Cross-Level Hierarchical High-Level Synthesis*

Oliver Bringmann and Wolfgang Rosenstiel

Forschungszentrum Informatik an der Universität Karlsruhe (FZI)
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
and Universität Tübingen, Sand 13, 72076 Tübingen, Germany

Abstract

This paper presents a new approach to cross-level hierarchical high-level synthesis. A methodology is presented, that supports the efficient synthesis of hierarchical specified systems while preserving the hierarchical structure. After synthesis of each subsystem the determined component schedule and the synthesized RT-structure are added to its algorithmic specification. This provides an automatic selection of optimized complex components. Furthermore, the component schedule enables the sharing of unused subcomponents across different hierarchical levels of the design.

1 Introduction

The specification of complex systems, which can be hierarchically composed of several subsystems is becoming more and more important. In this context, the complexity of the subsystems, also called components, is increasing as well. Examples for such subsystems are microprocessor cores, application specific functional units (e.g. DCT, FFT), and interface controllers. However, state-of-the-art high-level synthesis systems produce insufficient results in terms of quality of the result and execution time when considering large applications [1].

This paper addresses the problem of an efficient synthesis of hierarchical specified systems. Main feature of the presented hierarchical synthesis approach is the optimized integration of already synthesized module specifications as complex register-transfer components, in the further high-level synthesis flow. Note that such components can be autonomous in the entire system. For this, it is important that the behavioral specification, the synthesized register-transfer structure, and the already determined schedule of the used components are known throughout the whole high-level synthesis process. Therefore this information is added to the RT component library. This allows an efficient specification of less area consuming hierarchical designs while synthesis time can be reduced.

* This work is partially supported by the DFG.

1.1 Related Work

A closer investigation of existing approaches shows that the terms *hierarchical synthesis* and *complex components* are not used uniformly at algorithmic level. Three different methodologies, which use the term hierarchical synthesis, can be identified: First, data-flow graph clustering or partitioning methods followed by the synthesis of the clusters and partly of the clustered data-flow graph. Second, using already synthesized systems as components, but without regard to internal component structures ("*black-box reuse*"). Last, using already synthesized systems as components with the possibility of sharing subcomponents with regard to internal component structures ("*white-box reuse*").

A common technique of most of the clustering methods is the collection of operations with a high similarity measure into one cluster [2]. Then allocation, and binding is performed separately for each cluster. Further approaches can be distinguished by the clustering or partitioning strategy used [3]. The partitioning strategies are mainly based on the control-flow [4], the data-flow, the procedure-calls, or use techniques of regularity extraction [5]. During scheduling first the determined clusters and subsequently the clustered data-flow graph are considered. Another clustering approach tries to merge clusters with high similarity measure after the clustering step, in order to build complex data-path elements and to increase the cluster size [6].

Clustering techniques perform scheduling in a bottom-up traversal of the cluster or the given loop/subroutine hierarchy. The second approach for hierarchical synthesis allows the usage of already synthesized components for further high-level synthesis tasks without regarding specific component structures. The register-transfer library used in the Olympus system [7] can contain any component that is specified in HardwareC. A similar approach is integrated in AMICAL [8], with the difference that a proprietary component intermediate format is used for synthesis. However, all previous mentioned approaches perform component reuse and resource sharing just at one hierarchical level of the design, without respect to its component structures ("black-box reuse").

The approach presented in this paper offers a technique of hierarchical synthesis according to the last mentioned methodology and supports “white-box reuse”. This methodology requires an enhanced library model in order to consider already synthesized component structures during synthesis of modules at a higher hierarchical level.

A closer investigation of existing library models shows that specific libraries developed for the corresponding high-level synthesis system and technology-oriented libraries can be distinguished. Specific libraries are used in the high-level synthesis tools System Architect’s Workbench (SAW) [3], Synopsys Behavioral Compiler [4], and Olympus [7]. The libraries differ in the complexity of their components, but specific component structures and the behavioral component specification are not considered. Only some recently introduced approaches consider an enhanced component model. OSCAR [9] and ISE [10] represent complex components as behavior templates in order to match multiple operations by a single component. Additionally, in [10] components may contain multiple functional outputs. In contrast to the other systems mentioned, CATHEDRAL-III [6] uses a constructive approach. Complex data-path elements are constructed from primitive operators, which are mapped to primitive library components, or to hardware building blocks of a module generator. Reusing complex components as primitive operators or complex data-path elements is not possible. A similar technique is used in the pre-synthesis system ACE [11]. Their component models are more abstract, but the system only provides some architectural transformations, like the merging of components. GENUS is a generic, technology oriented register-transfer library used by the synthesis system BdA [12]. GENUS automatically generates a component for an operation from elementary function units. But it is not possible to hierarchically combine elementary components to build complex components. Hence, several operations of a given behavioral specification can not be mapped onto such complex components.

This paper is organized as follows: Section 2 describes the basic concepts of hierarchical synthesis including the underlying component model and the identification of complex components as one important subtask. Section 3 addresses the implementation of our approach into the high-level synthesis system CADDY-II. Some examples, including experimental results, are presented in section 4. Finally, this paper concludes with a summary in section 5.

2 A Concept for Hierarchical Synthesis

In this section, we explain in more detail our hierarchical synthesis concept. First, the underlying component model is described. Then an outline of our hierarchical synthesis concept is given. Finally, the main topic of this paper, the identification of complex components is presented.

2.1 Component Model

The underlying component model provides the reuse of arbitrary, already synthesized modules and includes the VHDL behavioral description, the RT structure, and the calculated schedule to the conventional high-level synthesis library model. Hence, the components may contain a separate controller, such their can be autonomous in the entire system. This component model introduces a “white-box reuse” approach, where the synthesis system can decide automatically, whether the additional component information are used or not. As opposed to conventional approaches no inline-expansion of the component specification is needed to perform optimization across different hierarchical levels. Therefore, the overall synthesis time can be kept low. Furthermore, the visible component structure can be used to improve technology-dependent optimizations during high-level synthesis.

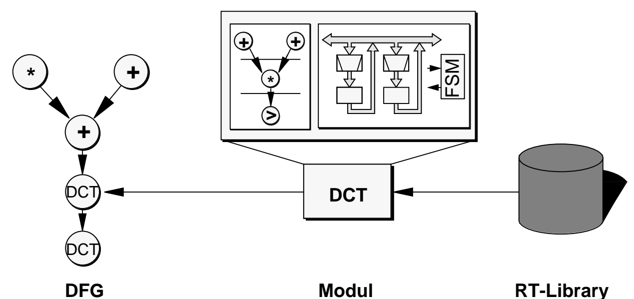

Figure 1. Example of a “White-Box” Component

2.2 Hierarchical Synthesis Concept

The proposed hierarchical synthesis technique is illustrated in figure 2. The entire system is synthesized in bottom-up traversal of the hierarchy. Each symbol represents a subdesign which is saved in the component library after synthesis. Based on the enhanced component module an automatic selection of optimized complex components and some optimizations across different levels of hierarchy can be performed. The most important optimization is the sharing of autonomous subcomponents across different levels of hierarchy. Due to space limitations, a more detailed description of that task is beyond the scope of this paper. For further information the reader is referred to [13].

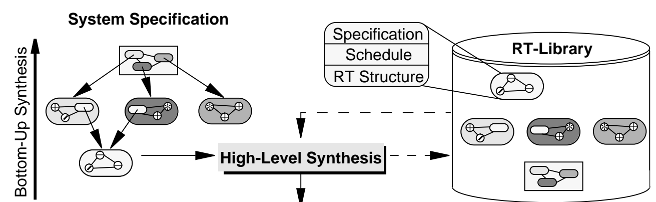

Figure 2. Hierarchical Synthesis Model

2.3 Identification of Complex Components

One task that has to be solved during hierarchical synthesis is the identification of complex components in the control-data flow graph, distinguishing *direct component instantiation* and *component matching*. Direct component instantiation denotes a user specified component instantiation in the algorithmic specification. In this case, the user may preset the allocation of a complex component by invoking the corresponding procedure in the specification. The term component matching denotes the matching of component and system data-flow subgraphs, in order to identify suitable optimized complex components for the design. The basis of component matching is the component behavioral specification, which can easily be transferred into a control-data flow graph. Thus, the matching problem needs to be solved first for the control-flow subgraphs and second for the data-flow subgraphs of the specification, in order to reduce the complexity.

Figure 3 illustrates both mentioned possibilities of complex component identification with three levels of hierarchy. In the first step, the mult-add subcomponent has been identified as a part of component f_3. Component f_3 has been instantiated directly by node f_3 of the overall system and can also be identified as a part of the overall system.

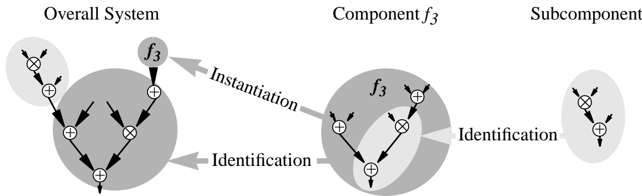

Figure 3. Identification of Complex Components

For the sake of clarity, this example illustrates only the data-flow graph matchings. The task of component identification is to be integrated in allocation as well as scheduling. Thereby, the synthesis system has to decide automatically whether specialized and optimized complex components or several primitive components are allocated. After scheduling, all DFG operations covered by one component are folded into a single complex operation node. Hence, no enhancements are needed in the further synthesis steps.

3 Implementation into CADDY-II

3.1 Overview of the CADDY-II Synthesis System

This section concentrates on the methods used for hierarchical synthesis regarding complex component structures in the high-level synthesis system CADDY-II. The underlying synthesis algorithms are only summarized here and can be found more detailed in [14], [15], [16] and [17]. The main synthesis steps used in the CADDY-II system are *data-flow analysis*, *allocation*, *scheduling*, *component adaption*, *assignment*, *data-path generation* and *controller generation*. Tasks of the allocation are the selection of the component types to be used from the library including the number of each component type, and the optimization of the clock frequency. The scheduling has two tasks: One is the assignment of a control step in which the operation starts. The other is the assignment of an operation to a component type. Task of component adaption is the insertion of additionally needed component ports including registers and multiplexers in order to select one of the inserted input ports. This is needed when common subcomponents shall be shared. Task of the assignment, also named binding, is the mapping of operations to component instances, the allocation of an optimized number of registers, and the mapping of variables to the allocated registers. For a detailed description of the assignment the reader is referred to [14]. The presented hierarchical synthesis methodology involves mainly the allocation and the scheduling phases.

3.2 Definitions

The hierarchical synthesis is based on the presented flexible concept of complex components. In the following, we will define some useful notations. We define two main data structures for hierarchical synthesis. First, the tuple $S := \langle D_s, CFG_s, A_s \rangle$ which denotes the data structures of the currently synthesized subsystem, where D_s represents the data-flow graph, CFG_s the control-flow graph, and A_s the currently allocated components of the system. The data-flow graph D_s is annotated with additional information about the assigned components and the assigned clock cycle after each synthesis step. After synthesis, the estimated or back-annotated physical component parameter P_c and a list of all allocated subcomponents C_c are added to the tuple S, before this new synthesized component can be added to the component library L.

Second, the component list L, which contains all available components C_c of the library can be defined by the tuple $C := \langle C_c, D_c, CFG_c, P_c \rangle$ with $C \in L$, where C_c represents all used subcomponents, D_c represents the scheduled and assigned component data-flow graph, CFG_c represents the component control-flow graph, and P_c the list of estimated or backannotated physical component parameters. $C_i \in C_c, \forall\, i \in \{1, \ldots, n\}$ denotes the i-th subcomponent out of n used subcomponents from the component at higher hierarchical level and is of the type $\langle C_c, D_c, CFG_c, P_c \rangle$.

3.3 Allocation with "White-Box" Components

As mentioned above, the main task of the allocation is the determination of the component types including their

number, from a library. Objective of this section is the extension of the existing allocation method [15], in order to handle the white-box component model. The principal algorithm is given in the following:

```
algorithm Allocation(S, L)
   A_s := Initial_Allocation(D_s, L);
   repeat
      inc_dec_components_and_evaluate(A_s, S, L);
      A_opt := best_allocation;
      if A_s <> A_opt then
         A_s :=A_opt;
      end if;
   until no_improvement_found_in_last_iterations;
end Allocation;
```

Algorithm 1. Allocation of Complex Components

The search process strongly depends on the function *inc_dec_component_and_evaluate*, which generates iteratively a series of new allocations by incrementing and decrementing the number of components and evaluates the allocations. This function is guided by a global estimation function, based on the probabilities of scheduling DFG operations into given control steps. Important for white-box components is the calculation of an initial allocation and the matching of component DFGs, if the number of allocated components should be increased. The following algorithm determines an initial allocation with respect to white-box components:

```
algorithm Initial_Allocation(D_s, L)
   M_c := set of all components C ∈ M, the operations of which cover
          at least one DFG node;
   for all C ∈ M_c do
      save D_s; N(C) := 0;
      n_c := choose a root node of the component DFG D_c;
      for all n_s ∈ D_s, with oplist(n_s) ⊂ oplist(n_c) do
         D_m := matching(D_s, D_c, n_s);
         if D_m ≠ ∅ then
            N(C) := N(C) + |D_m|;
            D_s := D_s \ D_m;
         end if;
      end do;
      cost(C) := area(P_c)^α · performance(P_c)^β ·
                 est_resynth_area(C)^δ / N(C)^γ ;
      restore D_s;
   end do;
   A_s := component set, selected in order of increasing cost(C) and
      the specific component types, so that no resource conflicts
      occur during ASAP scheduling;
   return A_s;
end Initial_Allocation;
```

Algorithm 2. Generation of Initial Allocations

Task of the matching algorithm is to solve the data-flow graph isomorphism problem between D_s and D_c with the known root node n_s. This can easily be done in a topological sorted graph with a complexity of $O(|V|+|E|)$ by a simple comparison of the operation lists, the ports, and the interconnections, where $|V|$ denotes the number of nodes and $|E|$ the number of interconnections of the component data-flow graph D_c. The matching algorithm is also used in the main allocation phase, especially in the algorithm *inc_dec_component_and_evaluate* to calculate a set of feasible components within an increment step [15], based on the same cost function *cost(C)*. The weighting parameters α, β, γ, and δ can be chosen by the user. The complexity of the overall allocation algorithm amounts $O(|C_r|^3)$, where C_r denotes the set of matching components.

The algorithm *Initial_Allocations* heuristically determines the number of covered data-flow graph nodes *N(C)* of D_s, for each component, in order to generate a ranking of components with similar operation lists. The actual initial allocation step is directed by the generated component ranking. Note that the function *oplist* returns the operation list of a subcomponent. For a detailed description of further allocations steps the reader is referred to [15].

3.4 Scheduling with “White-Box” Components

Goal of the scheduling is to assign of operations to control steps and to component types. Furthermore, the respective component DFG and the already determined component schedule are to consider. Inputs of the scheduling algorithm are the data-flow graph of the system D_s and the scheduled component data-flow graph D_c. In contrast to the system-DFG D_s, the component-DFG D_c consist of subcomponent nodes, which are assigned to component types and control steps, instead of operation nodes.

As scheduling algorithm list scheduling is implemented where the single clock cycles are scheduled consecutively. To keep the run time complexity low and to avoid backtracking a global heuristic estimation function, based on the probabilities of scheduling DFG operations into given control steps, is used to guide the decisions in every clock cycle. In addition to the estimation function the cost function *cost(C)* presented for allocation is also used here. If no valid schedule can finally be obtained, the allocation is rejected and a new allocation is generated.

The extension of the scheduling algorithm has to consider the additionally allocated components caused by white-box components, in order to find subcomponents which can be shared with other components. Furthermore, in each step of the list scheduling algorithm and for every operation out of the ready set, all possible component types must be determined and evaluated. Therefore the above matching algorithm must be called for every operation of the ready set in each clock step. Note that the ready set contains all operations of the current clock step with already determined inputs. Due to the linear time complexity of the

matching algorithm, the effect in terms of execution time can be neglected. Assigning DFG operations of the ready set to component types at a given clock step is done by the following algorithm. Note that this algorithm is called in every iteration of the list scheduling algorithm, that is for every clock step.

---

algorithm generate_and_evaluate_assignments(S, C, R, c_{step})
  while $R \neq \emptyset$ **do**
    select $op \in R$; $R := R \setminus \{op\}$; $M_c := \emptyset$;
    for all $C \in L$ **do**
      M_m := matching(D_s, D_c, op);
      if $M_m \neq \emptyset$ **and** ($C \in A_s$ **or** ($C \in C_c$ **and** unused(C,c_{step}))) **then**
        $M_c := M_c \cup \{M_m\}$;
      end if;
    end do;
    for all $C \in M_c$ **do**
      evaluate_component_assignment(S, C, op, c_{step});
      generate_and_evaluate_assignment(S, C, R, c_{step});
    end do;
  end do;
end generate_and_evaluate_assignments;

Algorithm 3. Component Type Binding during Scheduling

The algorithm recursively generates all feasible assignments to component types and all operation subsets in the current clock step. Premise of the algorithm is the already determined set of actual ready components R. The function *unused* checks by using the component CDFG, if a subcomponent is not used in the current control step. Note that the component library L can be reduced before calling the algorithm *generate_and_evaluate_assignments* to components which covers at least one node of the DFG D_s. A detailed description of the underlying scheduling algorithm including the used global estimation function can be found in [16].

4 Experimental Results

Experimental results of our approach on several designs, including some benchmark circuits, are given in this section. First, just for reasons of comparability the results for the fifth order elliptic wave filter benchmark are given. This benchmark demonstrates two essential features of our hierarchical synthesis approach: First, identification of complex component structures in the system data-flow graph, and second, the possibility of sharing subcomponents. The filter benchmark consists of eight data-flow subgraphs which match the multiply-accumulate component. Table 1 shows some results for different allocations with and without subcomponent sharing, and compares this with the traditional component model. In this table, a component with a data initiation interval of one clock cycle and an execution time of two clock cycles is specified by the notation '1 : 2', for instance.

As a result, we get the performance improvements listed in column entitled "gain". In this example the, speed-up of the design is up to 5 clock cycles with equal hardware costs. This is because the multiply-accumulate component may share the internal adder, if an additional adder is needed. In contrast, if subcomponent sharing is not supported, a resource conflict can only be solved by adding a further clock step. The CPU time for the filter on a Sun SPARC 20 was less than 2 seconds for a fixed set of allocated components and less than 12 seconds for an enlarged design space exploration by synthesizing different sets of automatically allocated components.

Table 1. 5th Order Elliptical Wave Filter

with complex components						without complex components			
resources			clock steps (cs)			resources			clock steps (cs)
+	MAC	MAC	with sharing	without sharing	gain	+	*	*	
1 : 1	1 : 2	1 : 3				1 : 1	1 : 1	1 : 2	
1	1	0	16	20	4	2	1	0	16
2	1	0	15	15	1	3	1	0	15
1	2	0	14	19	5	3	2	0	14
1	0	1	18	21	3	2	0	1	19
1	0	2	17	21	4	3	0	2	17

Second, we will present the results of the FDCT benchmark, shown in table 2. The given component costs are taken from [9] and amount 10 units for using an adder, 20 units for using an multiplier, and 25 units for using a multiply-accumulate unit. The column entitled "costs" is filled with the area-time product as cost function. Nine different data-flow subgraphs are identified and mapped to the multiply-accumulate unit by the system. Applying subcomponent sharing, a speedup of up to 8 clock cycles can be achieved. In comparison to non-encapsulated components, the performance results of the synthesized circuits are equal in most of the determined cases, while the area costs can be reduced. Particularly, the optimal circuit in relation to the area delay ratio is synthesized using two multiply-accumulate units and two adders. The CPU time for the FDCT benchmark was less than 4 seconds for a fixed set of allocated components and less than 16 seconds for an enlarged design space exploration.

Table 2. Fast Discrete Cosine Transformation

with complex components							without complex components			
resources				clock steps (cs)			resources			clock steps (cs)
+ / −	*	MAC		with sharing	without sharing	gain	+ / −	*		
10	20	25	costs				10	20	costs	
1	0	3	765	9	14	5	4	3	900	9
1	0	2	720	12	18	6	3	2	770	11
1	1	2	990	11	19	8	3	3	810	9
2	0	2	700	11	16	5	4	2	880	11
2	1	2	900	10	13	3	4	3	900	9
2	0	3	855	9	13	4	5	3	990	9
2	0	1	810	18	20	2	3	1	900	18
1	1	1	715	13	21	8	2	2	780	13
2	2	1	850	10	13	3	3	3	810	9
2	1	1	715	11	13	2	3	2	770	11

Finally, the results of the simulated annealing processor taken from [18] are presented. At first, the needed floating-point components and then the overall simulated annealing algorithm have been specified. The high-level synthesis system CADDY-II maps all floating-point operations to the previous designed components and synthesizes the entire system hierarchically with respect to the used floating-point components. All instantiated subcomponents of the floating-point components are now ready to be used as shared components within the entire simulated annealing design. The floating-point multiplier for instance, consists of an integer multiplier, an integer adder, and a barrel shifter. These components can be used additionally for other integer arithmetic operations of the whole design. As a result, all specified arithmetic operations could be covered by the subcomponents of the floating-point units. Table 3 lists the synthesized results for a hierarchical and a inline-expanded description. By using encapsulated components, the controller size of the simulated annealing processor could be reduced to 46% of the controller size of the inline-expanded description by an equal performance. The CLB count is related to the Xilinx XC4000 family and is given to get more detailed information. The CPU time could be reduced from 20 to 7 seconds when using the hierarchical description.

Table 3. Simulated Annealing Processor

Components		Hierarchical Description	Inline-Expanded Description	Area Reduction
Datapath:	Multiplier	1	1	0 %
	Add/Sub	2	2	0 %
	Barrelshifter	1	1	0 %
	Leading_Zero	1	1	0 %
	Comparator	2	3	33 %
	Multiplexer	19	27	30 %
	Register	11	12	8 %
Controller:	Gate Equivalents	158	307	49 %
	CLBs	56	104	46 %
	States	11	26	58 %

In summary, the presented results show several advantages of the hierarchical synthesis method regarding complex component structures. The system automatically identifies optimized complex components which cover parts of the data-flow graph. When using subcomponent sharing, the synthesized circuits need less clock cycles under resource constraints, and are less area consuming under timing constraints. Furthermore, the run-time of the synthesis algorithm could be decreased compared to non-hierarchical approaches.

5 Summary and Conclusion

This paper presented a new approach for hierarchical high-level synthesis regarding complex component structures. The presented experimental results encourage further investigations in this area. The advantages of the presented approach are: First, the concept of complex components offers the basis for a hierarchical synthesis methodology with respect to specific component structures, in order to increase the degree of optimization. Second, resource sharing can be performed across different levels of hierarchy of autonomous components, with a separate controller. Third, each synthesized module can be reused in the same design as a complex register-transfer component. Finally, multiple instances of one component have to be synthesized only once.

6 References

[1] R. Bergamaschi: *Productivity Issues in High-Level Design: Are Tools Solving the Real Problems?.* Proceedings of DAC, 1995.

[2] M. McFarland, T. Kowalski: *Incorporating Bottom-Up Design into Hardware Synthesis.* IEEE Transactions on CAD, vol. 9, 1990.

[3] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, R. L. Blackburn: *Algorithmic and Register-Transfer Synthesis: The System Architect's Workbench.* Kluwer, 1990.

[4] T. Ly, D. Knapp, R. Miller, D. MacMillen: *Scheduling using Behavioral Templates.* Proceedings of DAC, 1995.

[5] D. Sreenivasa Rao, F. Kurdahi: *Hierarchical Design Space Exploration for a Class of Digital Systems.* IEEE Transactions on CAD, vol. 1, pp. 282-295, 1993.

[6] W. Geurts, F. Catthoor, H. De Man: *Quadratic Zero-One Programming-Based Synthesis of Application-Specific Data Paths.* IEEE Transactions on CAD, vol. 14, pp. 1-11, 1995.

[7] D. C. Ku, G. De Micheli: *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer, 1992.

[8] P. Kission, H. Ding, A. Jerraya: *VHDL Based Design Methodology for Hierarchy and Component Re-Use.* Proceedings of EURO-VHDL, 1995.

[9] B. Landwehr, P. Marwedel, R. Dömer: *OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming,* Proceedings of EURO-VHDL, 1994.

[10] R. Ang: *Library Insertion and Reuse of Datapath Components in High-Level Synthesis.* PhD Thesis, University of California, Irvine, 1996.

[11] B. G. Hald, J. Madsen: *A Flexible Representation for High-Level Synthesis.* Proceedings of 2nd Asian Pacific Conference on Hardware Description Languages, 1994.

[12] P. Jha, N. Dutt: *Design Reuse through High-Level Library Mapping.* Proceedings of European Design & Test Conference, 1995.

[13] O. Bringmann, W. Rosenstiel: *Resource Sharing in Hierarchical Synthesis.* Proceedings of ICCAD, 1997.

[14] W. Rosenstiel, H. Krämer: *Scheduling and Assignment in High-Level Synthesis.* In R. Camposano, W. Wolf: High-Level VLSI Synthesis, Kluwer, 1991.

[15] P. Gutberlet, J. Müller, H. Krämer, W. Rosenstiel: *Automatic Module Allocation in High-Level Synthesis.* Proc. of EURO-DAC, 1992.

[16] P. Gutberlet, H. Krämer, W. Rosenstiel: *CASCH - a Scheduling Algorithm for High-Level Synthesis.* Proceedings of EDAC, 1991.

[17] P. Gutberlet, W. Rosenstiel: *Timing Preserving Interface Transformations for the Synthesis of Behavioral VHDL.* Proceedings of EURO-VHDL, 1994.

[18] B. Eschermann, O. Haberl, O. Bringmann, O. Seitz: COSIMA: *A Self-Testable Simulated Annealing Processor for Universal Cost Functions.* Proceedings of EuroASIC, 1992.