USING LOGIC PROGRAMMING AND COROUTINING FOR ELECTRONIC CAD

ULRICH BIEKER AND ANDREAS NEUMANN

▷ We show how an extended Prolog can be exploited to implement different electronic CAD tools. Starting with a computer hardware description language (CHDL) several problems like digital circuit analysis, simulation, test generation and code generation for programmable microprocessors are discussed. For that purpose the MIMOLA (machine independent microprogramming language) system MSS (MIMOLA hardware design system) is presented. It is shown that logic programming techniques have several advantages especially in the area of integrated circuit design. One of the main advantages is the small code size which translates to easy maintenance. We make extensive use of two main features of standard Prolog and constraint logic programming, i.e. backtracking and coroutining mechanism to express Boolean constraints. ◁

1. Introduction

Due to the increasing complexity of digital circuits, the design process is supported by design tools covering a wide range of problems like synthesis, simulation, verification, test generation, microcode generation, placement, routing etc. For readers not familiar with VLSI design we first describe the design subtasks before describing our work. Many of these subtasks are of high complexity, e.g. test generation is even NP-complete. Therefore, electronic CAD systems, commonly written in imperative languages, consist of a very large amount of source code. Maintenance, portability and adaptability are problems. We will describe significant software engineering advantages by the use of Prolog for these problems.

Address correspondence to Ulrich Bieker, University of Dortmund, Department of Computer Science, D-44221 Dortmund, Germany, e-mail: bieker@ls12.informatik.uni-dortmund.de

Address correspondence to Andreas Neumann, University of Trier, Department of Computer Science, D-54286 Trier, Germany, e-mail: neumann@ti.uni-trier.de

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Inc., 1994
655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

MIMOLA [2] is a computer language with Pascal-like constructs. It supports design, test, simulation and programming of digital computers and is integrated into the CAD system MSS [19, 20]. MIMOLA, influenced by other hardware description languages like VHDL [15], allows structural and behavioral descriptions of circuits. Originally the complete system was written in Pascal but beginning with MIMOLA 4.0 we started to redesign tools using Prolog.

Using the extended Prolog system ECLIPSE [11] new concepts to solve problems in the area of digital circuit design have been found. For example coroutining, which allows the user to express a condition under which a call to a specified goal will be delayed, is a very useful mechanism to avoid unnecessary backtracking during simulation, test and code generation.

Several approaches to digital circuit design using logic programming have been presented [13, 27, 12, 8, 26, 25, 10], most of them concentrating on the gate level or even lower levels of abstraction. Only a few contributions consider higher levels of abstraction in the context of logic programming [21, 24, 17, 9, 14].

In this paper we describe the use of Prolog for a very high level of abstraction. A very elegant simulator, based on a hardware description language and a suitable Prolog circuit representation based on trees is presented. The simulator is able to simulate a processor together with a given microprogram. We also present a concept to generate microcode for a given hardware structure which can be used to test the processor. The part of the MSS system concerning this paper is shown in figure 1.1.

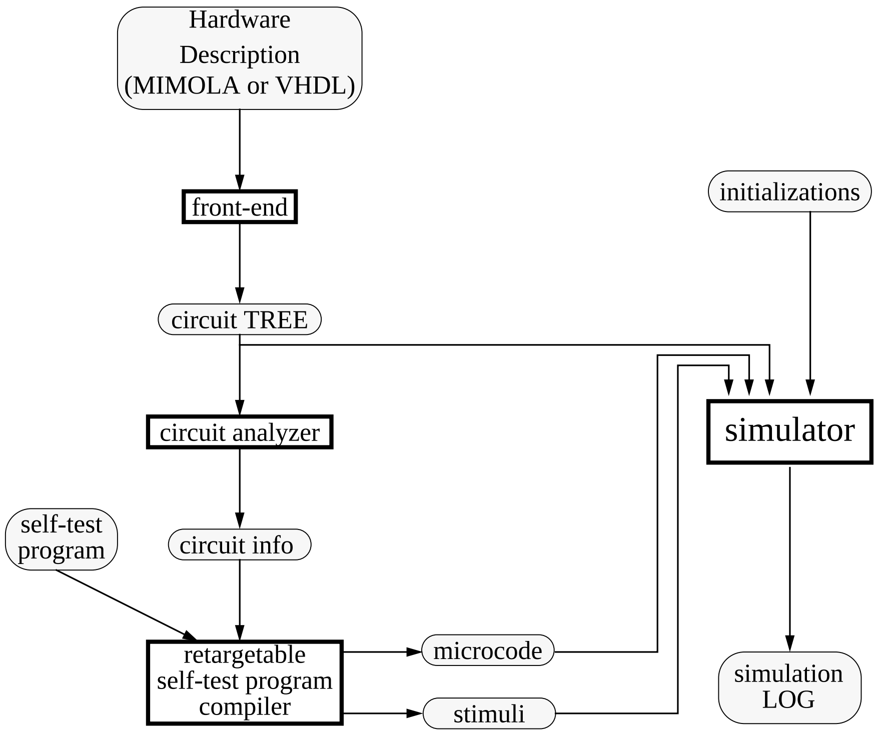


FIGURE 1.1. System Overview

Starting with a circuit given as a MIMOLA or VHDL RT-level hardware description a tree-based Prolog circuit representation is generated by a front-end compiler. Afterwards, a circuit analyzer creates a circuit info file that can be used as input for the code generator. A retargetable compiler maps a program onto the given hardware, resulting in a microprogram and a set of external stimuli patterns for the primary inputs. Finally the generated program can be simulated together with the circuit description, an initialization file for registers and memories and the set of stimuli.

In what follows we first introduce a small processor to be used as an example throughout the paper. We continue with the simulator concept based on three levels of abstraction, followed by a section describing the circuit analyzer. In the last section we generate a load instruction as a typical example of microcode generation.

2. SIMPLECPU: A small example processor

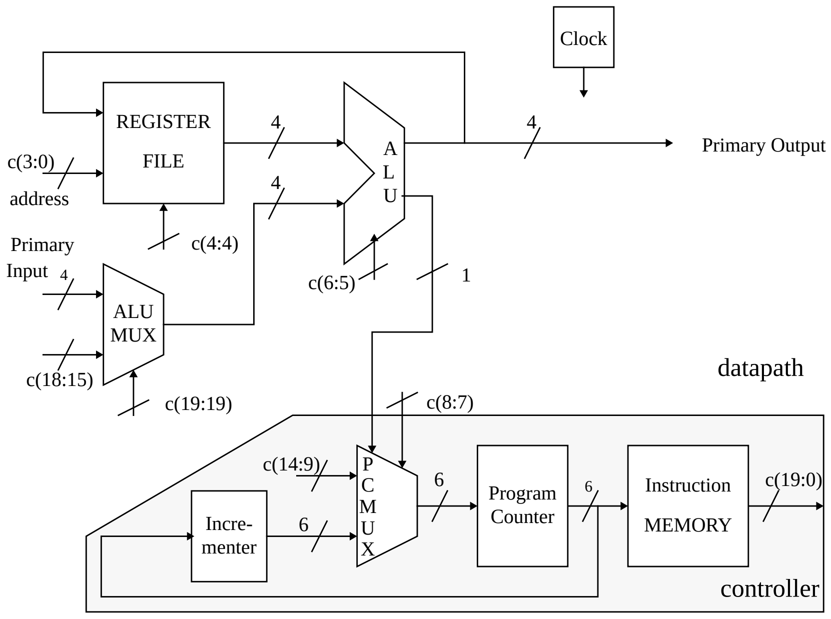


FIGURE 2.1. SIMPLECPU

Figure 2.1 shows SIMPLECPU, a small programmable microprocessor consisting of 8 modules. The SIMPLECPU controller (shaded area) consists of a program counter, an instruction memory, an incrementer and a multiplexer. A 16x4 register file, a 4-bit ALU, a second multiplexer and a clock are also part of the CPU structure. The register file and the program counter are connected to the clock (not shown) and control signals are denoted by ‘c’ followed by an index range. MIMOLA hardware descriptions contain register transfer modules, their behavior and their interconnections. For instance, the 4-bit ALU is specified in MIMOLA as follows:

Example 2.1.

```
MODULE ALU      (IN a, b : (3:0); IN ctr : (1:0);
                OUT result: (3:0); OUT condition:(0:0 ));
BEHAVIOR IS     CONBEGIN
                result <-       CASE ctr OF
                                0 : a ;
                                1 : b ;
                                2 : a+b ;
                                3 : a-b ;
                                END AFTER 1;
                condition <-    CASE ctr OF
                                0 : a = 0 ;
                                1 : b = 0 ;
                                2 : a+b = 0 ;
                                3 : a-b = 0 ;
                                END AFTER 0;
                CONEND;
```

CONBEGIN and CONEND denote a concurrent block, containing two case expressions as assignments to the outputs. In MIMOLA, the default data type is the bit vector. Its index range is denoted as (high-bit : low-bit), i.e. the ALU has two 4-bit data inputs *a* and *b*, a 4-bit output *result*, a 1-bit output *condition* and a 2-bit control input *ctr* selecting the ALU function.

Using MIMOLA as the input language, we generated a tree-based Prolog intermediate format in two steps. First MIMOLA is transformed into TREEMOLA [7, 5], an intermediate language of the MSS. The second step is done by a converter written in Prolog, which leads to a circuit representation as a list of module descriptions. Every module consists of a list of connections, a list of storing cells and a behavior tree as shown in figure 2.2 for a part of the ALU. Such a tree is easily represented by a Prolog structure. The list of connections contains information about inputs and outputs of the module and interconnections to other modules. Every signal is represented by a logic variable and this variable also occurs in the behavior tree when the signal is referenced. If signals are instantiated elsewhere, this leads to an immediate signal propagation to all modules using this signal.

3. Simulation of a CHDL

The implemented simulator is based on three levels of abstraction: the built-in operators, an interpreter for the behavior of a single component and an event driven simulator for a circuit together with the microcode. Especially for the implementation of the operators, we made extensive use of the coroutining concept of the Eclipse language.

3.1. Implementation of Operators

In order to interpret a Hardware Description Language, an implementation of its built-in operators is necessary. These range from logic primitives to complex arithmetic operators. They are represented as Prolog predicates, which mainly have to meet the following criteria:

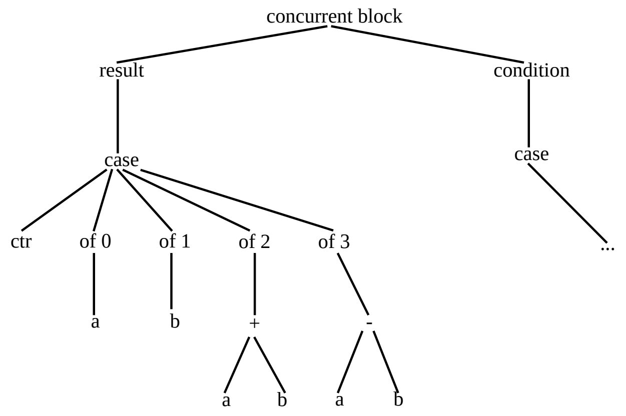


FIGURE 2.2. Behavior Tree

1. The operators must work bidirectionally, so that they can also be used for backward simulation of a circuit.
2. They should work deterministically, i.e. subsequent backtracking steps do not produce the same solution. This is especially important for backward simulation. This is because the mapping of an operator is not necessarily definitely reversible. Certain backtracking alternatives have to be pruned to avoid duplicate solutions.
3. The computation must be, at least at the operator level, data driven, i.e. the application of an operator to unbound variables is propagated symbolically as a delayed goal, until the instantiation of the variables is absolutely unavoidable. In this way, the number of backtracking steps is reduced.

The third point is achieved by using the coroutining mechanism of the Eclipse language, which allows the programmer to specify conditions, under which the execution of a goal shall be delayed, depending on the bindings of its parameters. Whenever a variable occuring in one of these is bound, either to a value or another variable, the goal will be enabled, and the delay conditions will be checked again.

At the end of a simulation the set of all delayed constraints must be consistent. There should be a constraint solver[1]which finds contradictions and, if possible, solutions for variable bindings. Since such a constraint solver is rather complex, there should only be a few types of constraints. It would be sufficient to consider a minimal complete set of operators, but for efficiency reasons we used a set containing AND, OR, XOR and NOT. The Prolog code for those operators is divided into delay clauses and program clauses, e.g. the logical AND is implemented as *and/3*, with X and Y as input parameters and Z as output parameter:

```
delay and(X,Y,Z) if var(X), var(Y), var(Z), X\ ==Y.
delay and(X,Y,Z) if var(X), var(Y), Z==0, X\ ==Y.

and(X,Y,Z) :- nonvar(Y), !, and1(Y,X,Z).
```

[1]We developed a Boolean constraint handler using the constraint handling rules (CHR) of the Eclipse system.

```
and(X,Y,Z) :- nonvar(X), !, and1(X,Y,Z).
and(X,X,X).

and1(0,_,0).
and1(1,X,X).
```

The delay clauses cover the case, when the two input parameters are distinct unbound variables, and the output parameter is either unbound or zero. In these cases it is impossible to draw any conclusion, so the call to the predicate is delayed. The program clauses use the commutativity of the logical AND. The first two of them deal with the case when one of the inputs is bound, and invoke *and1/3* with the bound input as the first argument. For the third clause there are, due to the delay clauses, only two possibilities left: either the output is 1, which forces the inputs to take the same value, or the two inputs are identical variables, to which the output will be bound, too. The auxiliary predicate *and1/3* expects its first input to be instantiated. If it is bound to a 0, the result must be 0 either, if it is 1, the output is identical to the second input.

The more complex operators are based on these four logical primitives, e.g. a full adder can be defined as follows:

```
halfadd(In1, In2, Sum, Cout) :-
        and(In1, In2, Cout),
        xor(In1, In2, Sum).

fulladd(In1, In2, Cin, Sum, Cout) :-
        halfadd(In1, In2, Sum1, Carry1),
        halfadd(Cin, Sum1, Sum, Carry2),
        or(Carry1, Carry2, CarryOut).
```

Of course the set of operators is not restricted to single bit operations, but for each of them there is also a version for bit strings, which are represented as lists. On top of these there are built arithmetic operators like addition and multiplication and string manipulation operators like shifting and concatenating.

3.2. Interpretation of a Behavior Tree

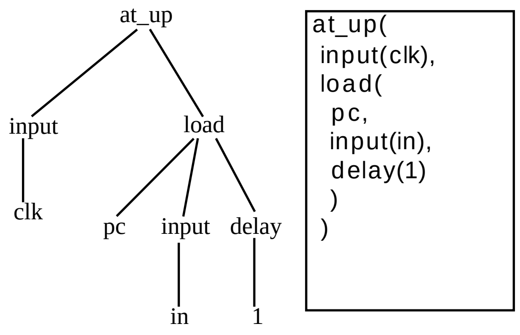
For the interpretation of the behavior tree of a module it is necessary to model the context, i.e. the contents of memory cells and input signals at a given time. A signal is now represented as a sorted binary tree, with a time and a value mark at each node. Readers familiar with logic programming recognize this as a common dictionary. Updates on a signal are realized by the following predicate:

```
sigValue((T,Val, _Before, _After), T,Val) :- !.
sigValue((Time, _Val, Before, _After), T,Val) :-
        T<Time, sigValue(Before,T,Val).
sigVal((Time, _Val, _Before, After), T,Val) :-
        T>Time, sigValue(After,T,Val).
```

Lookups are realized in a similar way, except that if there is no entry for the specified time, the least recent entry must be found, because the signals are assumed to be holding.

Input ports of a module and memory cells are represented by port descriptions, which are nothing more than lists of such signals. The contents of memory cells of a circuit are held in a dictionary. This dictionary is stored in a binary tree structure similar to the signal tree. In this case the search key is an atom consisting of an identifier and a list of values. The identifier can be the name of a register, or a pair consisting of a memory name and an address. The values are port descriptions. The comparison predicates in the clauses of the concerned lookup-predicate must then be replaced by the standard term order comparators.

The interpreter itself has only three parameters: the behavior tree, a time frame and the dictionary with all global values in it, and is implemented inductively on the structure of the behavior tree. Such a tree normally consists of some concurrent statements, which may contain nested expressions. For the interpretation of statements, a behavior tree and the corresponding representation as a Prolog term are shown in figure 3.1. The figure shows the statement behavior tree for loading the program counter when the clock rises.



```
at_up(
 input(clk),
 load(
  pc,
  input(in),
  delay(1)
  )
 )
```

FIGURE 3.1. Program Counter Behavior Tree

Calling the interpreter with this statement tree will invoke one of the following clauses:

```
interpret(at_up(ClkExp,Statement), Time,Dictionary) :-
        Time1 is Time-1,
        interpret_exp(ClkExp, Time, Dictionary,[1]),
        interpret_exp(ClkExp, Time1, Dictionary,[0]),
        !,
        interpret(Statement,Time,Dictionary).
interpret(at_up(_Clk, _Stmnt), _Time, _Dictionary).
```

If the calls to *interpret_exp/4* are successful, the interpreter calls itself with the *load* statement as an argument. This call will relate to the following clause, which adds the specified delay factor to the current time and enters the value of the input expression into the port description of the program counter, which is taken from the dictionary:

```
interpret(load(RegId, Expr, delay(Delay)), Time,Dictionary) :-
        lookup(RegId,Dictionary,PartPort),
        interpret_exp(Expr,Time,Dictionary,Value),
        NewTime is Time+Delay,
        portValue(PartPort,NewTime,Value).
```

Note that the *delay* structure in the behavior tree is distinct from the coroutining built-in with the same name. Other constructs like conditional or case statements, writing to an output port, concurrent nodes etc. are implemented similarly.

The interpreter for expressions has one more argument for returning the value of an expression. Except for this, its structure is the same (consider the behavior of the program counter incrementer (fig. 3.2)). The interpreter clause for the *output*

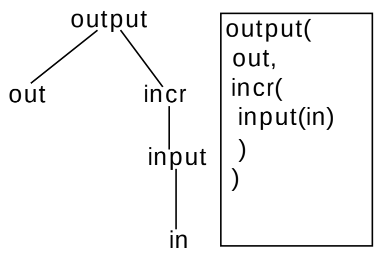


FIGURE 3.2. Incrementer Behavior Tree

tree, which is very similar to that for the load statement, will call *interpret_exp/4* with the *incr* subexpression, invoking the following clause:

```
interpret_exp(incr(Expr), Time,Dictionary,Value) :-
        interpret_exp(Expr,Time,Dictionary,Value),
        incr(Expr,Value).
```

The method is, first to evaluate the arguments of an operator and then to apply it to the results. The dictionary is needed here only for the *read* expression, which is evaluated as the value of a storage. More complex expressions like the conditional or case construct are implemented in the same way.

3.3. An Event Driven Simulator

The task of the simulator is to simulate the behavior of a circuit, given the initial state of the storage and the values of the primary inputs for the considered time interval. The circuit consists of a set of modules with a specified behavior which are interconnected by some signals. In an event driven simulator an event is a pair consisting of a time and a module behavior. All events yet to be simulated are held in a queue, which is initialized at the start of the simulation by all events which are involved by the change of a primary input, the toggle of a clock or the initialization of a register or memory. A new event for a module is generated if and only if at least one of its input signals or one of its storing cells has changed due to simulation of a former event. Thus the execution of one event is the following:

```
doOneEvent(Modul,Time,Dictionary,NewEvents):-
        Modul = (Name, Behavior, Connections, Stores),
        interpret(Behavior,Time,Dictionary),
        storeEvents(Stores,Name,Time,Events1),
        newEvents(Connections,Time,Events2),
        append(Events1,Events2,NewEvents).

newEvents([ ] , _, [ ] ).
newEvents([(Mod,Signals)|RestCons],Time, [(Mod,ChangeTime—RestEvents]):-
        lastChange(Signals,ChangeTime),
        ChangeTime > Time, !,
        newEvents(RestCons, Time,RestEvents).
newEvents([_|RestConnections], Time,Events):-
        newEvents(RestConnections, Time,Events).
```

The predicate *storeEvents/4* is similar to *newEvents/3*, but checks the storage of the module for changes and if any is detected, generates an event for the same module. Note that the event queue must be sorted and allow no duplicates. Moreover, there must be a kind of priority for the order of simulation of two events with the same time, a feature which was omitted here.

The simulator itself is now defined as follows:

```
simulate_circuit(CircuitName,MaxTime) :-
        ... % get the circuit informations
        doClocks(Clocks,MaxTime,ClockEvents),
        doInPorts(Stimuli,InPorts,InputEvents),
        initStores(InitLines,Dictionary,InitEvents),
        mergeEvents(ClockEvents,InputEvents,InitEvents,Events),
        doAllEvents(Events,Dictionary,MaxTime).

doAllEvents([ ] , _, _).
doAllEvents([( _, Time) | _], _, MaxTime) :-
        Time > MaxTime.
doAllEvents([Event| RestEvents], Dict, MaxTime):-
        doOneEvent(Event,Dict,NewEvents),
        merge(NewEvents,RestEvents,EventsAfter),
        doAllEvents(EventsAfter,Dic,MaxTime).
```

The predicate merges the new events after each step with the remaining ones from the queue and calls itself recursively with the result, until the queue is empty or the maximum time is reached.

For simulating a circuit together with a microprogram, one only has to specify the code as initialization to different lines of the instruction memory and start the simulator. Consider the following example program for SIMPLECPU:

Example 3.1.

```
PROGRAM sum_up IS
VAR x : nibble;
```

```
BEGIN
        x := 1;
        REPEAT x := x+pi; UNTIL x = 0;
STOP;
```

After initializing a variable x with 1, a loop adds x to the value of the primary input, until x is zero. The microcode shown in table 3.1 consists of 5 instructions. IM 0 denotes the memory content of the instruction memory with address 0.

Bits	19	18:15	14:9	8:7	6:5	4	3:0
IM 0	0	0001	XXXXXX	00	01	0	0000
IM 1	X	XXXX	XXXXXX	00	XX	1	XXXX
IM 2	1	XXXX	XXXXXX	00	10	0	0000
IM 3	X	XXXX	000001	10	00	1	0000
IM 4	X	XXXX	XXXXXX	00	XX	1	XXXX

TABLE 3.1. Example microprogram

With the primary input constantly set to [0,1,0,1], the simulation of this program passed 149 events, which took 0.83 seconds of cputime on a SPARC 10 workstation. Note that every event means simulation of a complete behavior tree.

4. Circuit Analysis

4.1. Simulator Priorities

When simulating a circuit it is necessary to give priorities to different modules concerning the order in which to simulate two events at the same time. The reason for this are the causal dependencies between components which are connected without delay. This priority can be compared to the Δ-delay of VHDL. The intention is that an event may be simulated only when all events its inputs depend on have been considered before, i.e. the priority of a module is the maximum of the priorities of all its predecessors incremented by one. Assume that we have already computed a priority list of triples *(Mod,Prio,Preds)*, where *Pred* is a list of pairs *(Mod',Prio')*, so that every occurence of a module in the whole structure have its *Prio* component bound to the same variable. Now, for each element of the priority list, we only have to compute the maximum priorities in the predecessor list and bind the priority to this value incremented by one.

```
delay max(A,B,M) if var(A).
delay max(A,B,M) if var(B).

max(A,B,A) :- A > B,!.
max(A,B,B).
```

```
maxPriority([ ], Max, Max).
maxPriority([(_, Prio)| Rest], Max0,Max) :-
        max(Prio,Max0,Max1),
        maxPriority(Rest,Max1,Max).

setPriorities([ ]).
setPriorities([(Mod,Prio,Preds) | Rest]) :-
        maxPriority(Preds,0,MaxPrio),
        plus(MaxPrio,1,Prio),
        setPriorities(Rest).
```

In standard Prolog this would lead to difficulties because we could not compare unbound variables. This is easily resolved by delaying the *max/3* predicate. If there are no critical races in the circuit, i.e. there are no cyclic dependencies, there must be at least one module whose predecessor list is empty, so it will get priority 1. This will wake up at least one other *max* goal, and so on, so that all priorities will be computed correctly. If there is a cycle, then a conflict occurs and an error must be raised. Such a con flict can easily be detected by checking for delayed goals by a system call. Note that the *plus/3* predicate must also be delayed, which is done automatically by Eclipse.

4.2. Microcode Preparation

To prepare code generation, several tasks are done by the circuit analyzer. The main task is to generate a lot of facts describing special characteristics of a given circuit to reduce the complexity of code generation. Application of some facts is shown in the following section. Table 4.1 gives an overview of some generated facts but due to the lack of space not all generated facts can be considered in detail. In the following we want to describe these facts and the methods to generate them.

One of these facts is *transparent/3*, denoting an identity mapping from one input to at least one output, so that the module becomes 'transparent'. That means that with a special control code, the considered module is able to pass one input to one output. Most of these facts might be found at multiplexer modules. On the other hand the *transparent/3* example of table 4.1 shows a possibility to switch input *a* of the SIMPLECPU alu to the output *result*, i.e. the signal list [D,C,B,A] is switched. This is done by unifying input *b* with the neutral element [0,0,0,0], to perform an identity mapping for the selected operator. The binary control code c(6:5) = [1,0] selects the add operator of the concerned ALU.

How can we generate a *transparent/3* fact? Using the interpreter and the operators defined in section 3, this task is easy to solve. The basic idea is to unify one module input with one module output and to perform an interpretation step for this module. The interpretation step has to lead to an instantiation of some inputs for the following reasons:

1. Choosing a control code to select an operation that is able to perform an identity mapping (e.g. c(6:5) = [1,0] to select ALU addition).

2. If necessary, choosing a neutral element for the selected operation (some operations do not need a neutral element, e.g. a multiplexer or the ALU

fact/arity	arguments	example
transparent /3	module name list of inputs list of outputs	transparent(alu, [[D,C,B,A], [0,0,0,0], [1,0]], [[D,C,B,A], [Condition]]).
path /3	source destination Path	path(im,reg, [(im,[[_,_,_,_,_,_]], [[0,D,C,B,A,_,_,_,_,_,_,_,_,0,1,_,_,_,_,_]]), (mux, [[_,_,_,_], [D,C,B,A], [0]], [[D,C,B,A]]), (alu,[[_,_,_,_], [D,C,B,A],[0,1]], [[D,C,B,A],[_]]), (reg, [[_,_,_,_], [D,C,B,A],[_],[_]], [[_,_,_,_]])]).
incrementPC /2	delayed goal Path	incrementPC(incr([F,E,D,C,B,A], [L,K,J,I,H,G]), [(inc, [[F,E,D,C,B,A]], [[L,K,J,I,H,G]]), (pcmux, [[_],[0,0],[L,K,J,I,H,G],[_,_,_,_,_,_]], [[L,K,J,I,H,G]]), (pcreg, [[L,K,J,I,H,G], [_]], [[F,E,D,C,B,A]])]).
jump /1	Path	jump([(im,[[_,_,_,_,_,_]], [[_,_,_,_,_,_,F,E,D,C,B,A,0,1,_,_,_,_,_,_,_,_]]), (pcmux,[[_],[0,1],[_,_,_,_,_,_],[F,E,D,C,B,A]],[[F,E,D,C,B,A]]), (pcreg, [[F,E,D,C,B,A], [_]], [[_,_,_,_,_,_]])]).

TABLE 4.1. Some selected facts, generated by circuit analysis

operation selected by the control code c(6:5) = [0,0] to switch input *a* to the output *result*).

A successive selection of all operations performed by a module is done by backtracking. Afterwards, the selected operation has to be executed symbolically, holding the input port to be switched as list of variables. Execution of the selected operation *Op* is done by the clause *findTransparent/4*. The lists library predicate *checklist/2* succeeds if *var/1* succeeds for every element of *SwitchPort*, ensuring that the selected input is switched to the selected output for all possible values of *SwitchPort*. Finally, we assert the generated fact.

```
findTransparent(Module, Op, InPorts, OutPorts):-
        member(SwitchPort, InPorts),
        member(SwitchPort, OutPorts),
        Operation =.. [Op, InPorts, OutPorts],
        call(Operation),
        checklist(var, SwitchPort),
        assert(transparent(Module, InPorts, OutPorts)),
        fail.
findTransparent(_, _, _, _).
```

The fact considered next is *path/3*, describing a path from a source module to a destination module, possibly through certain other modules which are able to perform an identity mapping. A fact *path/3* is a triple with parameters source, destination and Path. Path is a list of triples (module name, list of inputs, list of outputs). The first element of the list is the source module whereas the last element is the destination module. All modules between source and destination are able to switch an input to an output by the use of *transparent/3*. A *path/3* fact contains all control codes, i.e. signals which have to be 0 or 1 to switch the *Path*. As source and destination only sequential modules, i.e. modules that are able to store a value, are considered. Additionally, modules able to yield a constant, e.g. a decoder, can serve as a source. The example given in table 4.1 shows a *Path* from the instruction memory *im* through the multiplexer *mux* and the *alu* to the register file *reg*. Therefore binary control codes c(19) = [0] for the multiplexer and c(6:5) = [0,1] to switch a via through the *alu* are selected. [D,C,B,A] is the list of values connected by this path.

A simplified version of the predicate generating *path/3* facts is *findPath/3*. The first clause terminates the search of a path if *Destin* is a direct successor of *Source*. In the second clause we try to find a path through a module *Next*, which has to be a successor of the current *Source* and must be switched into a transparent mode. Afterwards, a recursive search with *Next* as source is started. A lot of implementation details are omitted, e.g. the check to prevent entering a cycle and the complete circuit representation.

```
findPath(Source, Destin, [Source, Destin]):-
        successor(Source, Destin).

findPath(Source,Destin, [Source | RestPath]):-
        successor(Source, Next),
        transparent(Next, Inputs, Outputs),
        findPath(Next, Destin, RestPath).
```

A frequent subtask of microcode generation is to increment the program counter. Therefore we generate a symbolic increment instruction where the address is unbound. The real address will be instantiated at the end of code generation. For that reason we generate a delayed goal, so that the code generator is able to bind these addresses to real values with respect to certain constraints. As a consequence of that, an increment instruction *incrementPC/2* is a pair, containing a delayed goal which performs the increment operation and a *Path* from the output of the program counter to the input of the program counter. In the generated *Path* two occurences of the program counter are avoided by omitting the program counter as source. *Path* is a list of triples as described above. The given example of table 4.1 shows the unique solution to increment the program counter *pcreg* for the example processor. Therefore the binary control code c(8:7) = [0,0] is selected for the multiplexer *pcmux*. [F,E,D,C,B,A] is the current state of the program counter whereas [L,K,J,I,H,G] will be the next state. The delayed goal incr([F,E,D,C,B,A],[L,K,J,I,H,G]) denotes the operation to be executed at the end of code generation.

A further subtask of code generation is to perfom unconditional jumps, i.e. to move a constant value into the program counter without consideration of a condition from the arithmetic unit. Therefore, *jump/1* is simply a fact denoting a *Path* from a sequential source module to the program counter. SIMPLECPU has only one possibility to perform such an unconditional jump by selecting c(8:7) = [0,1] as control code for the multiplexer *pcmux* as shown in table 4.1. The new symbolic jump address [F,E,D,C,B,A] originates from the instruction memory *im*.

The facts *incrementPC/2* and *jump/1* are mainly generated by the use of *path/3* and *transparent/3*. Using failure driven loops (see e.g. *findTransparent/4*), all possible solutions of the described facts are generated and asserted.

We conclude this section by enumerating some additional facts not considered here:

1. *constant/3*: denotes a module that is able to yield a constant as output (e.g. a decoder).
2. *conditionalJump/2*: denotes a conditional jump version, i.e. a conditional path to the program counter.
3. *noload/3*: denotes a micro instruction, indicating that the contents of a register or memory must not change to prevent side effects.

We have tested the circuit analyzer with several examples. One of them is PRIPS, a coprocessor with a RISC-like instruction set, which provides data types and instructions supporting the execution of Prolog programs. The structure consists of 50 register transfer modules. A complete circuit analysis took 77 seconds leading to 1/2 MB of facts.

5. Code Generation

A microcode generator is a tool for mapping algorithms to predefined hardware structures, by generating the required binary code. If such a compiler is target independent, i.e. the programmable microprocessor is an input of the compiler, we call this method retargetable compilation [6, 23, 22]. The original intention for this work is to generate self-test microcode, i.e. microcode that is able to perform a test for programmable microprocessors. The following example describes code generation for a variable assignment, called load instruction.

Assuming the assignment reg[0] := 1 to be generated as used as first instruction of the simulation example, i.e. we want to load register 0 of the register file with 1. The binary values are [0,0,0,0] for the address and [0,0,0,1] for the data to be loaded. After a justification step has driven necessary values for the load instruction to the inputs of the register file, the following three values have to be generated:

address = [0,0,0,0]; data = [0,0,0,1]; c(4:4) = [0];

Having unified the input ports of the register file with these values, we have to perform a backward simulation to search for modules which are able to yield the constants. This module is usually the programmable instruction memory or a decoder. Backward simulation in general is non-deterministic and therefore backtracking and bidirectionality of Prolog is advantageous.

In our example, the control code c(4:4) and the address = c(3:0) are direct predecessors of the instruction memory. More difficult is to have the data loaded, because we have to pass the value [0,0,0,1] through certain modules. However, with the use of the *path/3* facts generated before, the problem is easy to solve. The *path/3* fact shown in table 4.1 gives all information to generate a solution for the required data transfer. Table 5.1 shows the resulting binary code. If this instruction is part of a complete microprogram, additional tasks could be done concurrently. The address for the next instruction has to be determined which could be done by incrementing the program counter by c(8:7) = [0,0]. Alternativly a jump or a conditional jump could be performed, leading to values for the 6-bit jump address c(14:9). Therefore the facts *jump/1* and *conditionalJump/2* are used, whereas *incrementPC/2* is used to increment the program counter.

Bits	19	18:15	14:9	8:7	6:5	4	3:0
Code	0	0001	XXXXXX	00	01	0	0000

TABLE 5.1. Binary code for reg[0] := 1

At the end of code generation the microprogram has to be bound to real addresses of the instruction memory. We perform global scheduling while concurrently compacting and binding the code. Here we make extensive use of linear constraints over the integer domain. In this way it is possible to exploit the parallelism of the target processor (e.g. in VLIW architectures). At the beginning we unify the symbolic address of the first instruction with the start address e.g. 0. Delayed goals like *incr/2* are woken and this leads to a successive binding of concerned addresses. The process is supported by a labelling procedure. The resulting microprogram can be simulated by the simulator described above. We would like to extend the code generator to handle pipelined architectures.

6. Experimental Results

The tools described above have been applied to several target structures. Table 6.1 gives information about the example circuits: simplecpu, as shown in section 2, demo [2], prips [1] and mano [18]. The number of RTL components, the width of the microinstruction controller and the width of the datapath are given. The results shown here indicate that the tools can be applied even to realistic structures. All times are measured on a SPARC 10 workstation.

circuit	RTL modules	instruction width	datapath width
simplecpu	10	20	4
demo	16	84	16
prips	50	83	32
mano	21	50	16

TABLE 6.1. Example Circuits

The times shown in table 6.2 are achieved by simulating a simple loop, such as the program mentioned in section 3. Every event means simulation of a complete behavior tree (RT-events). The original MIMOLA simulator (written in Pascal for an earlier, more restricted version of MIMOLA) simulates on average about 300 RT-events/sec. Although the Prolog simulator is slower by a factor of 3 to 5, its main advantages are the support for backward and symbolic simulation. These features are important for test generation, e.g. to justify signals.

The results shown in table 6.3 are measured for the microcode preparation phase of section 4.2. We can see that for larger circuits a lot of facts are generated by the circuit analyzer. Therefore, the given hardware has been analyzed and microoperations which can be executed by the hardware have been extracted.

circuit	events	CPU sec	events/sec
simplecpu	149	0.83	179.5
demo	1394	25.05	55.6
prips	1003	21.99	45.6
mano	478	4.3	111.1

TABLE 6.2. Simulation CPU times

circuit	generated facts	CPU sec
simplecpu	26	0.56
demo	61	2.96
prips	415	77.03
mano	131	11.85

TABLE 6.3. Circuit Analysis Times

7. Conclusion

We have described how logic programming and coroutining can be exploited for some tools in the MIMOLA hardware design system. A simulator for structural hardware models, described in a hardware description language, has been presented. The simulator consists of 2700 lines of code whereas the original Pascal simulator has about four times more lines of code. Most of the new simulator can be used bi-directionally and symbolically which is very important for code and test generation. Using coroutining to express certain constraints, many backtracking steps can be avoided. The circuit analyzer consists of 2200 lines of code whereras a compareable C++ implementation [16] has about 10000 lines. The circuit analyzer cooperates with a retargetable self-test program compiler [4].

The original simulator was very difficult to maintain. The time to develop VLSI tools using logic programming is much shorter than for imperative languages. On the other hand, software written in standard Prolog is slower. With the new concept of constraint logic programming [3] this disadvantage becomes smaller, because this technique leads to a significant reduction of unnecessary backtracking steps.

Additionally, a tool to generate schematics for structural hardware models has been implemented in Prolog.

This work was supported by the DFG, the German research foundation.

REFERENCES

1. C. Albrecht, S. Bashford, P. Marwedel, A. Neumann, W. Schenk. The Design of the PRIPS Microprocessor, 4th EUROCHIP-Workshop on VLSI Training, Toledo - Spain, September 1993, pp. 254-259.
2. S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer. The MIMOLA Language - Version 4.1, Technical Report, Computer Science Dpt., University of Dortmund, September 1994.

3. F. Benhamou, A. Colmerauer (editors). Constraint Logic Programming: Selected Research, Cambridge, MA: MIT Press, 1993.

4. R. Beckmann, U. Bieker, I. Markhof. Application of Constraint Logic Programming for VLSI CAD Tools, First Int. Conference Constraints in Computational Logic, Munich, September 1994, LNCS 845, pp. 183-200.

5. U. Bieker. On the Semantics of the TREEMOLA Language Version 4.0. Report No. 435, Computer Science Dpt., University of Dortmund, 1992.

6. U. Bieker, P. Marwedel. Retargetable Self-Test Program Generation Using Constraint Logic Programming. 32nd Design Automation Conference, San Francisco, June 1995.

7. R. Beckmann, W. Schenk, D. Pusch, R. Joehnk. The TREEMOLA Language Reference Manual, Version 4.0. Report No. 391, 2nd Edition, Computer Science Dpt., University of Dortmund, 1991.

8. W. F. Clocksin. Logic Programming and Digital Circuit Analysis, The Journal of Logic Programming, March 1987, pp. 59-82.

9. G. Cheng, C. Tsui, I. Pyo, I. Huang, Y. Koh, C. Su, S. Liu, K. Pan, S. Wu, H. Chen, A. Despain. A Full-Range Design Automation System for Instruction Set Processors. First International Conference on the Practical Applications of Prolog, London, April 1992.

10. M. Dincbas, H. Simonis, P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. Journal of Logic Programming, August 1990, pp. 75-93.

11. ECLIPSE 3.4 User Manual, ECRC Common Logic Programming System. ECRC GmbH, Arabellastr. 17, Munich, Germany, 1994.

12. E. Gullichsen. Heuristic circuit simulation using PROLOG. North-Holland, Integration, the VLSI-Journal, No. 3, 1985, pp. 283-318.

13. P. W. Horstmann. Automation of the Design for Testability Using Logic Programming, Dissertation, University of Missouri, October 1983.

14. I. Huang, A. M. Despain. High Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers, 29th Design Automation Conference, 1992.

15. Design Automation Standards Subcommittee of the IEEE. Draft standard VHDL language reference manual, IEEE Standards Department, 1992.

16. R. Leupers, M. Marwedel. Instruction Set Extraction from Programmable Structures. EURO-DAC, Grenoble, September 1994.

17. Y. Lichtenstein, B. Welham, A. Gupta. Time Representation in Prolog Circuit Modelling. 3rd Annual Conference on Logic Programming, Edingburgh, Springer, 1991, pp. 78-93.

18. M. Morris Mano. Computer System Architecture, Prentice-Hall Int., Inc., Third Edition, 1993.

19. P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors, Proc. 21st Design Automation Conference, 1984, pp. 587-593.

20. P. Marwedel. Matching system and component behavior in MIMOLA synthesis tools. Proc. EDAC 1990, 1990.

21. M. D. Neill, D. D. Jani, C. H. Cho, J. R. Armstrong. BTG: A Behavioral Test Generator, Computer Hardware Description Languages and their Applications, Proceedings of the IFIP WG 10.2 Ninth Int. Symposium on Computer Hardware Description Languages and their Applications, Washington, USA, June 1989, pp. 347-360.

22. L. Nowak, P. Marwedel. Verification of Hardware Descriptions by Retargetable Code Generation. 26th Design Automation Conference, Las Vegas, June 1989, pp. 441-447.

23. L. Nowak. Graph based retargetable microcode compilation in the MIMOLA design system. 20th Annual Workshop on Microprogramming (Micro-20), 1987, pp. 126-132.

24. P. B. Reintjes. A Setof Tools for VHDL Design. Logic Programming, Proc. of the eigth Int. Conference, 1991, pp. 549-562.

25. H. Simonis. Test Generation using the Constraint Logic Programming Language CHIP, In Proceedings of the 6th International Conference on Logic Programming, Lisboa, Portugal, June 1989, pp. 101-112.

26. H. Simonis, N. Nguyen, M. Dincbas. Verification of Digital Circuits Using CHIP. The Fusion of Hardware Design and Verification (Ed. G. J. Milne). North-Holland, 1988, pp. 421-442.

27. D. Svanaes, E. J. Aas. Test generation through logic programming, North-Holland, INTEGRATION, the VLSI journal, February 1984, pp. 49-67.