

# A Novel Specification Model for IP-based Design

Stephan Klaus and Sorin A. Huss

Integrated Circuits and Systems Laboratory  
Department of Computer Science  
Darmstadt University of Technology  
Alexanderstr. 10, 64283 Darmstadt  
Germany

E-mail: {klaus|huss}@iss.tu-darmstadt.de

## Abstract

*The proposed specification method denoted as hierarchical CoDesign Model allows a hierarchical system level specification of distributed embedded systems including both definition and reuse of IP cores. Input/Output Relations are introduced as a very general and powerful means of encapsulating internal implementation details and of describing data as well as control flow information on different levels of abstraction. Two novel algorithms are proposed, which allow to derive dynamically task descriptions for the IP encapsulation as well as flat specification models for design space exploration on different levels of detail. This reduces considerably the descriptive complexity of specifications and therewith the execution times of synthesis and design space exploration algorithms.*

## 1. Introduction

Shorter time to market and increasing complexity of distributed embedded systems force developers to design larger systems in less time, while reducing the production costs too. Therefore, reuse of IP cores becomes mandatory for a successful design at system level. However, even medium scaled embedded systems are so complex, that a flat representation becomes too difficult for humans to handle and to comprehend. In the same manner as a flat representation becomes too complex, the execution time of synthesis and design space exploration tools grows unacceptably. Due to this fact, the design flow including all necessary implementation steps should be accomplished on the current adequate level of detail only. This means, that a hierarchical specification model has to provide dynamically flat representations at any submodule level and any cut of hierarchy. On the one hand it is too costly at system level to

schedule communication in a very detailed manner, but on the other hand the overall execution time can be optimized only, if the communication protocol is considered [13].

Synthesis of embedded systems involves various steps such as allocation, scheduling, binding, and verification, whereby real-time and other constraints must be met. In particular the non-functional requirements of embedded systems such as timing, cost, power consumption, are relevant for the later commercial success. These parameters and a description of both data and control flow must be captured by the specification in an uniform manner on different levels of abstraction. Such a multi-level representation should be able to cover the external behavior of IP cores, whereby implementation details should be hidden. Thus, a reduction of the descriptive complexity of embedded systems and both definition and reuse of IP cores become possible.

## Figure 1. Top Level Module

## 2. Problem Formulation

The example of Figure 1 highlights a typical scenario in embedded system design. A functional partitioning subdivides the top level module  $T$  into three tasks  $A$ ,  $B$ , and  $C$ , whereas for task  $A$  a predefined IP core is chosen. Task  $B$  and  $C$  shall form self-contained submodules, which are separately implemented in a bottom-up manner and may define new IP cores.

From this example the needs for a specification methodology are obvious. If a new embedded system is designed, IP cores are reused and possibly new cores are defined. A general and uniform means of description for IP cores and systems is thus needed. Therefore a representation of

the task behavior on different levels of detail is necessary, which is capable to represent IP cores in the end. Due to the complexity of design space exploration, this means of description should be as simple as possible. The behavior of tasks is not to be characterized by their detailed internal implementation but by their external properties necessary for a design space exploration. The mean of description should be able to capture the behavior of subtasks and to represent the data and control flow on higher levels of abstraction. State of the art synthesis algorithms should be used and therefore methods are needed to derive dynamically flat task graph models from parts of the hierarchical specification. Multiple views of the hierarchical model may be generated upon user request on different levels of detail, thus finding the current necessary trade of between accuracy of the model and execution time of the design steps.

### 3. Related Work

Next to Finite State Machines and to abstract models like UML, task graphs are a widely spread means for specification of embedded systems. Beside data flow captured by task graphs, embedded systems mostly include control flow parts, which must be definitely taken into account for a complete and correct specification. Therefore, a lot of research was done extending task graphs by control flow information. The CoDesign Model (CDM) [1] assigns Input/Output Relations to each process, whereas the System Property Intervals (SPI) model [7] [15] uses mode tags to capture different control flow behavior. Conditional Process Graphs (CPG) [6] and extended Task Graphs (eTG) [10] model control flow by means of attributed data flow edges. In addition eTG supports conditions on input edges of tasks too, thus introducing a new class of tasks: The Select Task. But all of these models do not allow a hierarchical specification, which is mandatory for the design of large systems. In [5] a new hierarchical specification method is proposed, but the functionality and the process behavior are defined on the lowest level only, i.e., in the leaves of the model. Therefore, synthesis algorithms can be applied to this static level only. Hierarchical partitioning and scheduling based on task graphs is discussed in [4], but this model is not able to directly capture control flow information. The main disadvantage of hierarchical models like [3][12] is, that the synthesis algorithms are working on the static specification hierarchy only and not on flexible, i.e. dynamically, decompositions. Therefore, optimization just can take place inside the functional partitioning. From this fact a loss in optimization flexibility follows. Also in the area of high level synthesis hierarchical models are explored in the past. An overview can be found in [11]. But in contrast, this work concentrates on system level specification and IP core encapsulation.

## 4. Computational Model

The proposed specification method denoted as hierarchical CoDesign Model (hCDM) is intended as a means for the description of IP core encapsulation and for system level synthesis on different levels of detail. The model is a hierarchical extension of [2] in the same manner as [8] extends finite state machines to a hierarchical model. In contrast to this model, behavioral compilers are aimed to lower levels of hardware synthesis only. A dynamic hierarchical decomposition property is its main advantage compared to the state of the art of computational and specification models for embedded systems.

### 4.1. Hierarchical Specification Method

A hCDM is a directed acyclic graph consisting of computational tasks, inputs, outputs, and dependencies between the tasks. Additionally, each task can be defined as a separate hCDM thus introducing recursively a hierarchical structure.

**Definition 4.1 (hCDM Graph) :**

*The hierarchical CoDesign Model consists of a directed acyclic graph  $\mathcal{G} = (V, E)$  whereas*

- $V = I \cup T \cup O$
- $E$  is a set of 2-tuples  $(v_i, v_j)$ ,  $v_i, v_j \in V$
- $I$  denotes the set of inputs:  
 $\forall i \in I. \neg \exists (v, i) \in E. v \in V$
- $O$  denotes the set of outputs:  
 $\forall o \in O. \neg \exists (o, v) \in E. v \in V$
- $T$  is the set of computational tasks:  
 $\forall t \in T. \exists (t, v_1) \in E \wedge \exists (v_2, t) \in E. v_1 v_2 \in V.$

*Each task  $t \in T$  in a hCDM  $\mathcal{G}$  can be defined recursively as a separate hCDM denoted as  $\mathcal{G}' = (V', E')$ . Thereby each input edge of  $t$   $(v, t) \in E, v \in V$  becomes an input  $i' \in I'$  and each outgoing edge  $(t, v) \in E, v \in V$  becomes an output  $o' \in O'$  of  $\mathcal{G}'$ .*

The assignment of inputs/outputs to edges of the parent hCDM introduces hierarchical edges and thus a hierarchical data flow. These edges are visualized as the links between the two levels in Figure 1.

In addition to this complete hierarchical composition, the current model view embodies a subarea of interest: It is defined by the root of the subsystem and the current level of detail.

**Definition 4.2 (Current Model View) :**

*The current model view of a hCDM  $\mathcal{G}$  is defined by a 2-tuple  $\mathcal{A} = (r, L)$  whereas*

- $r \in T$  denotes the root task and
- $L \subseteq T$  the current level of detail.

The root defines the entry point of the subgraph and the level of detail denotes the current leaf tasks. Note that the current leaves are not necessarily leaves of the original hCDM. Figure 2 illustrates the definition of the current model view by means of the complete hierarchical view of the example of Figure 1. Two different model views are defined: First, task  $T$  represents the root of the submodule and  $A, G, H, I, E, J, K, C$  the current level of detail, respectively. The second model encapsulates task B on a higher level and explores task C in a more detailed manner.

### Figure 2. Two Hierarchical System Views

The current model view forms a flat hCDM and therefore standard scheduling, allocation and binding algorithms can be applied in a straight forward manner. Figure 3 depicts the resulting flat hCDMs for both models of Figure 2. The dashed lines visualize the hierarchy in the original model and the arrows denote the data flow.

### Figure 3. Partially flattened hCDM Models

In Section 5 algorithms are proposed to derive the current model view according to user requests. So, it is possible to manipulate dynamically the hierarchical structure in a systematic way, which allows to adopt flexibly the level of detail to the necessary level. So, the complexity of the system level synthesis steps can be easily reduced to the necessary minimum, considering the needed accuracy.

## 4.2. Task Behavior

A hCDM is intended to jointly represent data and control flow information. Transformative applications like digital signal processing or multimedia, e.g. MP3 decoder or encoder, are dominated by data flow behavior, but include also a few control flow decisions. The data and control flow information of tasks should be captured on different levels of abstraction by means of one and the same mechanism. Especially because of modeling the control flow, a more sophisticated behavior description than a simple Petri-net activation rule, which is frequently exploited in task graph specifications, is mandatory.

In order to address this problem Definition 4.3 introduces behavior classes to denote possibly different control flow behaviors in a hCDM. A behavior class consists of an identifier and of a set of different behaviors according to possible control flow decisions.

#### Definition 4.3 (Behavior Class) :

A behavior class  $b$  is a 2-tuple  $b = (n, W)$ , whereas  $n$  is

a unique identifier and  $W$  a set of at least two different behaviors.  $\mathcal{B}_G$  is the set of all possible behavior classes in a hCDM  $G$ .

Figure 4 illustrates possible reasons for behavior classes. Each code fragment contains control flow decisions, which are reflected by the denoted behavior classes.

### Figure 4. Behavior Classes

Since a behavior class defines the set of all possible behaviors, a Condition restricts this set to a current valid behavior. During execution of a system a current behavior is selected from the set of all possible behaviors. Therefore, a condition embodies control flow decisions.

#### Definition 4.4 (Condition, Conditionlist) :

A condition  $c$  is a 2-tuple  $c = (n, w)$ , whereas  $w \in W$  with  $(n, W) \in \mathcal{B}_G$ .

A sequence of condition is a state-sequence  $Z$ .

A set of state-sequences is a conditionlist  $\mathcal{C}$ .

A possible state-sequence for example of Figure 4 is:  $Z = \{(type1, OFF)(a, T)\}$ . The conditions in a state-sequence are connected with logical *and*, while the state-sequences in a conditionlist are connected with logical *or*. Two equal conditionlists and the complete decision tree are presented in Figure 5.

### Figure 5. Conditionlists

Input/Output Relations detail the external behavior of a task. The input set defines the edges, which must present data to activate the task. Upon activation new data is generated on all edges specified by the output set. The IO Relation becomes valid under the referenced condition only.

#### Definition 4.5 (Input/Output Relation) :

For every  $t \in T$  a set of Input/Output Relations is assigned. A Input/Output Relation is a 3-tuple  $\langle in|out, \mathcal{C} \rangle$  whereas:

- $in \subseteq \{(v, t) | (v, t) \in E \wedge v \in V\}$ :  
subset of the input edges of  $t$
- $out \subseteq \{(t, v) | (t, v) \in E \wedge v \in V\}$ :  
subset of the output edges of  $t$
- $\mathcal{C}$  denotes the conditionlist.

Figure 6 illustrates the definition of IO Relations by means of task B on the left side of Figure 3. Later on, this behavior will be subsumed to a high level description of

this task. The example contains one behavior class named  $c$ , which has two different behaviors. Task  $H$  splits the control flow according to condition  $c$ , while task  $J$  merges the different paths. The output on edge 6 occurs only if condition  $c$  is true. The other tasks need data on all their input edges and upon activation new data on all their output edges is generated.

**Figure 6. Input/Output Relations**

In order to map and to transfer IO Relations to different levels of abstraction, some operators are to be introduced for the manipulation of conditionlists. Especially the union and the intersection of conditionlists and three more auxiliary functions are needed.

**Definition 4.6 (Operators) :**

*The two operators on conditionlists  $\oplus$ ,  $\ominus$  and three auxiliary functions  $Names$ ,  $All$ ,  $Expand$  are defined as follows:*

- **Names:**  $\mathcal{C} \rightarrow \{n_1, \dots, n_i\}$   
 $Names(\mathcal{C}) = \{n | (n, v) \in \mathcal{C}\}$
- **All:**  $\{n_1, \dots, n_i\} \rightarrow \mathcal{C}$   
 $All(\{n_1, \dots, n_i\}) =$   
 $\bigcup_{\forall s_1 \in W(n_1)} \{(n_1, s_1), \dots, (n_i, s_i)\}$   
 $\dots$   
 $\bigcup_{\forall s_i \in W(n_i)}$
- **Expand**  $\mathcal{C} \times \{n_1, \dots, n_i\} \rightarrow \mathcal{C}$   
 $Expand(\mathcal{C}, \{n'_1, \dots, n'_i\}) =$   
 $\bigcup_{\forall Z \in \mathcal{C}} Z \cup \{(n_1, s_1), \dots, (n_i, s_i)\}$   
 $\bigcup_{\forall s_1 \in W(n_1)}$   
 $\dots$   
 $\bigcup_{\forall s_i \in W(n_i)}$   
 $\{n_1, \dots, n_i\} = \{n'_1, \dots, n'_i\} \setminus Names(Z)$
- $\oplus : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$   
 $C_1 \oplus C_2 = Expand(C_1, Names(C_1 \cup C_2)) \cap$   
 $Expand(C_2, Names(C_1 \cup C_2))$
- $\ominus : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$   
 $C_1 \ominus C_2 = Expand(C_1, Names(C_1 \cup C_2)) \cup$   
 $Expand(C_2, Names(C_1 \cup C_2))$

The operator  $Names$  determines the set of all behavior classes in a conditionlist. According to a set of behavior classes  $ALL$  calculates a conditionlist, which contains all possible state-sequences.  $Expand$  takes a conditionlist and a set of behavior classes as an input and produces an identical conditionlist in canonical form, which allows the application of simple set operations for  $\oplus$  and  $\ominus$ .  $\oplus$  defines the intersection of two condition lists. The result is a conditionlist, which is valid under the conditionlists  $C_1$  and  $C_2$ .

In contrast,  $\ominus$  defines the union of two conditionlists. The resulting conditionlists is valid, if conditionlist  $C_1$  or  $C_2$  is valid.

## 5. Generating Current Model Views

In the sequel two algorithms are presented aimed to the derivation of flat subgraphs at variable levels of abstraction. Algorithm 1 calculates the flat hCDM defined by the given root, the current level of detail, and the complete hCDM. Different views on different levels of detail can be generated dynamically. Therefore, the normally static structures are traversed by this flexible method.

---

**Algorithm 1** BuildCurrentModelView

---

**In:**  $\mathcal{G}$ : hCDM  
 $r$ : Current Root  
 $LoT$ : Current Level of Detail

**Out:**  $aTasks$ : Current Task List  
 $aInputs$ : Set of Current Input Ports  
 $aOutputs$ : Set of Current Output Ports  
 $aEdges$ : Dependencies between the Tasks

- 1: **for all**  $p \in r.ports \wedge p.type=IN$  **do**
- 2:    $aInputs.insert(p)$  -- **Insert Input**
- 3: **end for**
- 4:  $testTasks = r.GetSubTasks()$
- 5: **while**  $testTasks.empty()=FALSE$  **do**
- 6:    $t=testTasks.removeFirst()$
- 7:   **if**  $(t \in LoT \wedge t.GetSubTasks()=\emptyset)$  **then**
- 8:      $aTasks.insert(t)$  -- **Insert Task**
- 9:   **else**
- 10:      $testTasks = testTasks \cup t.GetSubTasks()$
- 11:   **end if**
- 12: **end while**
- 13: **for all**  $p \in r.ports \wedge p.type=OUT$  **do**
- 14:    $aOutputs.insert(p)$  -- **Insert Output**
- 15: **end for**
- 16: **for all**  $n \in aTasks \cup aInputs$  **do**
- 17:   **for all**  $h \in n.GetHierarchicalOutEdges()$  **do**
- 18:      $n'=h.findSink(t, aTasks \cup aOutputs)$
- 19:      $aEdges.insert(edge(n, n'))$
- 20:   **end for**
- 21: **end for**

---

A flat representation allows for the application of standard algorithm for the subsequent design steps. The ports of the root task become the inputs and outputs of the current model view. The set of tasks is established by traversing the hierarchy tree starting from the root of the current model view down to a leaf or to a task in the current level of detail. This restricts the current task list to a well defined cut tree, since  $L$  in Definition 4.2 may not necessarily define a cut

tree. The data dependencies of the current model are determined by following the hierarchical data flow edges starting from all current tasks and inputs up to a task or to an output of the current model view.

After creating a flat representation of the hierarchical model, it is necessary to summarize the behavior of the subtasks. Algorithm 2 takes the IO Relations of these subtasks and derives a compact description of the encapsulated IP core.

---

**Algorithm 2** DeriveIORelFromSubTasks

---

**In:**  $t$ : Task

**Out:**  $ioR$ : New Input/Output Relations

```

1:  $N = \emptyset$ ;  $ioR = \emptyset$ ;  $ioR' = \emptyset$ 
2: for all  $io \in t.ioRelations$  do
3:    $N = N \cup Names(io.c)$ 
4: end for
5:  $condSet = All(N)$ 
6: for all  $c \in condSet$  do
7:    $in = \emptyset$ ;  $out = \emptyset$ 
8:   while  $in$  or  $out$  changed do
9:     for all  $io \in t.ioRelations$  do
10:      if  $((io.c \oplus c) \neq \emptyset \wedge (io.in \text{ has external port } \vee$ 
       $in \cap io.out \neq \emptyset \vee out \cap io.in \neq \emptyset))$  then
11:         $in = in \cup io.in$ ;  $out = out \cup io.out$ 
12:      end if
13:    end for
14:    for all  $e \in in \wedge e \in out$  do
15:       $in = in \setminus e$ ;  $out = out \setminus e$ 
16:    end for
17:  end while
18:   $ioR'.insert(in, out, c)$ 
19: end for
20: for all  $io \in ioR'$  do
21:    $c = io.c$ 
22:   for all  $io' \in ioR'$  do
23:    if  $io.in = io'.in \wedge io.out = io'.out$  then
24:       $c = io.c \ominus io'.c$ 
25:       $ioR' = ioR' \setminus io \setminus io'$ 
26:    end if
27:  end for
28:   $ioR.insert(io.in, io.out, c)$ 
29: end for

```

---

This algorithm can be used in a bottom-up design as well as in a top-down verification flow. In the first case the IO Relations are generated with respect to the behavior of the subtasks. In the second case it can be checked that the implementation of the subtasks is equivalent to the specification of the parent task.

Algorithm 2 takes a task as input and determines its IO Relations with respect to the behavior of its subtasks. This can be done recursively to bridge more than one level of ab-

straction. The generation process is illustrated in Figure 7 for the subgraph of Figure 6. At first, the set of all possible conditionlists is determined considering the IO Relations of all subtasks using the *ALL* operator. This example has two different conditionlists, depending on whether  $c$  is true or false. Then, the input and output sets for each conditionlist are determined considering all conditionlists and the underlying connectivity. The *for* loop in line 14 of Algorithm 2 removes all internal data edges. In a final step IO Relations are merged in case that the input and outputs sets are equal and all dispensable conditionlists are removed.

**Figure 7. Generating IO Relations**

The resulting two IO Relations highlighted in Figure 7 represent the behavior of task  $B$ , which was previously defined by eight IO Relations and six subtasks as illustrated in Figure 6. Thus, the proposed approach to IO Relation encapsulation hides the internal implementation and defines the external view of the envisaged IP core. In order to get a complete description of the IP core, the timing of the submodule must be determined as well. Therefore a state-of-the-art list scheduling algorithm was applied to the flat current model view defined by  $\mathcal{A} = (B, \{G, H, I, E, J, K\})$ . A detailed description of this algorithm can be found in [9].

These proposed algorithms support designers of embedded systems in generating flexible submodules on all desired levels of detail and in dealing with the dynamic control behavior of tasks on different levels of abstraction.

## 6. Results

In the previous section the behavior of the newly defined IP core of task  $B$  of Figure 6 was generated, thus reducing the descriptive complexity from 8 to 2 IO Relations. The reduction of the descriptive complexity is summarized in Table 1 for additional application examples. The numbers of IO Relations and conditionlists as well as the execution times for scheduling are compared for the detailed models with subtasks and the resulting higher level models. The execution times were measured on an AMD Athlon 750 MHz processor. First, the example of Figure 6 is illustrated. Next, a MP3 encoder specification is considered [14]. The last two examples are large randomly generated task graphs.

The reduction of the descriptive complexity and the resulting execution times of the scheduling step is about 65%. Not just the reduction of the number of IO Relations is important, but also the reduction of the number of conditionlists influences extremely the execution time of synthesis algorithm, because the complexity grows exponentially with the number of conditionlists. A reduction of the execution

	Detailed Model	System Level IP Model	Reduction
<b>Example of Figure 6</b>			
No. of IO Rel.	8	2	75%
No. of Cond.	1	1	0%
Exec. Time	2.4 ms	1.2 ms	50%
<b>MP3 Decoder</b>			
No. of IO Rel.	15	2	86 %
No. of Cond.	1	1	0%
Exec. Time	2.1 ms	0.54 ms	74%
<b>Large Example 1</b>			
No. of IO Rel.	35	4	88%
No. of Cond.	3	2	33%
Exec. Time	10.8 ms	2.8 ms	74%
<b>Large Example 2</b>			
No. of IO Rel.	65	8	88%
No. of Cond.	5	3	60%
Exec. Time	77.4 ms	10.4 ms	87%

**Table 1. Reduction of Descriptive Complexity**

times in a similar range can be expected for subsequent synthesis steps too.

**Figure 8. Schedule of Model 2 of Figure 3**

In general, a loss in optimization flexibility stemming from merging submodules into a smaller description is to be expected. But, if no resources are shared between the system level modules, a loss of implementation quality is not mandatory. As it can be seen from Figure 8, the overall scheduling result is the same considering the detailed model or the high level model. Of course, the high level model does not explore the detailed schedule, which is depicted in the highlighted block of the execution of task B. In this case just three scheduling steps are necessary in contrast to eight for the detailed model. In general, IP cores do not share resources and therefore no optimization flexibility is lost. Otherwise this problem has to be taken into account and a suited level of detail must be found in order to meet the required accuracy.

## 7. Conclusion

This paper describes a formal hierarchical specification method, which is aimed to both flexible definition and reuse of IP cores. Flat representations on different levels of ab-

straction can be generated on demand. So, at each point in time only the current necessary level of detail has to be taken into account and, therefore, the whole design process can be flexibly structured in a hierarchical manner. IO Relations are introduced as a very general and flexible means of description for data and dynamic control flow behavior on different levels of abstraction. This method of description captures the external behavior of IP cores thus hiding internal implementation details and it is therefore very well-suited for IP exploitations. The encapsulation of IP cores by merging IO Relations according to the proposed algorithm decreases considerably the descriptive complexity and thereby reduces the execution times of subsequent system level synthesis steps. A subject for future work is to determine cases and reasons, which allow encapsulation of modules without reducing the resulting accuracy of the model.

## References

- [1] W. Boßung, S. Huss, and S. Klaus. High-Level Embedded System Specifications Based on Process Activation Conditions. *Journal of VLSI Signal Processing, Special Issue on System Design*, Kluwer Academic Publishers, vol. 21, no. 3, pp. 277-291, 1999.
- [2] W. Bounq. Funktionale Spezifikation und Codesign eingebetteter Systeme. Phd thesis,, Technische Universität Darmstadt, 2001.
- [3] K. Chatha and R. Vemurl. Magellan: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, pages 42–47, 2001.
- [4] B. Dave and N. Jha. Cohra: hardware-software co-synthesis of hierarchical distributed embedded system architectures. In *Eleventh International Conference on VLSI Design*, pages 347–354, 1998.
- [5] M. Deegener and S. Huss. Design space exploration techniques for the codesign of embedded data processing systems. *IEE. Proc. Comput. Digit. Tech.*, Vol 145, No. 3, pp. 161-170, 1998.
- [6] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems. *IEEE/ACM Proc. Design, Automation and Test in Europe*, pp. 132-138, 1998.
- [7] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich. Hardware/Software Codesign of Embedded Systems - The SPI Workbench. *Proc. IEEE Workshop on VLSI, Orlando, USA, June 1999*.
- [8] D. Harel. Statecharts - A Visual Formalism for Complex Systems. *Sci. of Comp. Progr.*, Aug. 1987.
- [9] S. Klaus and S. Huss. Interrelation of specification method and scheduling results in embedded system design. In *Proc. ECSI Int. Forum on Design Languages*, Lyon, France, Sept. 2001.

- [10] S. Klaus, S. Huss, and T. Trautman. Automatic generation of scheduled systemc models of embedded systems from extended task graphs. In *Proc. ECSI Int. Forum on Design Languages*, Marseille, France, Sept. 2002.
- [11] A. A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviours for high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 7(3):380–412, Juli 2002.
- [12] Y. Li and W. Wolf. Hierarchical scheduling and allocation of multirate systems on heterogeneous multiprocessors. In *Proceedings European Design and Test Conference*, pages 134–139, 1997.
- [13] P. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *10th International Symposium on Hardware/Software Codesign (CODES 2002)*, pages 187–192, Estes Park, Colorado, USA,, May 2002.
- [14] T. Steiniger and J. Bieger. Implementation of a MP3 player on an Atmel FPSELIC configurable SoC. Project Alfa Core, Internal Report, Darmstadt, University of Technology, 2002.
- [15] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI- a system model for heterogeneously specified embedded systems. *IEEE Trans. on VLSI Systems*, pages 379–389, August 2002.