Matching System and Component Behaviour in MIMOLA Synthesis Tools *

Peter Marwedel
University of Dortmund, Informatik XII
D-44221 Dortmund, Germany

e-mail: marwedel@ls12.informatik.uni-dortmund.de

Abstract

This paper discusses the selection of available components during high-level synthesis. We stress the importance of describing the behaviour of available components in some language which is readable for the designer. This behaviour is internally represented by implications. This concept is the key for formal reasoning about the component's capabilities. Alternative functions and sequential and concurrent cooperation of components can be easily described.

1 Introduction

Synthesis can be defined as the process of automatically selecting and interconnecting *components* such that the resulting *system* performs as specified. This definition is applicable to the synthesis of electronic circuits as well as to the synthesis of chemical compounds and other areas, where the term 'synthesis' is used. In this context the term *system* has its general meaning: 'group of things or parts working together in a regular relation' [Hor74].

For high-level (or architectural) synthesis, the specification is given in a behavioural description language such as ISPS, VHDL, VERILOG, MIMOLA or PASCAL. RT-level modules (ALUs, registers, multiplexers, etc.) are typically used as components.

This definition implies that components have to be selected and interconnected according to the required behaviour. Hence, some matching between the required behaviour and the properties of available components is an *essential* feature of any synthesis tool. In fact, this feature is more essential than scheduling, but it has not yet received the same attention.

There are different approaches to component selection. In a number of systems, the synthesis tool starts by generating a system being composed out of a number of virtual components. These virtual components are created during the synthesis process itself and they are described by their performed function. This function corresponds to the functions required by the the specified high-level algorithm. Later, virtual components are bound to or created from available library components.

*Reprint from latex-source. Affiliation updated.

The advantage of this approach is that the virtual components correspond to the 'ideal' components required for implementing the system behaviour. The disadvantage is that this two-level mapping is a possible source of inefficiency. Currently, most researchers have already realized, that synthesis of area-efficient circuits cannot be done in a multi-level approach, not considering area-efficiency right from the beginning. For all design decisions, including scheduling decisions, the effect on the required area should be considered. This is impossible in a strict top-down, multi-level approach. It must be feasible to compute the cost of an implementation at all states of the synthesis procedure. This is only possible, if the cost of components is known to the synthesis algorithm.

Therefore, synthesis algorithms in the MIMOLA hardware design system MSS (or, more precisely, in the MSS2 to distinguish it from the system used by Honeywell [Har89]) assume that a library of available components with known costs exists. This may be either a library of real, predesigned components or just an interface to a set of module generators, informing synthesis tools about the availability of module generators and predicted costs. Hence, synthesis algorithms in the MSS have to describe designs in terms of available components. New modules can only be created if they are described in such terms.

2 Behavioural Descriptions in MIMOLA

2.1 System Behaviour

With respect to the description of the system behaviour, MIMOLA [Jöh89] is an extension of PASCAL. In addition to PASCAL, MIMOLA contains concurrent and parallel blocks, signal assignments (denoted by a left arrow), bit level addressing, and a rich set of operators.

2.2 Component Behaviour

2.2.1 External Representation

It has been one idea of the MSS right from its first implementation [Mar79], to describe the behaviour of the system and of available components in the same language. This contrasts with the approaches taken e.g. by Dutt [Dut89] and Leive [Lei81]. They describe the behaviour of available components in a special language.

Using the same language for the system and the components has a number of advantages. For example, the set of standard operators and their semantics is the same. Furthermore, a single simulator can be used to simulate the behaviour at the system level and at the RT-structure level.

In early versions of MIMOLA [Mar79], the description capabilities for component behaviour were extremely restricted. Many restrictions have been dropped in the meantime. However, certain restrictions still apply, especially to older synthesis tools. Below we will present the most general behavioural descriptions applicable to synthesis (there are no limitations for simulations). Specific tools may accept only a subset of these. We are working towards generalizing the current approach for all tools (see below).

Currently (version 4.0), behavioural views of components are restricted to a single concurrent block of signal assignments and assignments to the local state. For each signal or state variable, there is at most one CASE-statement, describing the operation being performed for a certain control code. Assignments to signals and state variables are assumed to be executed concurrently. Hence,

multiple CASE-statements have to be enclosed within CONBEGIN ... CONEND. A single CASE-statement may be enclosed within BEGIN ... END. Clocking conditions may be specified in an AT ... DO-prefix.

Examples:

```
MODULE  alu ( IN a, b: word; IN c: twobits; OUT f: word);
        BEHAVIOUR AtRtLevel IS
          BEGIN
            f <- CASE c OF
              0 : a + b;
              1 : a - b;
              2 : a OR b;
              3 : a NAND b
            END;
          END;
MODULE  sadd ( IN a, b: word; IN c: twobits; OUT f: word)
        BEHAVIOUR AtRtLevel IS
          BEGIN
            f <- CASE c OF
              0 : SHIFTLL(a);   - -shift left logical
              1 : a + b
              2 : a - b
              3 : SHIFTLL(a + b);   - -composite function
            END;
          END;
MODULE  comp ( IN a: word; OUT f: bit)
        BEHAVIOUR AtRtLevel IS
          BEGIN
            f <- (a = 0);
          END;
MODULE  cc ( IN a: bit; IN c: bit; OUT f: bit)
        VAR state: bit;
        BEHAVIOUR AtRtLevel IS
          CONBEGIN
            CASE c OF
              0 : state := a ;
              1 : NOLOAD;
            END;
            f<- state;
          CONEND;
```

2.2.2 Internal Representation

The external MIMOLA format is first translated into an internal tree format, called TREEMOLA. Synthesis tools then implicitly or explicitly represent component behaviour as a set of implications:

alu (a,b,0,f) ⇒ f=(a + b)
alu (a,b,1,f) ⇒ f=(a - b)
alu (a,b,2,f) ⇒ f=(a OR b)
alu (a,b,3,f) ⇒ f=(a AND b)
sadd (a,b,0,f) ⇒ f=SHIFTLL(a)
sadd (a,b,1,f) ⇒ f=(a + b)
sadd (a,b,2,f) ⇒ f=(a - b)
sadd (a,b,3,f) ⇒ f=SHIFTLL(a + b)

Arguments at the right hand side are *free variables* in the sense of transformational reasoning. At the left hand side of the implication, components are called in the form of a procedure. This call is denoted as *component activation*. Each argument position denotes a signal. In the final structure, these signals have to be implemented as interconnections. Component activations describe relations between input and output signals (for the present, we ignore delay times).

The behaviour at the right hand side can be implemented by the component activations at the left hand sides. A separate analysis performed by all synthesis tools guarantees that selected component activations do not have undesirable side effects (e.g. do not destroy register contents).

Each entry in the above list of implications is called a *function list entry* (FLE). For the sake of simplicity, we have omitted the bitwidth information, which in practice is part of every FLE.

3 Matching System and Component Behaviour

In this section we describe the matching procedures of the MIMOLA synthesis tools, and how they fit into the FLE-framework. In what follows the tools are regarded as examples for the the matching of *simple operations* and *complex operations*. Currently, three synthesis algorithms have been designed for the MSS2.

The matching process of the first tool [Mar86, Mar86a] treats components with simple operations only. The algorithm is based upon a decomposition of the synthesis problem into a number of now classical subproblems (scheduling, module selection, module allocation, register allocation, etc.). For each subproblem, smart optimization algorithms have been designed. But there is a strict sequence of design decisions. Top-level problems (e.g. module selection) are solved first, low-level problems (e.g. the generation of control) are solved last. Hence this algorithm is now called the vertical synthesis algorithm.

The matching with complex operations is implemented in two synthesis tools:

- The second algorithm [Bal89] integrates scheduling and module binding. It makes intensive use of 0/1 integer programming to establish the required bindings between behaviour and structure. Due to the integration of scheduling and module binding, module selection is not a separate step. Rather, it is part of the scheduling phase and uses 0/1 integer programming to select a set of modules that is able to perform the required functions. Furthermore, this algorithm is able to consider tradeoffs between speed and cost of different modules.
- The third synthesis algorithm [Bro89] integrates all subproblems. It is based upon successive transformations of an initial hardware structure. Each transformation tries to improve the current solution. It is therefore called 'successive approximation synthesis system' (SuccASS).

3.1 Preprocessing

In order to simplify the pattern matching procedure, some preprocessing is recommended. This applies to all FLE-applications. The following is a set of useful transformations:

- *operator canonisation:* $\circ$ and $\bullet$ are called *converse* operations iff $\forall$ a,b : a $\circ$ b = b $\bullet$ a. For non-commuting ($\circ \neq \bullet$) converse operations, one of the two is replaced by the other. This is done in all behavioural descriptions. Additional transformations could be implemented, but this has not been done at the time of writing this paper.
- *commutativity:* for commuting operations, constant arguments are always used as right arguments.
- *additional FLEs*: the matching process can be extend by storing a set of valid algebraic properties in a rule base library. These rules can be used to created new valid FLEs.

  For example, assume that a FLE alu(a,b,0,f) $\Rightarrow$ f=(a+b) and a library rule f=(a+a) $\Rightarrow$ f=(a$*$2) exist. From these two implications, it can be deduced that alu(a,a,0,f) $\Rightarrow$ f=(a$*$2). This implication can be stored as an additional FLE. The same mechanism can be used to allow an implementation of multiplications by arithmetic shifts.

  Another example is the implementation of VIAs. A VIA is an identity operation which propagates its input-value to the output of the component. A multiplexer has a VIA operation for each input by default. In order to avoid unnecessary interconnections, it might be desirable to feed a value through e.g. an adder. This can be expressed by a library rule f=v+0 $\Rightarrow$ f=v for the neutral element of the addition, which yields the FLEs alu(v,0,0,f) $\Rightarrow$ f=v, and alu(0,v,0,f) $\Rightarrow$ f=v. Additional FLEs are an easy way of describing choices. Their effect on the complexity of the matching algorithm can be neglected.

Strength reduction, which sometimes is cited as a possible preprocessing step, has its problems. Even if applied to both the system and the component behaviour, it cannot be guaranteed that all possible implementations will be found. For example, consider multiplication by two. Using additional FLEs, we will create entries at components, which are able to multiply, add, or shift. A unique reduction to one of the three operations is not possible.

In the following, we are going to use an abstract data type *expression.* Expressions are described by trees. Each node in the tree represents some operation, constant, or a free variable. Functions $IsOper$, $IsConst$, $IsFree$, $Oper$, $Value$, and $Width$ are defined on these nodes and return the type of a node, the operation or the value it represents and its bitwidth, respectively.

3.2 Matching Simple Operations

Within this section we treat the matching process with simple operations, and how it is used in the vertical synthesis algorithm. The behavioural description of components is restricted to *simple operations*: Each statement in the component declaration may contain only a single operation. Arguments may be input/output signals, constants or the internal state.

Module alu of the previous section performs simple operations only. But module sadd don't, because 'SHIFTLL (a + b)' is a composite operation. Therefore, it is required to implement complex system behaviour by a set of component activations.
Example:
The assignment cc := (SHIFTLL (a+b))=0 has to be covered by matching right hand side of FLEs.

The order in which this cover is computed, does not matter. The following is a sequence of replacements of expressions by component activations (f, g and h denote signals):

```
      cc := (SHIFTLL (a+b))=0
⟶    PARBEGIN alu* (a,b,0,f); cc:=(SHIFTLL(f))=0; PARENDD
⟶    PARBEGIN alu(a,b,0,f); sadd(f,0,0,g); cc:=g=0; PARENDD
⟶    PARBEGIN alu(a,b,0,f); sadd(f,0,0,g); comp(g,h); cc:=h; PARENDD
⟶    PARBEGIN alu(a,b,0,f); sadd(f,0,0,g); comp(g,h); cc(h,0,X); PARENDD
```

The matching of system and component behaviour is defined by the following three functions:

1. Range: expression → boolean.

$$Range(o) := \begin{cases} true & : \text{ if it is semantically legal and desirable to use only a subrange of input arguments if the output is also correspondingly restricted} \\ false & : \text{ otherwise} \end{cases}$$

   The purpose of this function is to allow e.g. 8-bit additions to be implemented by an 16-bit adder but to inhibit the use of subranges, if this would require implementing sign-extensions. Hence, Range is true for additions, but false for signed magnitude comparisons.

2. Arg: expression × expression × operation → boolean.

$$Arg(Sys, fle, op) := \begin{cases} true & : \text{ if } IsConst(fle) \wedge IsConst(Sys) \wedge Value(fle) = Value(Sys) \\ true & : \text{ if } IsFree(fle) \wedge (Width(fle) \geq Width(Sys)) \wedge Range(op) \\ true & : \text{ if } IsFree(fle) \wedge (Width(fle) = Width(Sys)) \wedge \neg Range(op) \\ false & : \text{ otherwise} \end{cases}$$

   The purpose of this function is to check if the argument Sys of an operation op in the system specification matches an argument fle in the FLE. If fle denotes a free variable, it is sufficient to check the bitwidth. If the argument denotes a constant, the constant's value must be checked.

3. Match: expression × expression → boolean.

$$Match(Sys, fle) := \begin{cases} true & : \text{ if } Oper(fle) = Oper(Sys) \\ & \quad \wedge\ Width(fle) \ \geq \ Width(Sys) \\ & \quad \wedge\ Arg(s, f, Oper(fle)) \text{ is } true \text{ for all pairs } (s, f) \\ & \quad \text{of the operation's arguments} \\ false & : \text{ otherwise} \end{cases}$$

   This function checks, whether or not two operations and their arguments match.

Let us now turn to the problem of selecting a sufficient number of instances of component types. In some synthesis tools, the *number* of components is minimized by using NP-hard clique partitioning [Sie83]. Pfahler [Pfa88] pointed out, that this is the wrong solution method, because the minimum number of components can be computed in linear time using the left edge algorithm. In fact, the problem itself is oversimplified. One should at least try to minimize component *costs*.

Let M be the set of component types and let $m \in M$ be any component type. Let c_m be the cost and let x_m be the number of instances of type m. The objective then is to minimize $c = \sum_{m \in M}(x_m * c_m)$.

We now have to compute the set of constraints for x_m. Let o be an operation. Define a relation *matches* such that o *matches* $m \iff$ there exists a FLE e of m such that $Match(o, e) = true$. Let $f_{i,j}$ be the number of operations of type j used in control step i. Let $F_i = \{j \mid f_{i,j} > 0\}$ be the set of operations used in control step i. Let F_i^* be the powerset of F_i, that is, the set of all subsets of F_i. Let $a_{g,m}$ be $1 \iff \exists\, o \in g$ o *matches* m and 0 otherwise. Then, a condition for a sufficient number of copies is that

$$\forall i, \forall g \in F_i^* : \sum_{m \in M} (a_{g,m} * x_m) \geq \sum_{j \in g} f_{i,j} \tag{1}$$

This means: for all control steps and for each subset of operations, the number of instances of components which are able to perform any of the operations in the subset, must at least be equal to the operation frequency in the control step.

These conditions are also necessary unless some components are able to perform several functions in one control step. The vertical synthesis tool does not try to allocate several operations to a single component (apart from common subexpressions and storage-READs and -WRITEs).

Let $b_g = \overset{max}{i}(\sum_{j \in g} f_{i,j})$ and let $F^* = \bigcup_i F_i^*$ be the union of the F_i's.
Then, from (1) it follows that:

$$\forall g \in F^* : \sum_m (a_{g,m} * x_m) \geq b_g \tag{2}$$

Module selection therefore reduces to minimizing $c = \sum_{m \in M}(x_m * c_m)$ subject to the set (2) of constraints. This is a classical integer programming problem.
Example:
Assume that the library components are able to perform the following functions (again, we omit bitwidths):

m=1: {+}; m=2: {+,-}; m=3: {+,OR}; m=4: {-}

A control step containing one addition and one subtraction would generate the following three relations:

$$\begin{aligned}
\{+\} &: 1 * x_1 + 1 * x_2 + 1 * x_3 + 0 * x_4 \geq 1 \\
\{-\} &: 0 * x_1 + 1 * x_2 + 0 * x_3 + 1 * x_4 \geq 1 \\
\{+,-\} &: 1 * x_1 + 1 * x_2 + 1 * x_3 + 1 * x_4 \geq 2
\end{aligned}$$

A second control step containing two additions would update this to:

$$\begin{aligned}
\{+\} &: 1 * x_1 + 1 * x_2 + 1 * x_3 + 0 * x_4 \geq 2 \\
\{-\} &: 0 * x_1 + 1 * x_2 + 0 * x_3 + 1 * x_4 \geq 1 \\
\{+,-\} &: 1 * x_1 + 1 * x_2 + 1 * x_3 + 1 * x_4 \geq 2
\end{aligned}$$

The last relation is redundant now because it is *covered* by the first. A relation y is covered by a relation z iff all coefficients $\neq 0$ in y are also $\neq 0$ in z and the constant term of z is equal to or greater than the constant term of y. In the vertical synthesis tool, covered relations are removed on the fly, that is, while updating the relations for new control steps. This example demonstrates, that the number of relations does not grow linearly with the number of control steps. In fact, it may even decrease.

The number of relations usually is rather small, because control steps typically contain only a small number of different operation types. However, if many operators of different bitwidths are present in some control steps, $|F_i^*|$ can become quite large. Therefore, we have added a partitioning step.

Let o and q be operations occuring in the system specification. Define a relation $\sim$ such that $o \sim q \iff$ there exists a component m with (o *matches* m) $\wedge$ (q *matches* m). Let $\sim^*$ be the transitive closure of $\sim$. Then, let o and q be in the same partition $p_i \iff o \sim^* q$. Let P_i be the set of component types matching with at least one operation of p_i. Then, relations (2) can be generated for each set p_i of operations and each set P_i of components separately. Note, only those operations, which actually occur in the system specification are used for partitioning. Two components, which are able to perform the same function, do not necessarily belong to the same partition (this function could be unused).

Frequently, hardwired constants, arithmetic operations, operations with boolean results, multiplications all belong to separate partitions p_i. The following table gives an impression of the complexity of the integer programming problem for two practical examples. The execution times for the Gomory-I integer programming algorithm could be improved significantly by setting up separate component index spaces (m-spaces) for each partition.

Example	partition p_i	operations	variables	relations	cpu-time [3 MIPS]
pdp-8	1	mux	2	1	$\sim$ 2 msec
	2	0	1	1	$\sim$ 1 msec
	3	+,-,AND,OR,NOT,<,>,$\leq$	7	7	$\sim$ 3 msec
mc68.000	1	+,-,AND,OR,*,DIV,XOR	7	11	$\sim$ 121 msec
	2	=,<,<>,<=	10	10	$\sim$ 120 msec
	3	SHIFTxx	2	2	$\sim$ 90 msec
	4	mux	2	1	$\sim$ 86 msec
	5-14	partitions with $\|P_i\|$=1	1	1	$\sim$ 87 msec

Note that the number of variables is equal to the number of available component types. Libraries of about 100 component types can be easily handled. Due to the low complexity, no special handling is implemented for the trivial cases $|p_i| = 1$ or $|P_i| = 1$.

3.3 Matching Complex Operations

The concept of matching complex operations allows for the implementation of complex assignments by a single component activation. The component description contains signal assignments to outputs and assignments to the local state. In either case the right hand side is an expression e made up from constants, signals, state-accesses, and operations among them:
$e ::= const \mid signal \mid var \mid var[e] \mid \circ(e_1, \ldots, e_n)$, where $\circ$ is an $n-$ary operation, *signal* is an input signal, and *var* is an internal state variable of the component.

The matching of complex operations is described using the SuccAss implementation as the example. The SuccASS tool accepts arbitrarily complex FLEs. The expression 'SHIFTLL (a + b)' of section 3.2 can be implemented by a single activation of a component of type alu e.g..

Matching system and component behaviour uses the function $Match'$, which is defined as follows:

$$Match'(Sys, fle, op) := \begin{cases} u & : \text{if } IsOper(fle) \text{ with } u \text{ defined as:} \\ & \quad u = IsOper(Sys) \wedge (Oper(Sys) = Oper(fle)) \wedge \\ & \quad Match'(s, f, Oper(Sys)) \text{ is } true \text{ for all arguments } (s, f) \\ v & : \text{if } IsConst(fle) \text{ with } v \text{ defined as:} \\ & \quad v = IsConst(Sys) \wedge Value(Sys) = Value(Comp) \\ w & : \text{if } IsFree(fle) \text{ with } w \text{ defined as:} \\ & \quad w = (Width(fle) \geq Width(Sys)) \wedge Range(op) \\ & \quad \vee (Width(fle) = Width(Sys)) \wedge \neg Range(op) \end{cases}$$

This function now recursively examines both expressions. Note, that free variables are never used as arguments of the initial call. Therefore, the value *op* of the initial call is redundant.

Computation of covers of system behaviours now should start at the root of assignments and continue at the subexpressions corresponding to the free variables of the FLEs. The following is a sequence of replacements of expressions by component activations for the example of section 3.2:

```
      cc := (SHIFTLL (a+b))=0
-->   PARBEGIN cc(h,0,X); h <- (SHIFTLL (a+b))=0 PAREND
-->   PARBEGIN cc(h,0,X); comp(g,h); g <- (SHIFTLL (a+b)) PAREND
-->   PARBEGIN cc(h,0,X); comp(g,h); sadd(a,b,3,h) PAREND
```

In the case of several covers, the *version* with minimum costs is selected. Costs are calculated by preliminary floor-planning.

SuccASS generalizes component allocation in that several operations can be allocated to one component unless this results in a conflict at some of the component's inputs. This alleviates the special handling of storage-READs and -WRITEs, which was necesaary in the vertical synthesis tool. To this end, SuccASS defines a conflict relation among FLEs. This conflict relation is used during component allocation.

Further details about SuccASS are beyond the scope of this paper and will be presented separately.

3.4 Codegeneration

The matching between system and component behaviour has also to be performed by our retargetable codegenerator called MSSC [Now87]. The codegenerator does not actually synthesize hardware structures. Rather, it accepts a given hardware structure and generates the binary program such that this hardware behaves as specified in a high-level program[†]. In this paper, MSSC is just mentioned for the sake of completeness because it also has to match different behavioural descriptions: it performs a match between the high-level program and the given RT-level structure. This match based upon a graph representation of the complete system [Now87]. Matching nodes of this graph and the system behaviour follows the principles described above.

4 Future Work

FLE-concepts make it easy to add extensions. Three extensions could be easily implemented:

[†]Some authors, however, refer to this process as microcode *synthesis* [Mue83]

1. *Parallel blocks:* the left hand side of the implications could consist of several activations in a parallel block. This could be used if several components have to be activated in order to perform the required function. For example, the activation of the alu and a comparator may be required to perform a certain test. Another example is a set of alu-slices, which have to be activated in the same control step.
2. *Sequential blocks:* the left hand side could contain a sequential block. This could be used e.g. to load the input registers of some component before actually performing the required operation.
3. *Generic components:* the matching mechanism could be extended to generic components. This would help in VHDL-environments.

5 Conclusion

We have presented the concept of functions list entries (FLEs), which allow formal reasoning about functions performed by available components. This concept has been shown to be general enough to model a large number of practical system implementation problems. These entries can be created from a source level description of available components.

References

[Bal89] M. Balakrishnan, P. Marwedel: Integrated Scheduling and Binding: A Synthesis Aproach for Design Space Exploration, 26th Design Automation Conference, 1989

[Bro89] O. Broß: RT-Synthese mittels fortgesetzter Annäherung, master's thesis, Institut für Informatik, University of Kiel, Aug. 1989

[Dut89] N. Dutt: A Framework for Behavioral Synthesis from Partial Design Structures, PhD-thesis, University of Illinois at Urbana-Champaign, 1989

[Har89] P. Harper, S. Krolikoski, O. Levia: Using VHDL as a Synthesis Language in the Honeywell VSYNTH System, 9th Int. Symposium on Computer Hardware Description Languages, 1989

[Hor74] A.S. Hornby: Oxford Advanced Learner's Dictionary of Current English, Oxford University Press, 1974

[Lei81] G.W. Leive: The Design, Implementation and Analysis of an Automated Logic Synthesis and Module Selection System, PhD thesis, Carnegie-Mellon University, Pittsburgh, 1981

[Mar86] P. Marwedel: A New Synthesis Algorithm for the MIMOLA Software System, Proc. 23rd Design Automation Conference, 1986, pp. 271-277

[Jöh89] R. Jöhnk, P. Marwedel: MIMOLA Reference Manual, Version 3.45, Report 8902, University of Kiel, Institut für Informatik, 1989

[Mar79] P. Marwedel: The MIMOLA Design System: Detailed Description of the Software System, Proc. 16th Design Automation Conference, 1979, pp. 59-63

[Mar86a] P. Marwedel: An Algorithm for the Synthesis of Processor Structures from Behavioural Specifications, Microprocessing and Microprogramming, Vol. 18, 1986, pp. 251-262

[Mue83] R.A. Mueller, J. Varghese: Flow Graph Machine Models in Microcode Synthesis, 16th Annual Microprogramming Workshop (MICRO-16), 1983, p. 159-167

[Now87] L.Nowak : Graph Based Retargetable Microcode Compilation in the MIMOLA Design System, Proc. 20th Annual Workshop on Microprogramming (MICRO-20), Dec.1987, p. 126-132

[Pfa88] P. Pfahler: Übersetzermethoden zur automatischen Hardware-Synthese, PhD-thesis, University of Paderborn, 1988

[Sie83] D.P. Siewiorek, C.J. Tseng: Facet: A Procedure for the Automated Synthesis of Digital Systems, 20th Design Automation Conf., 1983, p. 490-496