y)$, then $\tau(false) = (x \vee false) = x$.
A *fixed point* of a functional τ is any p such that $\tau(p) = p$. For example a fixed point of $\tau = \lambda y.(x \vee y)$, is $(x \vee y)$, because $\tau(x \vee y) = x \vee (x \vee y) = x \vee y$. For a monotonic functional two special fixed-points can be defined. The *greatest fixed point* is the union of all fixed points, and the *least fixed point* is the intersection of all fixed points. How can this be of any use for implementing CTL-formulas. First let's look at the operator EG p. EG p is logically

equivalent to ($p\wedge$ EX EG p). now we can see EG p as a greatest fixed point functional $\tau = \lambda y.p\wedge \mathrm{EX}y$.

Because we are checking only finite automata, the fixed point computation can be characterized as the limit of a series. This can be obtained by iterating the corresponding functional. These are given in equations 4.4 and 4.5.

$$\mathrm{EG}p = \cap_i(\lambda y.(p \wedge \mathrm{EX}y))^i(true) \quad (4.4)$$
$$p\mathrm{EU}q = \cup_i(\lambda y.(p \vee (q \wedge \mathrm{EX}y))^i(false) \quad (4.5)$$

The y in formula 4.4 is repesented in the program as Y_k. The size of the set represented by Y_k is monotonically decreasing for growing k until for some k,$Y_k = Y_{k+1}$. So instead of calculating the union of all Y_k, it is more efficient to calculate Y_k until $Y_k = Y_{k+1}$. The implementation is given in the next paragraph. The first assignment of Y_k can be changed to decrease the number of steps by one. The original algorithm starts with the true-function (bdd_1). By filling in p, the number of iterations is decreased with 1. This can be done because EX 1 = 1, so $Y_1 = p$.

```
BDD EG (BDD p)
{
  Yk = bdd_1 ();

  do {
    Yk+1 = bdd_and (p, EX (Yk));
    if (BDD_EQUAL_P (Yk, Yk+1))
      break;
    Yk = Yk+1;
  } forever;
  return Yk;
}
```

Example Again using the traffic-light example the next CTL-formula will be checked: *EG (Red*$_{\mathrm{EW}}$*)*. So the algorithm will select all cells for which at least one path Red_{EW} is always true. First in figure 4.1 the finite state machine will be given. In each node the color of the light for north-south and east-west are given. The values at the edges are the input values. EW Represents traffic from east or west, NS represents traffic from north or south.

In the original algorithm the first step would be to select all states. In the implemented version the first step is to select all states that satisfy Red$_{\mathrm{EW}}$. This is shown in figure 4.2.

The second iteration selects all states which satisfy $Red_{\mathrm{EW}} \wedge \mathrm{EX}Red_{\mathrm{EW}}$. This is shown in figure 4.3. The third iteration shows all states which satisfy $Red_{\mathrm{EW}} \wedge EX(Red_{\mathrm{EW}} \wedge \mathrm{EX}Red_{\mathrm{EW}})$. This is the same set as after iteration 2, and it can also be seen in figure 4.3.

For the EU-algorithm a similar piece of code is written in the next paragraph.

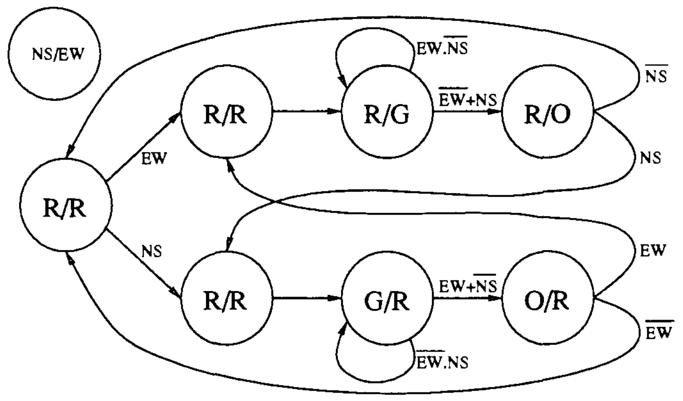

Figure 4.1: Traffic-light FSM

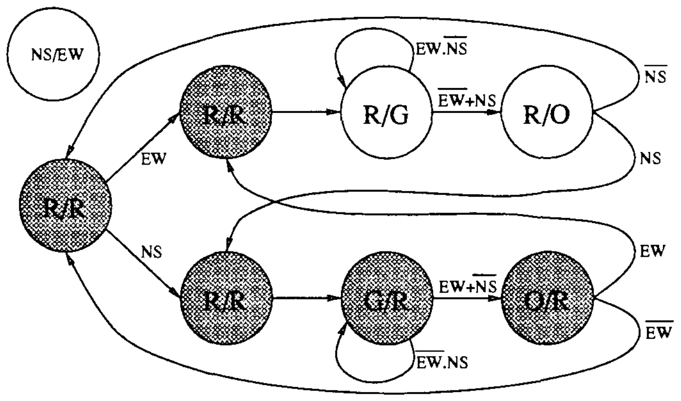

Figure 4.2: Traffic-light after one step

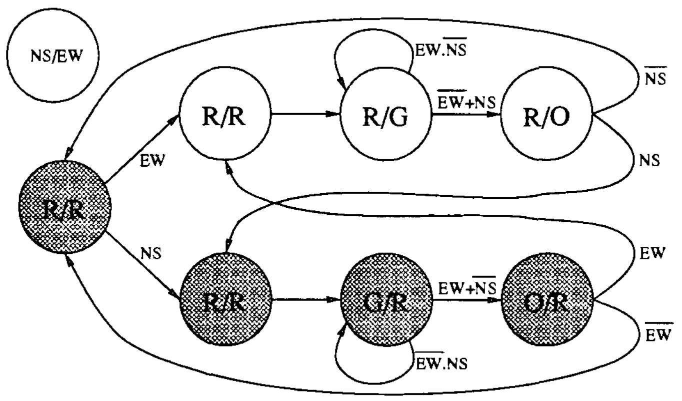

Figure 4.3: Traffic-light after two or more steps

```
BDD EU (BDD p, BDD q)
{
  Y_k = bdd_0 ();

  do {
    s1 = bdd_and (p, EX (Y_k));
    Y_{k+1} = bdd_or (q, s1);

    if (BDD_EQUAL_P (Y_k, Y_{k+1}))
      break;
    Y_k = Y_{k+1};
  } forever;
  return Y_k;
}
```

As can be seen, all implementations for formulas can be expressed as fixed-point calculations using the EX operator.This operator has yet to be implemented.
First lets look what the use of this operator means. If the following expression needs checking: EX p, what do we want to get as a result? Obviously the problem is to find a set of states, or a state, for which one of its successors satisfies p. So if we have all states that satisfy p, the problem would be reduced to finding all predecessors of these states. So instead of using the *image* like in the reachability analysis, the *pre-image* is used. The set of states that satisfy p is represented by p.

The second part of the problem is to get the pre-image function. The transition relation has already been used in the reachability calculation. So it is known if there is a transition from a state x to a state y. In the inverse transition relation there must then be a transition from y to x. This can be realized by exchanging all variables. So each current-state variable x is exchanged with its next-state variable y.

Example In the traffic-light example there is a transition from R/R to R/G (see figure 4.1). In the pre-image there will be a transition from R/G to R/R. If a FSM would be built for a pre-image, it would be the same as the FSM corresponding to the image, but with all edges reverted.

The rest is just the *and*-ing of the two representing functions. The current state variables can be quantified from the result of the *and*-operation. What remains is not yet the result as desired because it is still expressed in next-state (or should I say previous-state) variables. So substituting current-state variables for the others completes the process.

The model checking algorithm can be implemented without using the reachability calculation. That seems to save some time, but if we don't restrict the set of states to the set of reachable states the model checking algorithm only accepts little circuits. For larger circuits the model checker will run out of memory. This can be clearly seen in table 4.2. The results in the second column are retrieved from [Mcm93]. It is not known why there are no results beyond 12 cells. The data from our own program are put in the third and fourth column. The data from the third column is retrieved from the program without using reachability. McMillan's program was implemented in C and executed on a Sun3, our program was also implemented in C but executed on a HP 9000/735 on 99 MHz.

Table 4.3: Run times for model checking

#cells	McMillan	bsn2mc	bsn2mc
		$\overline{R}$	R
	time (s)	time (s)	time (s)
3	0.6	1.1	0.16
5	1.5	1.2	0.22
7	3.5	2.9	0.36
9	5.4	3.4	0.87
10	6.9	5.1	1.39
12	11.0	8.6	2.64
15	-	25.6	5.53
20	-	166.6	15.76
30	-	-	51.42
40	-	-	98.24
50	-	-	261.33
60	-	-	437.49

4.3 Verification with model checking

Verification is another way to check if a circuit works as wanted. Also it is possible to check if two circuits have the same functionality. For combinational circuits this is very easy. First for all outputs of both circuits the functionality of each output is written down, as a function of the inputs. Then two outputs that should have the same value can be compared. If the circuits outputs are represented as BDDs, it's just a case of comparing all BDDs, which can be done in constant time of the number of BDDs. For very large circuits it is possible to use cutpoints, a technique where the circuit is split up in smaller blocks, that should be compared with each other.
For sequential circuits, this technique cannot be used just like that. The BDDs for the combinational part of the circuits can be calculated, but if the latches contain different values, the previously mentioned technique is worthless. But having implemented some form of model checking, the following method can be used.

First the assumption is made, that both circuits have the same inputs and outputs, otherwise they can't be compared at all. Define a new circuit, containing the circuits that are to be compared. Connect the inputs in such a way that each input of the first circuit is connected with its corresponding input of the second circuit. These inputs will be defined to be the new inputs for the new circuit. For all outputs, a pairwise equivalence-function is used, and so each output is compared with its equivalent output on the other circuit.

Connect all outputs of these equivalence gates with the input of a multi-input and-gate. This will be the only output (let us call it *Identical*) of the new circuit. Figure 4.4 shows the new circuit.

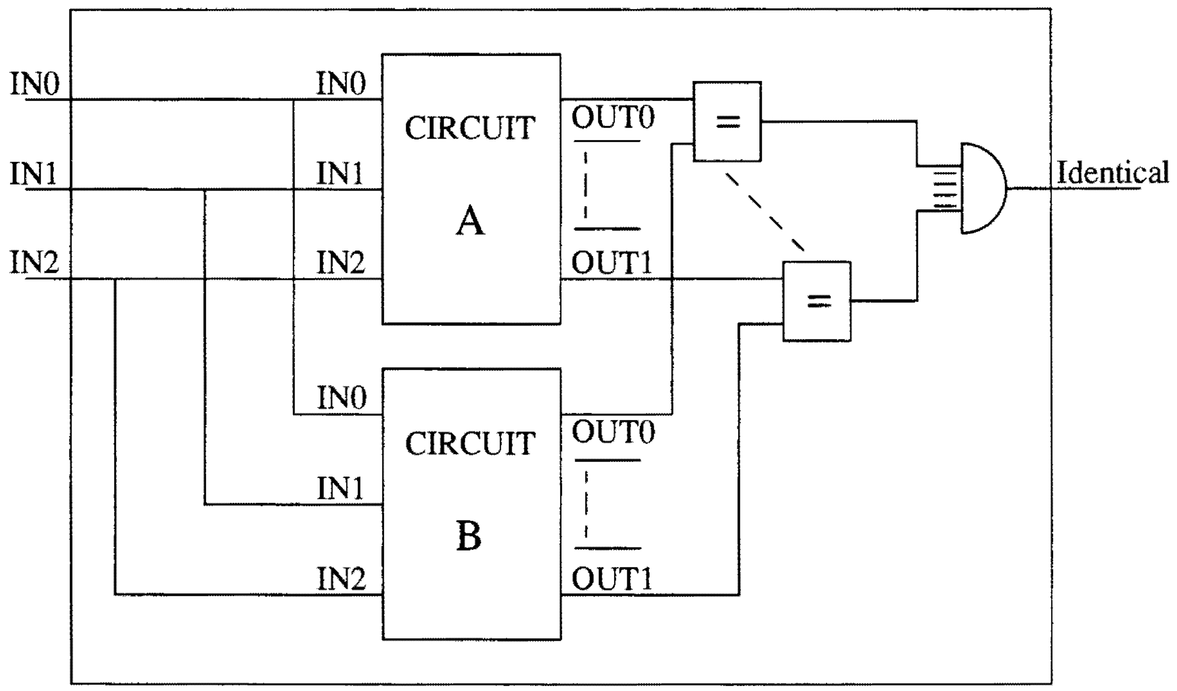

Figure 4.4: Verification circuit

Then a CTL formula can be stated: $s_0 \Rightarrow$ AG *Identical*. This formula only holds if for all inputs, all outputs of both circuits are identical. The CTL-formula can be checked by checking if (AG *Identical*) covers s_0 . If this is the case, than s_0 is an initial state for which both circuits have identical behaviour. This means also that there might be other states for which the machines act identically. Because symbolic model checking is used, it is not necessary to try all input combinations, or to find a correspondence between latches in both sub-circuits. Actually the programs are that transparent, that it is possible to compare two descriptions on completely different logical levels (for example compare a net-list with a behavioural description). The disadvantage of this technique is that the size of the BDD for the new circuit grows very large. So it is not yet possible to do verification on very large circuits.

4.4 Results

The program is capable of checking CTL-formulas for outputs of a circuit. Sometimes it is therefore necessary to create extra outputs to check latches without direct outputs. It also might be necessary to include a little logic to calculate temporary results, because the parser has not been implemented.
The arbiter is a circuit for which results were known, and some comparison with previous

checking can be done. McMillan stated three desired properties for this circuit. They are:

1. No two acknowledge outputs are asserted simultaneously
2. Every persistent request is eventually acknowledged
3. Acknowledge is not asserted without request

Another property that can be checked with respect to efficiency is, that if there is at least one request, there must be an acknowledge. All properties can be translated to CTL. This results in the next formulas:

1. $\forall_{i \neq j}$ AG $\neg(\text{ack}_i \wedge \text{ack}_j)$
2. $\forall_i$AG AF $(\text{req}_i \Rightarrow \text{ack}_i)$
3. $\forall_i$AG $(\text{ack}_i \Rightarrow \text{req}_i)$
4. AG $(\vee_i \text{ req}_i \Rightarrow \vee_j \text{ ack}_j)$

These formulas were implemented in two parts: one part in the BSN circuit-description file to create extra outputs, the CTL-part in the program for really doing the model checking. The following functions declare the subsequent checks for formulas:

```
/* No two acknowledge outputs are asserted simultaneously */
bbox proposition1[[n]] (input ack[0:n-1];output only1on)
{ only1on:= (+(i:0..n-1) ack[i]) <= 1; }
```

Proposition 1 claims that: $\sum_{i=0..n-1} ack[i] <= 1$, the number of acknowledges is zero or one. That is the same as to declare that no two acknowledges can be set at the same time. The part that is put in the program code is the following:

```
  r = AG (m, the_outputs[0]);
```

This claims that for model m, output[0] must satisfy the AG operator. What is really checked can be seen in the next function: AG $(\sum_{i=1..n} ack[i] <= 1)$. This is equivalent to the desired expression.
The BDD that can be drawn shows a functions, that demands that only token register is set, i.e. there is only one token. This BDD can be seen in figure 4.5. When the function that is expected is known and not too complex, it can be deduced from the graphical representation.

The second formula is more difficult to check. In this case it is not possible to *sum* or *and* the results in the circuit, so for each cell there must be a separate output.

```
/* A persistent request will eventually be acknowledged */
bbox prop2[[n]] (input req[0:n-1], ack[0:n-1]; output ctl[0:n-1])
{ (i:0..n-1) ctl[i]:= ack[i] | ^ req[i]; }
```

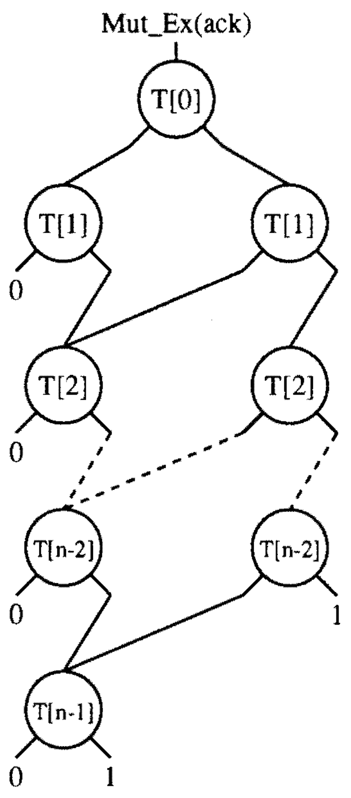

Figure 4.5: BDD for property 1 (maximum 1 ack)

In the program for each cell first AG AF is calculated, and then an *and* is done over all outputs in array `ctlout2`. The `r` that is the return value of Bdd_Trans_Calc_Lin is the wanted result. This BDD is 1, so this property is always satisfied.

```
for (test_count = nr_test_bits; test_count;test_count--)
  ctlout2[test_count-1] = AG (m, AF (m, the_outputs[test_count]));
r = Bdd_Trans_Calc_Lin (ctlout2, nr_test_bits);
```

The third and fourth property are constructed in a similar way. They both use only one output port.
In figure 4.6 both time used for reachability calculation and for model checking (including reachability calculation) are shown. As can be seen, at first the model checking doesn't cost much more time, but if the number of cells grows, the time needed earlier grows to larger values. This same effect can be seen in figure 4.7 where the memory used for model checking in early stages is only a little bigger, but for large number of cells, the number of nodes suddenly grows much bigger. For 54 and 58 bits the model checking algorithm needs more memory than allowed, so the program exits with an out-of-memory message.

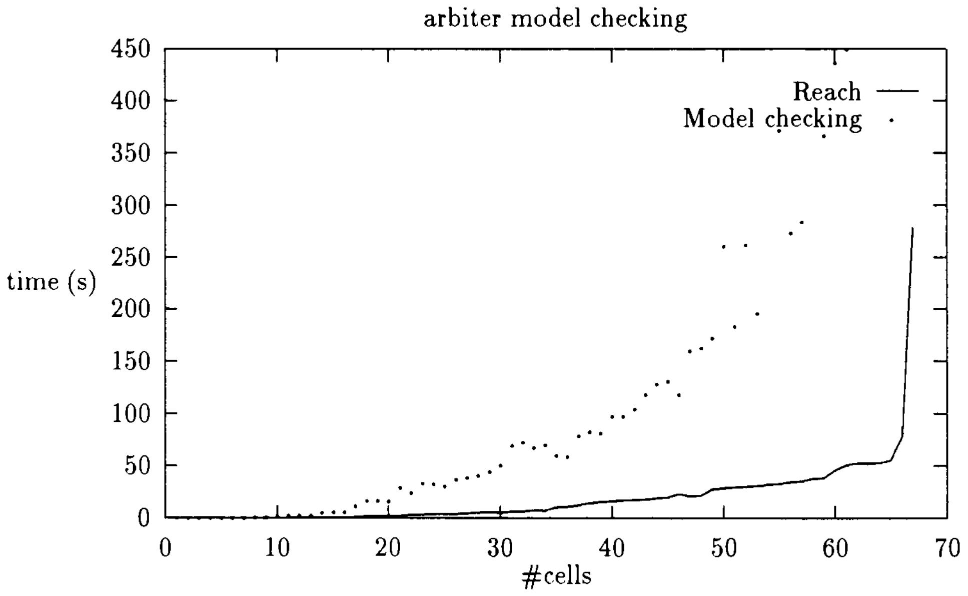

Figure 4.6: Model checking vs Reachability time use

4.4.1 Verification

For verification the arbiter as well as the minmax-processor have been used. In both cases there were two identical versions of the circuit used for verification. Only the names of the variables were different. Just up to a relatively little number of bits/cells the verification works. The data for minmax are given in table 4.4.1. The results for the arbiter are given

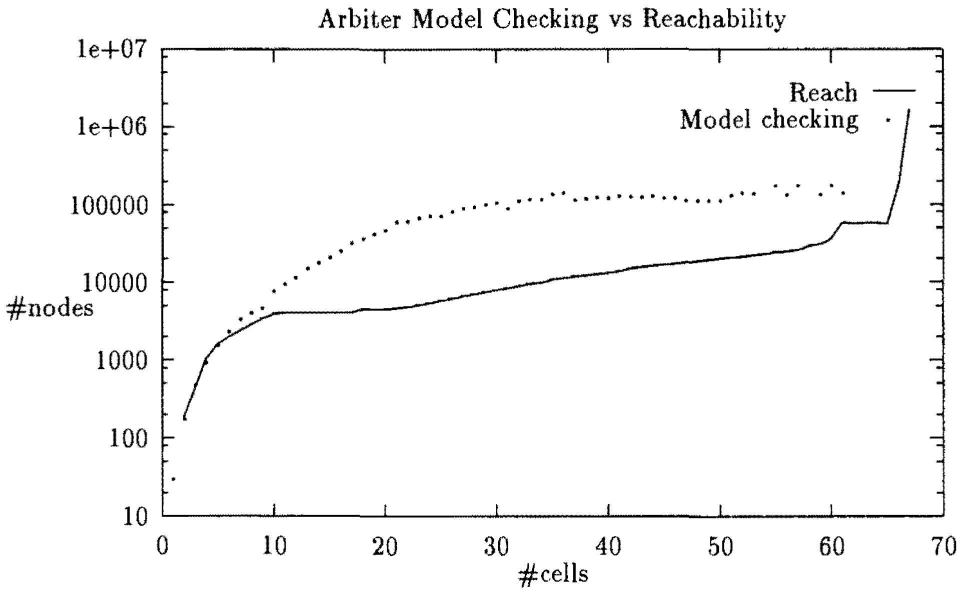

Figure 4.7: Model checking vs Reachability memory use

Table 4.4: Verification of Minmax processor

#bits	#nodes	Memory (kb)	Time (s)
1	314	268	0.18
2	1634	270	0.21
4	9335	496	5.84
6	22291	865	14.97
8	34208	1160	27.04
10	46528	1527	48.93
12	59957	2189	65.36
14	73510	2534	82.86
16	85929	2683	112.78
18	115457	3329	152.02
20	-	40000	63652[1]

in figure 4.8. Both results are calculated using the set of reachable states as a restriction for verification.

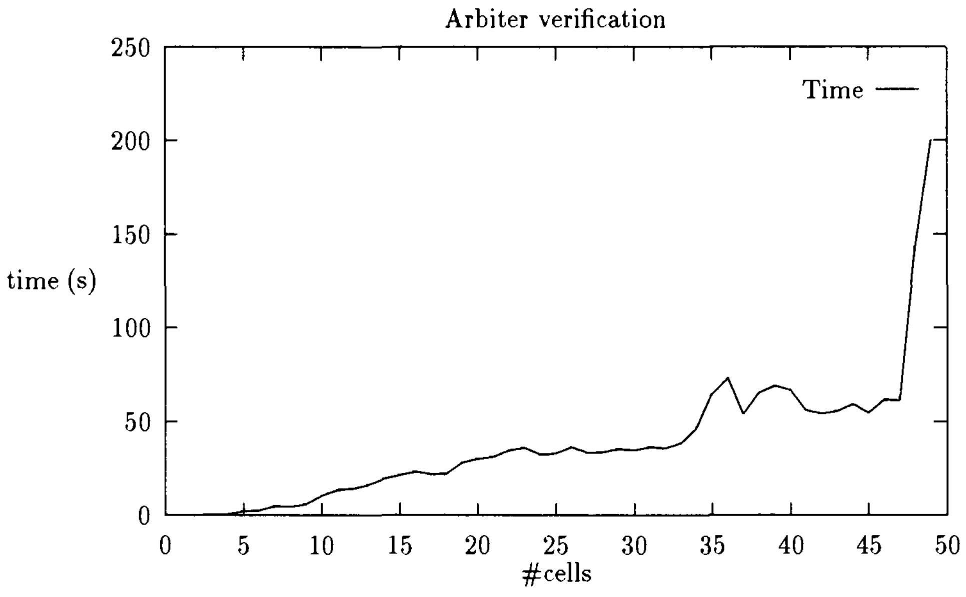

Figure 4.8: Verification time use

[1]Cancelled after 17 hrs

Chapter 5

Conclusions

5.1 Reachability

The program is quite able to do reachability analysis for not too large circuits. This is done in reasonable time, without excessive memory use. Clearly the heuristics leave their traces in the results, where in the middle of a test range the use of time and memory suddenly increases sharply, but decreases almost as fast as it increased. Also another sequence of actions in calculating the transition relation makes a big difference in maximum size of the circuit. Since the results does not change it is clearly the variable ordering that might be different. An extra option has been added to set the way the transition relation is calculated. It differs from example to example which way works faster.
Compared to other tests, this program does reachability calculation a lot faster, and can handle much more states. Introducing some algorithms that combine calculations made it even faster.

5.2 Model checking

In this part of the program we didn't fully implement the algorithm. It is possible to do model checking, but only on outputs. The possibilities until now remain restricted to entering the model properties in the program and compile and link them. For tests run, the results are ok. Times are slightly better compared to tests done by McMillan.

5.3 Future work

- Implementing a parser to read property-files and process them.
- Extending the CTL to a more powerful version with possibilities for entering restrictions on times, and with macro's and more understandable coding. Statements as given in [Pay94], [BBD94] and [BLP95] can be used as examples.
- Introducing cutpoints into the routines to handle bigger circuits.
- Implementing routines to determine a counterexample if the desired properties are not met by a circuit.

References

[ECl81] *Emerson E.A. & E.M. Clarke,*
Synthesis of synchronization skeletons for branching time temporal logic.
Logic of Programs: Workshop, Yorktown Heights, NY, May 1981,
Lecture Notes in Computer Science, volume 131. Springer-Verlag, 1981

[Bry85] *Bryant R.E.,*
Symbolic Manipulation of Boolean Functions Using Graphical Representation.
Proceedings 22th Design Automation Conference,
June 1985; pp 688-694

[Bry86] *Bryant R.E.,*
Graph Based Algorithms for Boolean function Manipulator.
In: IEEE Transactions on Computers Volume C-35(8), pp 677-691, August 1986.

[Jan90] *Janssen G.L.J.M.,*
Hardware Verification using Temporal Logic: A Practical View.
IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design 1988,
Houthalen, Belgium; 13-16 November 1989,
Editor : Dr. Luc Claesen,
Amsterdam : Elsevier Science Publishers B.V. 1989

[Mcm93] *McMillan K.L.,*
Symbolic Model Checking.
Kluwer Academic Publishers, Norwell Massachusetts 1993

[Pay94] *Payer Michael,*
CVE : Circuit Verification Environment,
Formal Verification for an Industrial Design Environment.
Siemens AG Munich 1994,
Internal Report,
Personal Communication

[BBD94] *Beer Ilan & Shoham Ben-David, Daniel Geist, Raanan Gewirtzman, Michael Yoeli,*
Methodology and System for Practical Formal Verification of Reactive Hardware.
IBM Science & Technology,
Haifa, Israel 1994,
Internal Report,
Personal Communication

[BLP95] *Bormann Jorg & Thomas Filkorn, Jorg Lohse, Michael Payer, Gerd Venzl, Peter Warkentin,*
CVE : An Industrial Formal Verification Environment.
Siemens Corporate R&D,
Munich Germany 1995,
Personal Communication

[Met95] *Mets Arjen,*
Formal Verification of Sequential Circuits Using Implicit State Enumeration.
IBM Research Division,
T.J. Watson Research Center,
Yorktown Heights, NY 10598,
Internal Report RC 19894 (88043) 1/6/95,
Personal Communication

Appendix A

The bus-arbiter

The arbiter is a circuit to control access for a number of devices to a bus. There is a fixed order of priority, the cell that has the lowest number has the highest priority and will go first. But there is some fairness included in the circuit. If a request persists for maximum of 2·N times, with N the number of cells, its request will be granted. This is done by using a wait register. If a request for a certain cell is present at the time there is a token in the same cell, and there is a cell with a higher priority requesting the bus, then the wait register is set. If the request still holds the next time the token is present, the cell gets absolute priority above all other cells. There is one peculiarity in the circuit. Geert Janssen discovered this when he was analyzing the circuit for his Phd. thesis. If a request is dropped, at the same time it gets granted through a persist, no acknowledge is set at that moment. So one access possibility is waisted. This error can be fixed with a simple adaption made to the circuit. The and-gate with inputs from both registers, of which the output has the name *persist[i]*, must be expanded with another input, and this input must be the *request*-signal. In the next BSN-description the defined persist must be

```
#define persists(i)              (W[i] & T[i] & req[i]).
```

The definition in BSN for the arbiter is the following :

```
/* McMillan's definition. */
#define persists(i)     (W[i] & T[i])

/* Registers are to be connected externally. */
/* n >= 1 */
bbox arbiter[[n]] (input  req[0:n-1],
                          W[0:n-1],
                          T[0:n-1];
                   output ack[0:n-1],
                          @T[0:n-1],
                          @W[0:n-1]
                  )
{
  (i:0..n-1) {
    /* Round-robin token scheme. */
    /* Cyclic shift-register (rotate towards higher index) */
    @T[(i+1) % n] := T[i];

    /* Waiting for next-time token around? */
    @W[i] := req[i] & (W[i] | T[i]);

    ack[i] := req[i] &
              (    /* It's my turn (lowest index has priority): */
                   (& (j:0..i-1) ^req[j])
                   /* Nobody (else) is persistent: */
                 & (& (j:0..n-1) ^persists(j))
               | persists(i) /* I am the persistent request */
              );
  }
}

/* n-bit register. */
bbox register[[n, Initval]] (input D[0:n-1];
                             latch Q[0:n-1] init Initval)
{
  Q := D;
}

/* The complete arbiter circuit. */
cbox ARBITER[[n]] (input req[0:n-1];output ack[0:n-1])
{
  net W[0:n-1], @W[0:n-1], T[0:n-1], @T[0:n-1];

  token:register[[n,0b1]] (@T, T);
  wait:register[[n,0]] (@W, W);
  arbiter[[n]] (req, W, T, ack, @T, @W);
}

top_instance ARBITER[[N]];
```

A single cell of the arbiter is depicted in figure A.1.
The total circuit is shown in figure A.2.

The BSN-implementation for the arbiter, and both figures A.1 and A.2 were written and put at my disposal by G. Janssen.

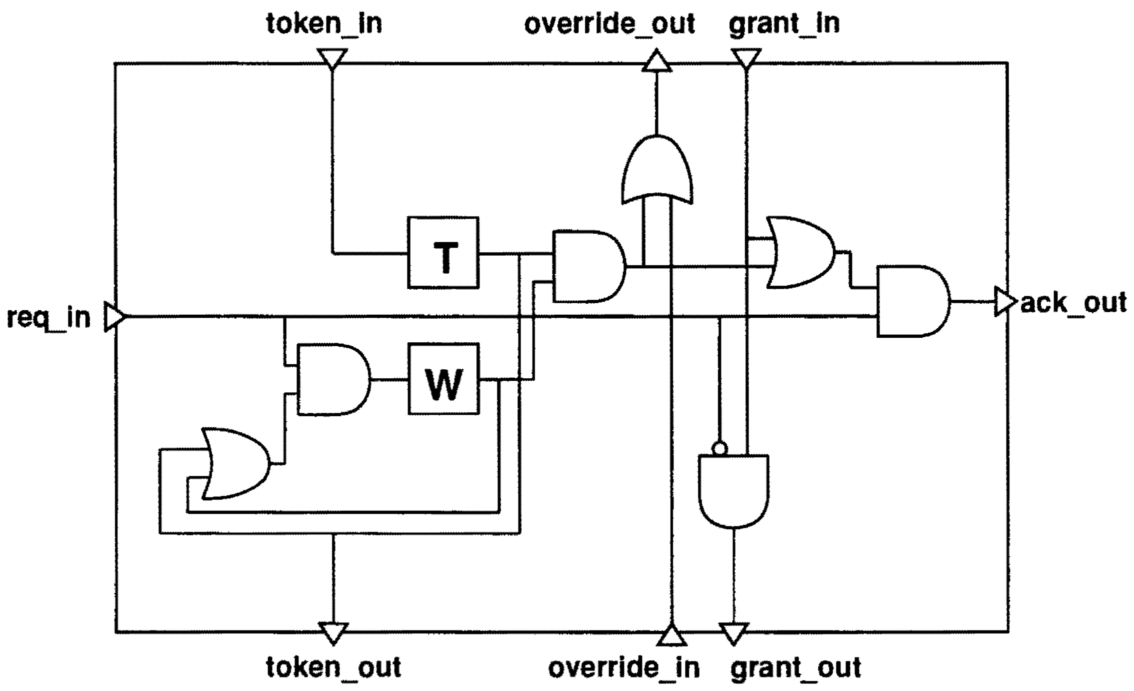

Figure A.1: One cell of the arbiter

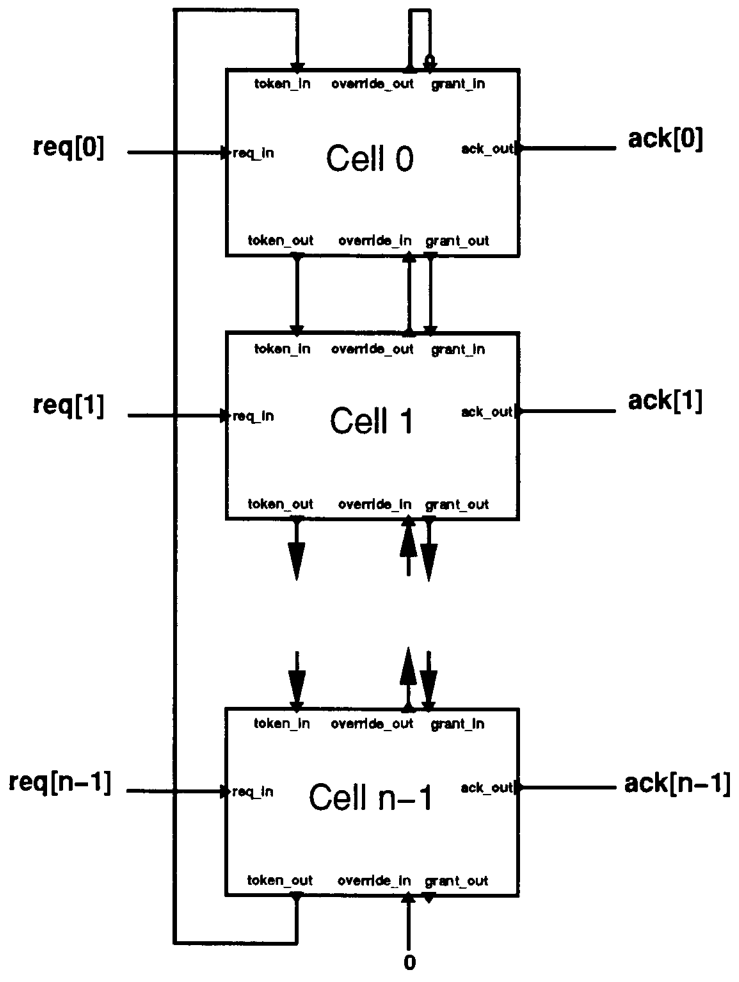

Figure A.2: The arbiter

Appendix B

Description for Minmax circuit

The Minmax signal processor is an electronic circuit to calculate some statistical data on a stream of data. The circuit calculates the minimum and maximum of the input data, and the average between maximum and minimum. There are three control inputs :

1. CLEAR sets all registers to zero, except for MIN, which is set to all one's, representing the maximal value possible.
2. define the handling of the data. If ENABLE is 1 and RESET is zero, the circuit accepts data and stores it in the registers. The output is equal to the average of MIN and MAX.

The BSN-description is given in the next paragraph. The parameter N in the top_instance is the number of bits for each register.

```
bbox Min_Max_3[[n]] (
                    input CLEAR, ENABLE, RESET;
                    input IN[1:n],
                    LAST[1:n],
                    MIN[1:n],
                    MAX[1:n];

                    output OUT[1:n];
                    output $LAST[1:n], $MIN[1:n], $MAX[1:n];
                    )
{
  local sup[1:n], inf[1:n], avg[1:n];
  local ones[1:n];

  ones := extend (1, n);

  table sup_tab
  (IN > MAX -> sup) {
    0 -> MAX;
    1 -> IN;
  }
  table inf_tab
  (IN < MIN -> inf) {
    0 -> MIN;
    1 -> IN;
  }

  /* Average: sum / 2: */
  avg || - := sup + inf;

  table tab
  (CLEAR, ENABLE, RESET ->  OUT, $LAST, $MAX, $MIN) {
       1,      X,      X ->    0,     0,    0, ones;
       0,      0,        -> LAST,  LAST,     ,     ;
        ,      1,      1 ->   IN,    IN,     ,     ;
        ,       ,      0 ->  avg,      ,  sup,  inf;
  }
}

bbox Register[[n,m]] (input IN[1:n]; latch OUT[1:n]
                       init m*(2**n-1))
{
  OUT := IN;
}

bbox sink (input A)
{
  - := A;
}

cbox Min_Max[[n]] (
                   input CLEAR, ENABLE, RESET;
                   input IN[1:n];
                   output OUT[1:n];
                   output MIN[1:n], MAX[1:n];
                   )
{
  net LAST[1:n];
  net $LAST[1:n], $MIN[1:n], $MAX[1:n];

  Min_Max_3[[n]] (CLEAR, ENABLE, RESET, IN, $LAST, $MIN, $MAX,
                  OUT, LAST, MIN, MAX);

  Register[[n,0]] (LAST, $LAST);
  Register[[n,1]] (MIN, $MIN);
  Register[[n,0]] (MAX, $MAX);
}

top_instance Min_Max[[N]];
```

A graphical view how the circuit might look is given in figure B.1. The three cells MIN, LAST en MAX are N-bit registers with a reset-input. The control-unit contains some combinatorial logic that selects one of its inputs, or calculates the average of the MIN and MAX registers.

The BSN-implementation for the Minmax signal processor has been written by G. Janssen.

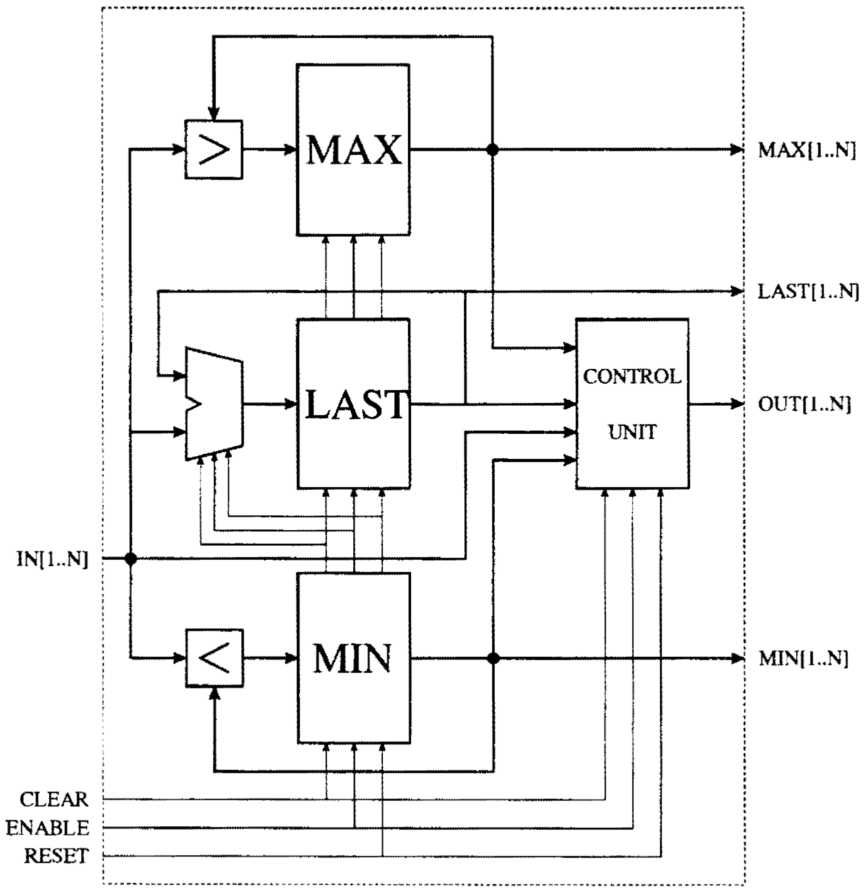

Figure B.1: Circuit for Minmax

Appendix C

BSN operators

This appendix will give a short overview of the BSN-operators that are used in the examples:

Table C.1: BSN operators

Symbol	Meaning
|	or
&	and
=	equal
^	not
^=	notequal
<=	less or equal
>=	greater or equal
>	greater
<	less
&&	exor
+	sum
%	modulo
||	concatenate