Application of Constraint Logic Programming for VLSI CAD Tools

Renate Beckmann, Ulrich Bieker and *Ingolf Markhof*

University of Dortmund
Department of Computer Science
D-44221 Dortmund, Germany
FAX: +49 231 755-6116

beckmann@ls12.informatik.uni-dortmund.de / +49 231 755-6124
bieker@ls12.informatik.uni-dortmund.de / +49 231 755-6113
markhof@ls12.informatik.uni-dortmund.de / +49 231 755-6142

Abstract: This paper describes the application of CLP (constraint logic programming) to several digital circuit design problems. It is shown that logic programming together with efficient constraint propagation techniques is an adequate programming environment for complex real world problems like high level synthesis, simulation, code generation, and memory synthesis. Different types of constraints - Boolean, integer, symbolic, structural, and type binding ones - are used to express relations between the components of a digital circuit and efficient propagation is achieved by the coroutining mechanism. To deal with the increasing complexity of digital circuits we use HDL's (hardware description languages) to represent structure and behaviour of circuits.

I Issues

This paper describes the successful use of logic programming extended by constraints to solve complex real world problems in the area of digital circuit design: high level synthesis, simulation, code generation, and memory synthesis. Instead of other systems in this area ours are able to work on a high level of abstraction and to deal with hardware description languages (HDLs).

II Results

It has been shown that logic programming extended by constraints is an adequate mechanism to solve problems in the area of digital circuit design. Working systems for synthesis, simulation, test, code generation, and memory synthesis have been implemented in ECLIPSE [5]. Boolean, linear, structural, and type binding constraints are used to restrict the design space. So unneccessary backtracking as performed in PROLOG could be avoided. Using the combination of logic computation and constraints leads to new concepts to solve these kinds of problems. Some examples of effective constraints are given. Results are considered in the context of runtime, length of code, and design space restriction.

III Significance and Relevance to the Conference

An important aspect for developing a new programming paradigm is to demonstrate its advantages and disadvantages. The only possibility to do this is by studying applications of the paradigm. Preferred application areas for the new style have to be found and tested to close the gap between the theoretical framework and potential applications.

So this paper describes practical applications of combining computational logics and constraints. Several types of constraints - Boolean, linear, structural and type binding constraints - are used to develop working systems for complex real world problems in the area of digital circuit design. It is shown that the combination of logic programming and constraints gets practical relevance even for large software products.

1 Introduction

1.1 Motivation

Starting with a specification of a circuit to be designed, the task of VLSI design is to create a set of documents that can be given to a manufacturing site to produce a chip that will realize the initial specification. Due to complexity, the design process is divided into a sequence of subtasks on different levels of abstraction. Solving these subtaks is not feasible without CAD tools. CAD for electronic circuits (ECAD) covers a wide range of applications like synthesis, simulation, verification, test pattern generation, microcode generation, placement, routing, etc. All of these application domains are very complex: The inherent complexity of the application domains which often contain NP-hard problems (e.g. scheduling with resource constraints) and the large number of design objects and possible design decisions result in a huge problem space.

In the traditional approach to VLSI design complexity consisted in separating related subproblems and solving them sequentially. Solving a set of interdependent tasks requires accurate handling of constraints, because each (partial) solution of one task constraints the set of solutions for dependent tasks. But due to the lack of support of constraint handling in most programming languages, a lot of constraints are just ignored or handling of constraints is hidden in complex algorithms and data structures.

As ECAD software must also adapted continuously to the ongoing progress in manufacturing technology, writing and maintaining ECAD software is difficult and costly. This is a problem even for larger companies (e.g. Mentor) and of course this is even more true in the field of research, where the realisation of new ideas is delayed due cumbersome implementation.

We are thus following the constraint logic programming approach to VLSI design. Programming in PROLOG can be done on a high level of abstraction. Rapid prototyping is easy and the resulting programs are pretty short. PROLOG shortens the development time of new software and also simplifies software maintenance. Moreover, the ability to use clauses bidirectionally is especially useful in the area of VLSI design. The same is true for the built-in backtracking mechanism, because solving complex problems usually involves some kind of search. But due to the large search space pure backtracking is not sufficient. Search has to be guided by constraints. This directly leads to constraint logic programming (CLP).

We use CLP to restrict the domain of variables denoting the components of a technical system instead of guessing their values. This is done by delaying clauses until necessary preconditions (e.g. the instantiation of some variables) are given. Therefore the search space is restricted more and more until it is small enough to be explored by simple search.

For these reasons we decided to develop ECAD software (synthesis, simulation, and code generation), commonly implemented by imperative languages, and a new tool in this area, memory synthesis with CLP languages. We show that, in opposite to pure logic programming, a significant amount of backtracking can be avoided using CLP.

1.2 Organisation of the Paper

In the following section we describe four different applications. We start with high-level synthesis which enables the user to generate a structure of interconnected circuit components automatically. We continue with the description of a simulator, based on a hardware

description language. Afterwards, a retargetable code generator is described. With this tool it is possible to generate code, like a compiler, for a previously generated register transfer structure. In the last section we describe a memory synthesis tool, which configures a memory hierarchy optimized for a given application specific processor. Each tool description is finished by results to point out advantages of the application of constraint logic programming. At the end of the paper some conclusions are given.

1.3 Related Work

Several approaches to digital circuit design using logic programming have been presented [28], [11], [9], [4], [25], [26], [8], 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 [19], [23], [13]. Especially the work of Helmut Simonis [27] considers CLP as a tool for digital circuit design. Until now, most of the approaches have been applied only for small examples and not for real life problems. Filkorn [31] uses Prolog and boolean unification in a large industrial circuit design application: verification of sequential circuits. Important parts are implemented in C because of restrictions of the used Prolog system, which are not all confirmed by us. The work described here tries to investigate usefulness and efficiency of CLP for very large ECAD software projects.

2 Applications

2.1 High-Level Synthesis

One of the first steps in VLSI design is to describe the system to be designed by a behavioural specification on the algorithmic level, which is usually a program in an imperative hardware description language. The task of high-level synthesis is to find a structure of interconnected components on register-transfer level which realizes the behaviour of the input description. It creates a description of the data-path and a specification of a controller. High-level synthesis can be decomposed into a number of distinct but not independent subtasks, i.e. scheduling, allocation and binding. Solving a set of interdependent tasks requires accurate handling of constraints, because each (partial) solution of one task constraints the set of solutions for dependent tasks.

Related Research. Due to the complexity of the overall transformation, most high-level synthesis systems perform the subtasks of high-level synthesis (more or less) in sequential order, often by utilizing some heuristics [17]. In general, the resulting designs are thus suboptimal. A different, analytical approach to high-level-synthesis is integer programming: The high-level synthesis problem is mapped to a set of variables denoting design decisions whereas design constraints are denoted by linear (in-)equations. In [12], an integer programming approach to high-level-synthesis is presented which subsumes the ideas described in [6], [16] and [24].

While traditional synthesis systems fail to deal with constraints accurately, the drawback of the integer programming approach is that the domain of high-level synthesis must be mapped to a very restricted mathematical model. The formulation of the model is cumbersome even for simple high-level synthesis models and it is probably impossible to represent all aspects of synthesis by linear equations. The resulting models get very complex for enhanced synthesis models which causes serious run-time problems.

We adopted the basic idea of handling the synthesis problem as a constraint satisfaction problem and focus on solving it by constrained branch-and-bound search.

Synthesis of Basic Blocks. A fundamental part of data-path synthesis is the synthesis for basic blocks. A *basic block* is a sequence of assignment statements in the input specification. A basic block can be represented by a directed acyclic graph in which nodes represent operations (e.g. +, *, shift) and edges the data-flow between these operations. This data-flow graph also reveals a partial order between operations due to data-dependencies: If there is a path from one node to another, the operation of the first node has to be finished before the operation of the second node may start.

Scheduling of operations, i.e. assigning them to a sequence of control-steps, is one subtask of high-level synthesis. Scheduling is a very common problem which can easily be mapped to CLP by using finite domains [8]: The start time CS*i* (control step) of an operation *i* is represented by a domain variable CS*i*::1..Cmax[1] and precedence between operations is declared by constraints CS1+CT1 #<= CS2[2], where CT1 is the number of control steps that are needed to perform the first operation (computation time). Resource constraints are introduced by restricting the number of functional units, e.g. there may be only one adder available. This can be represented by a disjunctive predicate stating that for two operations of the same type, either CS1+CT1 #<= CS2 or CS2+CT2 #<= CS1 must hold. Assuming that the delay is 1 for all operations, the atmost/3[3] predicate [5] can be used to impose resource constraints. If there are N functional units of type T, a list L of all control step variables belonging to operations of the same type is constructed and atmost(N,L,C) for C=1..Cmax is called. This idea can be extended to the more general case of realistic delays and libraries.

However, scheduling is only one subproblem in high-level synthesis. Allocation is another. While the number of resources and the duration time of the tasks to be scheduled are known a priori in common scheduling problems, this is not true for high-level synthesis. Here, resources, i.e. functional units, are described by a library. Each functional unit type is described by a set of attributes, e.g. the operations that an instance of this functional unit type may perform, the execution time for every operations, and its latency if it is a pipelined unit. It should be noted that there is no 1-1 mapping between the set of operations and the set of functional unit types. There may be several types of adders to implement an add operation and ALUs can execute different operations. The task of allocation is to allocate a minimal set of functional units sufficient to perform all operations in the data-flow graph.

The goal of scheduling is to minimize the number of control steps, the goal of allocation is to minimize the total area of all instanciated functional units. Hence, both tasks are interdependent and there is a trade-off between both goals. We use a set of constraints to propagate design decisions made in one task to the other task during a branch-and-bound search to solve both tasks. The search is guarded by a cost term, which is the weighted sum of the number of control steps and the total area of all instanciated functional units. In the following, we focus on the scheduling task.

---

1. *Var* :: {1..Cmax} denotes the constraint *Var*∈{1..Cmax}.
2. *Term1 #rel Term2* introduces the constraint *Term1 rel Term2*, where *rel* is <, <=, =, etc.
3. atmost(*N*,*L*,*Val*) denotes the constraint that no more than *N* values in list *L* may be equal to *Val*.

Propagation of Scheduling Decisions to Allocation. For each operation *i* in the data-flow, we declare three domain variables, CS*i*, FT*i* and CT*i*: Again, CS*i*::1..Cmax denotes the control step in which operation *i* is started. FT*i* is an index to the set of all functional unit types that can execute operation *i* and is used to denote the binding of the operation to a functional unit type (type-binding). When searching the library for candidates that may be used to implement an operation, a list CTList of all corresponding computation times of these functional units is also constructed. This list is then used to constrain the computation time of the operation and to relate it to the type-binding by *element*(FT,CTList,CT)[1].

As described above, precedence constraints due to data-dependencies between operations are now introduced, e.g. CS1+CT1 #<= CS2. But CT1 now is a domain variable which is also directly related to the operations type-binding index FT1. Thus, if the domain of CT1 is reduced due to precedence constraints in scheduling, this is also propagated to allocation and a slow functional unit may be discarded.

Because there is no fixed number of functional units for any type, it is impossible to impose a priori resource constraints. Nevertheless, if there are only two adders allocated, no more than two additions can be executed at the same time, unless an additional one is allocated. This is what we do. Of course, the allocation of an additional functional unit increases the cost of the current solution in the branch-and-bound search. The increased costs have to be propagated to the cost term guarding the branch-and-bound search.

For all operations, all functional unit types that may be used to execute the corresponding operation, and for all control steps we declare a binary domain variable which reflects the usage of a functional unit type in a single control step. E.g., if the number of control steps Cmax is 4 and there are two candidate types for the first operation with computation time of 2 and 1, respectively, we impose the constraints:

```
element(I1,[1,0,0,0, 0,0,0,0],O1F1U1),    element(I1,[0,0,0,0, 1,0,0,0],O1F2U1),
element(I1,[1,1,0,0, 0,0,0,0],O1F1U2),    element(I1,[0,0,0,0, 0,1,0,0],O1F2U2),
element(I1,[0,1,1,0, 0,0,0,0],O1F1U3),    element(I1,[0,0,0,0, 0,0,1,0],O1F2U3),
element(I1,[0,0,1,0, 0,0,0,0],O1F1U4),    element(I1,[0,0,0,0, 0,0,0,1],O1F2U4).
```

For the index I1, we call I1 #= (FT1*Cmax-Cmax+CS1), whereas FT is the type-binding, and CS1 the control step of the first operation. The 0/1 pattern in these constrains are used to model the computation times of the functional unit types. As the first unit type needs two control steps to perform the operation, O1F1U2 is 1 iff the operation is bound to this type (FT1=1) either in control step 1 or 2 (CS1=1 or CS1=2).

For each functional unit type and each control step, we also create a weighted sum of all usages for this type. Finally, there is also one domain variable for each functional unit type denoting the total number of used units of this type. This variable is constraint to be greater than or equal to the according sums of all control steps. The totals for all functional unit types, weighted by the type's area, contribute to the cost term guarding the branch and bound search.

Once a solution is found, the system searches for another one with reduced costs. Scheduling decisions are propagated to the cost term. Thus, the search space is pruned by abandoning any partial solution with increased costs. As the number of different operation types is usually small with respect to the number of operations (e.g. the fifth order wave filter [29]

1. element(*I*,*L*,*V*) denotes the constraint that *V* is the *I*'th element of List *L* [5].

contains 34 operations, but only two different operation types: Addition and multiplication) and only a few functional units are instanciated at all, pruning may happen just after a few scheduling decisions.

Results. To test the applicability of constraint logic programming in the domain of high-level-synthesis, we first implemented an experimental program for a simplified scheduling problem with resource constraints (unit delay, 1-1 mapping between operation types and the types of functional units) by using the ECLIPSE system. The implemented program consists of less than 200 lines of source code, whereas 2/3 of the program text handles input and output only; thus the code for the scheduling itself is really short. The program was successfully applied to the fifth order wave filter [29].

Currently, we are developing a more enhanced system which does scheduling and allocation simultaneously. The scheduling part, as described above is already implemented. Scheduling alone can be done optimally, but combined scheduling and allocation still sometimes causes high runtimes. This is, because the allocation is deduced only indirectly from scheduling. A subsystem imposing constraints on allocation directly is under development.

Our first experiences in application of CLP to high-level synthesis support the claims of this new programming paradigm with respect to software technology (cf. Section 1.1). But the most important aspect of CLP is that each design decision is immediately propagated by reducing the domains of all related domain variables. In contrast to traditional approaches to high-level synthesis, the CLP paradigm also supports propagation of design decisions between distinct subproblems such as scheduling and allocation. Finally, it is easy to allow user interaction or to include a priori design decisions made by the user.

2.2 Simulation

Using ECLIPSE, an event driven simulator for a hardware description language like VHDL [7] or MIMOLA (**m**achine **i**ndependent **m**icr**o**programming **la**nguage) [2], [15] has been implemented. The simulator is able to simulate a processor together with a given program and is based on three levels of abstraction: the built-in operators, an interpreter for the behaviour of single components and an event driven simulator for circuits together with microcode. Especially for the implementation of the operators, we made extensive use of the coroutining concept of the ECLIPSE language. Due to the lack of space we do not consider the interpretation level and the event driven simulation as described in [3].

For the interpretation of a HDL an implementation of its built-in operators is necessary, which range from logic primitives to complex arithmetic operators. These are represented as Prolog predicates, which mainly have to fulfil the following demands:

a) The operators must work bidirectionally, to be used simulating a circuit from the inputs to the outputs as well as vice versa, i.e. for backward simulation.
b) They should work deterministically, i.e. subsequent backtracking steps must not produce the same solution. This is especially important for the backward simulation, as the mapping of an operator is not necessarily definitely reversible. Certain backtracking alternatives have to be pruned to avoid duplicate solutions.
c) The computation must be - at least on 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. By this, 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 occurring in one of these is bound, either to a value or another variable, the goal will be woken, and the delay conditions are checked again.

Nevertheless, at the end of a simulation the set of all delayed constraints must be consistent, i.e. there should be a constraint solver which finds contradictions and - if possible - solutions for variable bindings. It would be sufficient to consider a minimal complete set of operators, but for efficiency we used a set containing AND, OR, XOR and NOT. The Prolog code for those operators is now divided into delay clauses and program clauses. To get an impression of the effectiveness of Boolean constraints, we consider a simple or/3 clause (fig. 1).

```
delay or(X, Y, Z) if var(X), var(Y), var(Z), X \== Y.
delay or(X, Y ,Z) if var(X), var(Y), Z == 1, X \== Y.

or(X, Y, Z) :- nonvar(Y), !,or1(Y, X, Z).
or(X, Y, Z):- nonvar(X), !, or1(X, Y, Z).
or (X, X, X).

or1(1, _, 1).
or1(0, X, X).
```

Fig. 1. or/3 implemented as a boolean constraint

The delay clauses cover the case, when the two input parameters are distinct unbound variables, and the output parameter is either unbound or 1. 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 OR: the first two of them deal with the case, when one of the inputs is bound, and call *or1* with this one in the first place. For the third clause there are - due to the delay clauses - only two possibilities left: either the output is 0, 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 *or1* expects its first input to be instantiated. If it is bound to 1 the result must be 1 either. If it is 0 the output is identical to the second input.

The more complex operators are now based on these four logical primitives (AND, OR, XOR and NOT) and about 50 different operators have been implemented. Of course the set of operators is not restricted to single bit operations. For each of them there is also a version for bit strings which are represented as lists. On top of these there are built-in arithmetic operators like addition, multiplication and string manipulation operators like shift and concatenate. Supposed, a half adder (consisting of an and/3 and a xor/3 gate) has already been defined, we consider increment/2 as an example for a more complex operator (fig. 2).

E.g. the goal incr([A, B], C) yields the following set of constraints:

{C = [Y, Y1], not(B,Y1), and(A, B, _Carry), xor(A, B, Y) }

Such operators can be used bidirectionally, symbolically and deterministically. If necessary, remaining variables can be bound by a labelling procedure. The event driven simulator mentioned above does not instantiate remaining variables but generates a set of constraints instead of making random guesses.

```
incr (In, Out) :- incr(In, Out, _Carry).

incr ([X], [Y], X) :- !, not(X, Y).

incr ([X | RestX], [Y | RestY ], Cout) :-
    incr(RestX, RestY , Cin),
    halfadd(X, Cin, Y, Cout).
```

Fig. 2. Example of an increment for a bit string, e.g. incr([0,0,0,0], [0,0,0,1]) is true.

Results. The event driven simulator mentioned above has been applied to several target structures described by hardware description languages. Table 1 gives informations about some example circuits: simplecpu [3], demo [2], prips [1] and mano [14]. The number of RTL components and the width of the microinstruction controller are given. The results shown here indicate that the tools can be applied to realistic structures. All times are measured on a SPARC 10 workstation and are achieved by simulating a simple loop. Note that

circuit	RTL modules	instruction width	events	CPU sec	events/sec
simplecpu	10	20	149	0.83	179.5
demo	16	84	1394	25.05	55.6
prips	50	83	1003	21.99	45.6
mano	21	50	478	4.3	111.1

Table 1: Example Circuits

every event means the simulation of a complete register transfer level component. The implementation of the whole simulator in ECLIPSE consists of 2700 lines of code whereas a Pascal implementation has about four times as many. Most of the ECLIPSE implementation can be used bidirectionally and symbolically which is very important for code and test generation.

2.3 Retargetable Code Generation

In this section we want to give an example for the application of structural constraints which we understand as constraints forcing variables to get bound to specific structures, e.g. trees or lists. We use structural constraints e.g. to simplify some problems in a tool called retargetable code generator. A retargetable code generator is a tool for mapping algorithms to predefined programmable structures by generating the required binary code. Fig. 3 gives an overview of the retargetable code generation process. We start with a programmable microprocessor, described by a computer hardware description language. This description may be generated by a synthesis tool as described above. The hardware description of a microprocessor serves as input of a retargetable compiler which generates binary code for a given program. Such a retargetable compiler is necessary since typical design processes require several iterations resulting in slightly changed architectures. Instead of writing target-specific compilers, we propose using target-independent compilers to allow the user to implement a procedural behaviour on a programmable register transfer structure without changing the compiler.

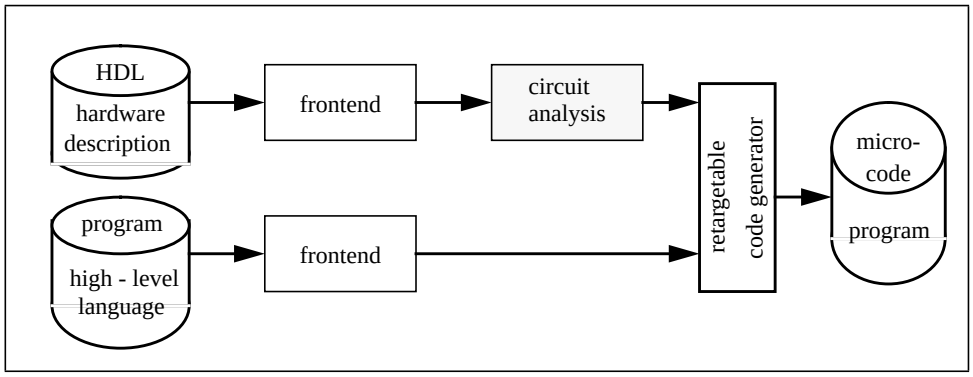

Fig. 3. Retargetable Code Generation

We describe a subtask of the circuit analysis (see Fig. 3, shaded box) phase. The capabilities of a microprocessor, i.e. the instructions which can be performed by a structure, can be extracted by a preceding circuit analysis phase. A retargetable code generator typically has to generate conditional jumps for a given programmable hardware structure, that is to translate a conditional jump (2 way jump) into a control word, i.e. a bit string that forces the hardware to perform a conditional jump as follows:

IF condition THEN increment program counter ELSE jump to label;

Figure 4 shows a part of a typical programmable microcoded controller of a processor. The

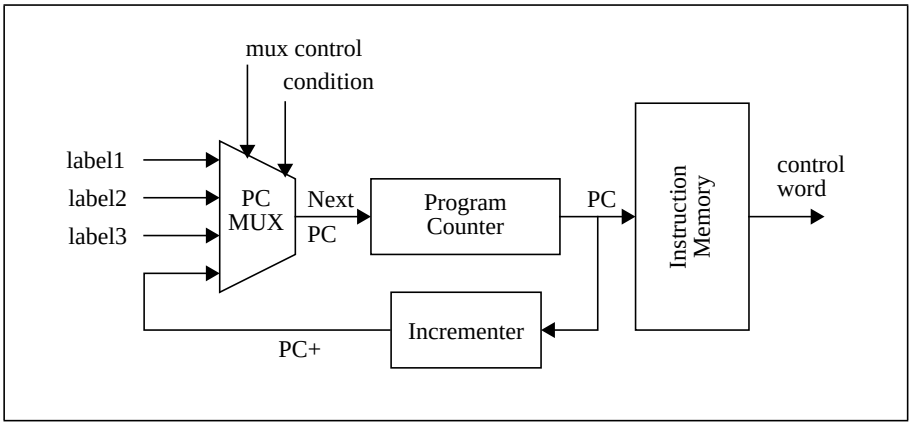

Fig. 4. A part of a typical programmable microcoded controller

controller consists of a program counter, an instruction memory, an incrementer and a multiplexer. The next state of the program counter is selected by the multiplexer control signals. The controller given above allows 4 way jumps. The behaviour of the multiplexer is given by the hardware description. At the beginning of the compilation phase, we do not know how the multiplexer is specified, because a retargetable code generator is target independent . A multiplexer can be specified by an IF or CASE construct as shown by the following equivalent statements:

a) IF Condition THEN S1 ELSE S0;
b) CASE Condition OF 0: S0; OF 1: S1;

Furthermore, case statements can have different numbers of branches. Starting with a circuit given as a hardware description a tree based Prolog circuit representation is generated by a frontend compiler. Every module consists of a list of connections, a list of storing cells and a *behaviour tree*. A behaviour tree is shown in figure 5 representing a part of the multiplexer shown above. Such a tree is easily represented by a Prolog structure. A non-empty tree is

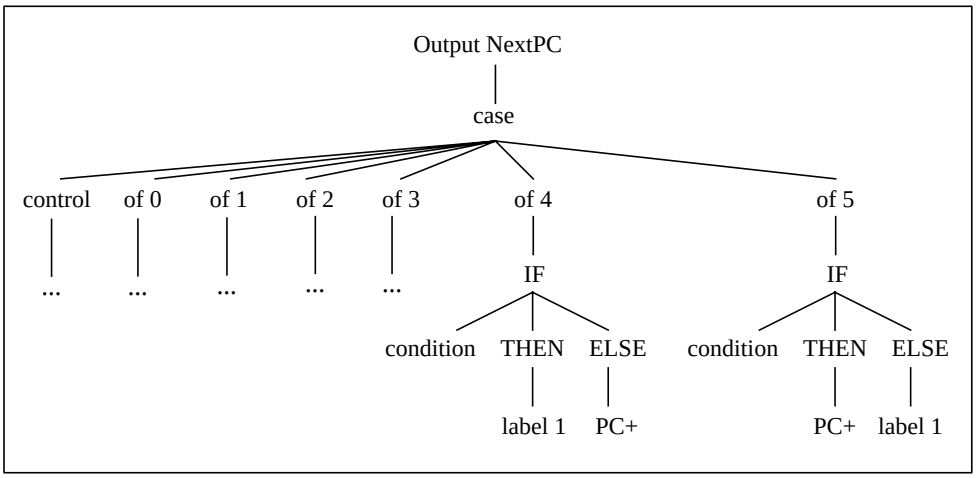

Fig. 5. Behaviour tree of a typical multiplexer

represented as a tupel: a term (called the root element of the tree) and a list of sons of the term (called subtrees). Figure 5 shows IF constructs nested in a CASE construct. The value loaded into the program counter depends on the condition if the multiplexer control input is 4 or 5. Consider the following examples:

a) control = 4 and condition = 1 => next state of the program counter is label 1
b) control = 4 and condition = 0 => next state of the program counter is PC+
c) control = 5 and condition = 1 => next state of the program counter is PC+
d) control = 5 and condition = 0 => next state of the program counter is label 1

Otherwise an unconditional jump (label 1 - 3) or an unconditional increment of the program counter (PC+) is performed (not shown in Fig. 5).

Figure 4 shows only one possible realization of a controller and figure 5 depicts only one possible realization of a multiplexer, but a target independent code generator has to consider different specifications of the same behaviour. We describe different alternatives by structural constraints. This will be shown by a simple example.

To compile a conditional jump we have to search for a multiplexer starting at the program counter and backwards through the circuit. This search can be accelerated significantly if unnecessary backtracking is avoided. Using coroutining we search for alternative specifications in parallel. Consider the multiplexer/2 clause shown in figure 6. The recursive search is able to find a candidate multiplexer construct with root term *if* or *case* and a list of sons of different lengths. The three multiplexer/2 facts given above show simplified alternatives

```
delay multiplexer(Operator, OperatorSons) if var(Operator), var(OperatorSons).

multiplexer(if, [Condition, Expression, then, ThenBranch, else, ElseBranch]).
multiplexer(case, [Control, Expression, of, S0, of, S1]).
multiplexer(case, [Control, Expression, of, S0, of, S1, of, S3, of, S4]).
```

Fig. 6. Structural constraints

to specify a multiplexer. Fact number 3 denotes a case construct with 4 branches whereas the first two facts can be used to realize 2 way jumps.

Such alternative specifications of the same behaviour are considered by transformation rules. Transformation rules can be applied to replace certain high-level language elements by equivalent statements. E.g. a statement *x := y+1* can be replaced by *x := increment(y)*. Even loops can be transformed (fig. 7). Structural constraints are used to implement such transformation rules.

```
<label>: REPEAT <block> UNTIL <condition>

(* can be transformed to: *)

<label>: <block>
ProgramCounter := IF <condition>
                        THEN incr(ProgramCounter)
                        ELSE <label>
```

Fig. 7. Loop transformation rule

We want to point out the advantage of structural constraints applied to this problem. At the beginning of the process of retargetable code generation we do not know how the hardware is specified by the user or a synthesis tool. So we have to take different alternative hardware specifications of the same behaviour into account. This is easily handled by structural constraints (implemented by the use of coroutining), because the behaviour of a given hardware component is represented as a tree and transformation rules, also representing alternative structures, are represented as alternative clauses (facts). Therefore it is better to generate constraints which ‘guard’ that a required behaviour is taken into account instead of searching for different alternatives by try and error.

Results. The results shown in table 2 are measured for the circuit analysis phase and the circuits (processors) mentioned in table 1. In the circuit analysis phase the given hardware is analysed and the microoperations which can be executed by the hardware are extracted. A conditional jump as described above is an example for such microoperations which are stored as facts. The implementation of the circuit analysis phase in ECLIPSE consists of 2200 lines of code whereas a comparable C++ implementation [30] has about 10000 lines of code. The results show that for larger circuits a lot of facts are generated by the circuit analyser.

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

Table 2: Circuit Analysis Times

2.4 Memory Synthesis

In the last years execution speed of processors increased much more than memory access speed leading to a gap between processor and memory speed. VLSI designers try to reduce this gap by developing memory hierarchies including small memory modules with short access time (e.g. cache) and larger slower ones (e.g. main memory and secondary memories). The increasing complexity of the memory hierarchy and the introduction of multiprocessor systems with shared data makes memory synthesis more and more complex. Therefore tool support is necessary.

Related Research. Up to now no tool support exists to design a memory hierarchy. The cache was the most examined memory module in the past: A good overview of all its design choices is given in [21]. Some general analysis are done in how to improve access performance [10] by varying some of the cache parameters. Concepts for main memory like interleaving are analyzed in [20]. Due to the needs of multiprocessor systems [22] gives an overview of mechanisms for data consistency. The influence of the memory access behaviour of programs on the memory design is rarely examined. [18] analyses address sequences to calculate an upper bound for necessary memory size. None of them describes methods how to design memory hierarchies optimized for a special application domain running on a special kind of computer system.

The Memory Synthesis Problem. *Memory synthesis* configures a memory hierarchy for some kinds of access sequences by optimizing the time to read/write a date from/into the memory, the required chip area and the cost. The features of these access sequences are highly influenced by the computer architecture the memory is designed for and the application programs. To minimize access time data should mostly be found in the fastest memory module, the cache.

A memory hierarchy can be described as a highly *generic model* expressing all configuration alternatives as *parameters* which has to be adjusted during memory synthesis. Table 3 shows some of the parameters of a cache with different domains.

Parameters	Meaning	parameter domain
size	size of an on-chip cache	1 - 256 K Bytes
line size	line size of an on-chip cache	1 - 256 Bytes
associativity	number of lines that are accessed in parallel to search for an address	1, 2, 4, 8, 32, full associative
replacement	decides which of the parallel accessed lines is replaced when a new data is brought into the cache	random, LRU, FIFO, ...
write	how to update the main memory	write through, copy back

Table 3: Some of the Cache Parameters in the generic memory hierarchy model

Memory Synthesis with Constraint Logic Programming. In memory synthesis only few structural decisions have to be done (e.g. availability of a cache, unified cache or seperarted data and instruction cache). This leads to a moderate number of parameters (less than 200). The huge design space is caused by the combination of the parameters with partly large domains. These domains are restricted by hundreds of relations between the parameters and the features of the architecture and the application domain. Therefore the main task in memory synthesis is to compute all these relations. They are adequately expressed and computed using CLP. Figure 8 shows two relations restricting the design space for caches expressed as constraints (in ECLIPSE notation). The first one *restrict_associativity/4* handles an integer constraint between the associativity and some of the other cache parameters: For data and unified caches the number of lines and/or the associativity must be high enough to avoid high miss rates[1], for instruction caches an associativity of 2 is enough. While it is unknown if the cache is separated or not, this constraint is delayed instead of guessing and backtracking. The second one, *make_powerof2/1* is used to restrict the size and the line size of a cache to powers of two. This constraint is not delayed but the domain of the variable is restricted.[2]

Solving a problem like memory synthesis with constraints may cause two problems:

The first one is an *underconstrained design space*, i.e. after evaluating all given constraints the remaining design space is still too large for a simple search strategy. The remaining parameters of the memory hierarchy must be adjusted by some heuristic strategy not guided by synthesis knowledge. Nevertheless analysis knowledge is available: After adjusting the parameters the memory hierarchy model together with a given address trace is put into a simulator to check if the requirements are fulfilled. If not, the reasons are analysed and one of a few redesign cycles is started with some additional redesign knowledge. After each iteration the results are checked by simulation.

The evaluation by an external trace driven simulator and redesigning is of special importance for memory synthesis because the direct influence of the parameter adjustments on the goals like access time is not completely clear[3].

Due to the optimization criterion given by the user, the clauses *optimization*/2 in Figure 8 describe two simple search strategies to adjust the parameters of a cache in the first design cycle when no analysis knowledge is available from the simulator: To minimize the area of the cache, its size and associativity is minimized by the *min_max/2* predicate. To minimize average access time the differences between size, line size, and associativity and their mean values is minimized by the *min_max/2* predicate.

The second problem is an *overconstrained design space*, i.e. the set of constraints is inconsistent. For this reason the constraints are split into hard and soft ones. Hard ones must not be inconsistent but soft ones may be relaxed if they lead to an inconsistency. In figure 8, *restrict_associativity/4* is a hard and *make_powerof2/1* applied to size and line size of a cache is a soft one.

---

1. A *miss* is an access to data not in the cache that must be fetched from main memory
2. *dvar_domain*/2, *dom_intersection/3*, and *var_update/2* are built-in predicates to manipulate domains of domain variables. The well known *labeling/1* predicate uses the built-in predicate *indomain/1* to instantiate a list of variables.
3. E.g., on one hand in a larger cache more of the accessed data is available and has not to be fetched from the slower main memory, on the other hand the larger the cache, the larger the access time for each date.

```
delay restrict_associativity (Content, _, _, _) if var(Content).
restrict_associativity (Content, Size, Linesize, Associativity) :-
    Content\==instruction, % unified or data cache,
    Size*Associativity#>=512*Linesize.
restrict_associativity (Content, _, _, Associativity) :-
    Content==instruction, Associativity#<=2.  % instruction cache.

make_powerof2(X) :- X==1.
make_powerof2(X) :- nonvar(X), X #= 2*X2, make_powerof2(X2).
make_powerof2(X) :- var(X),
    dvar_domain(X, Domx), powerof2_dom(Dom2),
    dom_intersection(Domx, Dom2, NewDomx, _),
    dvar_update(X, NewDomx).
powerof2_dom(Dom) :-
    make_powerof2_list(L), % L=[1,2,4,8, ... , maximal size]
    sorted_list_to_dom(L,Dom).

optimization ([Size,Linesize,Associativity,Replacement,...],Goal) :-
    Goal == area,  % minimize area of caches
    Objective #= Size + Associativity,
    min_max (labeling([Size,Linesize,Associativity,Replacement,...]), [Objective]).
optimization ([Size, Linesize, Associativity, Replacement, ...], Goal) :-
    Goal == accesstime,   % minimize access time
    S#=Size-64*1024, B#=Linesize-64, A#=Associativity-8,
    abs(S, S_abs), abs(B, B_abs), abs(A, A_abs),
    Objective = S_avs + B_abs + 8*A_abs,
    min_max (labeling([Size, Linesize, Associativity, Replacement, ...]), [Objective]).
```

Fig. 8. Constraints restricting the cache design space and a simple search strategy

This leads to the concept of a memory synthesis tool roughly shown in figure 9. Relevant informations about the architecture of the underlying computer system and the application domain are used to predict the memory access behaviour. The memory synthesis tool configures a suitable memory hierarchy by taking the given goals into account. The simulator then checks the goals for a given address sequence of the application domain and the configurated memory. If they are fulfilled the memory configuration is given to the user for further work but if they are not fulfilled or if the set of constraints is inconsistent a redesign cycle is started by adding or relaxing some of the soft constraints.

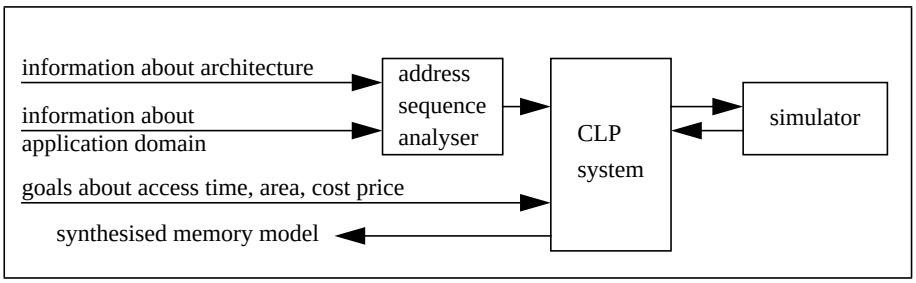

Fig. 9. Rough concept of the memory synthesis tool

Results. For the memory synthesis all constraints between the parameters of the caches are implemented. Due to the parameter domains of the on-chip cache only (without any consistency problems) the design space contains $0.5*10^{20}$ different designs. After evaluating the constraints between the on-chip cache parameters itself (about 50) the number of possible designs decreases to 10^7 and about 30 constraints are delayed, most of them because size and line size of the caches are not enough restricted yet. Just these cache parameters are highly dependent on the access behaviour of the application domain and the underlying computer architecture. It can be expected that they will be strongly restricted after evaluating the according constraints not yet implemented. Assuming that the cache size and line size is instantiated the design space decreases to 430 design possibilities. Not looking on the number of possible values of the parameters but on their domain ranges (difference between maximum and minimum of the domain of each parameter before and after evaluating the constraints and instantiating cache size and line size), we have a restriction to 0.1% of the design space. The search through this very restricted design space is done by heuristic search.

3 Conclusions

We presented applications of constraint logic programming to the area of CAD for VLSI. By the PROLOG-like programming style, implementation can be done at a high level of abstraction, resulting in short programs that can be easily maintained. This is important, because ECAD software must be adopted to new technologies frequently. Using CLP to express certain constraints, a significant amount of backtracking can be avoided. On the other hand, software written in standard Prolog is slower, but with the new concept of constraint logic programming this disadvantage becomes smaller, because this technique leads to a significant reduction of backtracking.

References

1. C. Albrecht, S. Bashford, P. Marwedel, A. Neumann, W. Schenk. The design of the PRIPS microprocessor, 4th EUROCHIP-Workshop on VLSI Training, 1993.
2. R. Beckmann, P. Marwedel, W. Schenk, and R. Jöhnk. The MIMOLA Language Reference Manual - Version 4.0. Research Report 401, Computer Science Dpt., University of Dortmund, February 1991.
3. Ulrich Bieker, Andreas Neumann. Using Logic Programming and Coroutining for electronic CAD. Second International Conference on the Practical Applications of Prolog, London, April 1994.
4. W. F. Clocksin. Logic Programming and Digital Circuit Analysis. The Journal of Logic Programming, pp. 59 - 82, March 1987.
5. European Computer-Industry Research Center, Munich. ECLiPSe - ECRC Common Logic Programming System, 3.4 edition, January 1994.
6. Catherine H. Gebotys and Mohamed I. Elmasry. Simultaneous scheduling and allocation for cost constrained architectural synthesis. 28th Design Automation Conference, pages 2–7, 1991.
7. Design Automation Standards Subcommittee of the IEEE. Draft standard VHDL language reference manual. IEEE Standards Department, 1992.

8. M. Dincbas, H. Simonis, and P. van Hentenryck. Solving Large Combinatorical Problems in Logic Programming. In: Journal of Logic Programming, 1990:8, pp. 75-93.
9. E. Gullichsen. Heuristic circuit simulation using PROLOG. North-Holland, Integration, the VLSI-Journal, No. 3, pp. 283 - 318, 1985.
10. J. Hennessy, D. Paterson: Computer Architecture. A Quantitative Approach, Morgan Kaufman, 1990.
11. P. W. Horstmann. Automation of the Design for Testability Using Logic Programming. Dissertation, University of Missouri, 1983.
12. Birger Landwehr and Peter Marwedel. Oscar: Optimum simultaneous scheduling, allocation and resource binding based on integer programming. Technical report #484, University of Dortmund, Departement of Computer Science. Also to appear in: Proceedings of the European Design Automation Conference (EURO-DAC), 1994.
13. Y. Lichtenstein, B. Welham, A. Gupta. Time Representation in Prolog Circuit Modelling. 3rd UK Annual Conference on Logic Programming, Edinburgh 1991.
14. M. Morris Mano. Computer System Architecture. Prentice-Hall International, Inc., Third Edition, 1993.
15. P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors, Proc. 21st Design Automation Conference, pp. 587 - 593, 1984.
16. Peter Marwedel. Matching system component behaviour in MIMOLA synthesis tools. In Proceedings of the European Design Automation Conference, 1990.
17. Michael C. McFarland, Alice C. Parker, and Raul Camposano. The high-level synthesis of digital systems. Proceedings of the IEEE, 78(2):301–318, February 1990.
18. G. McNiven, E. Davidson. Analysis of Memory Referencing Behaviour for Design of Local Memories, Computer Architecture News, May 88.
19. M. D. O. 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 International Symposium on Computer Hardware Description Languages and their Applications, Washington, DC, USA, pp. 347 - 360, June 1989.
20. B.R. Rau. Pseudo-Randomly Interleaved Memory, Computer Architecture, May 1991.
21. A.J. Smith. Cache Memories, Computing Surveys, Sept. 82.
22. M. Tomasevic, V. Milutinovic. The Cache Coherence Problem in Shared-Memory Multiprocessors. In: Hardware Solutions, IEEE Computer Society Press 1993.
23. P. B. Reintjes. A Set of Tools for VHDL Design. Logic Programming, Proc. of the Eigth Int. Conference, pp 549 - 562, 1991.
24. Minjoong Rim, Rajiv Jain, and Rento De Leone. Optimal allocation and binding in high-level synthesis. Proceedings of the 29th Design Automation Conference, pages 120–123, 1992.
25. H. Simonis, M. Dincbas. Verification of Digital Circuits Using CHIP. The Fusion of Hardware Design and Verification, Elsevier Science Publishers B.V., North-Holland, pp. 421 - 442, 1988.
26. H. Simonis. Test Generation using the Constraint Logic Programming Language CHIP. In Proceedings of the 6th International Conference on Logic Programming, Lisboa, Portugal, pp. 101 - 112, June 1989.

27. H. Simonis. Constraint Logic Programming as a Digital Circuit Design Tool. Dissertation (draft), 1992.
28. D. Svanaes, E. J. Aas. Test generation through logic programming. North-Holland, INTEGRATION, the VLSI journal, No. 2, pp. 49 - 67, 1984.
29. P. DeWilde, E. Deprettere and R. Notua. Parallel and Pipelined VLSI Implementations of Signal Processing Algorithms. In: S.Y. Kung, H.J. Whitehouse and T. Kailath, VLSI and Modern Signal Processing, Prentice Hall, pp 258-264, 1985.
30. R. Leupers. Instruction Set Extraction from Programmable Structures. EURO-DAC, Grenoble, September 1994.
31. Th. Filkorn, R.Schmid, E. Tiden and P. Warkentin. Experiences from a Large Industrial Circuit Design Application. Proceedings of the International Symposium of Logic Programming, San Diego, 1990.