Mapping Statechart Models onto an FPGA-Based ASIP Architecture

Klaus Buchenrieder
Siemens AG
Corporate R&D
Otto-Hahn-Ring 6
81730 Munich, Germany
Klaus.Buchenrieder@zfe.siemens.de

Andreas Pyttel
Siemens AG
Corporate R&D
Otto-Hahn-Ring 6
81730 Munich, Germany
Andreas.Pyttel@zfe.siemens.de

Christian Veith
Siemens AG
Corporate R&D
Otto-Hahn-Ring 6
81730 Munich, Germany
Christian.Veith@zfe.siemens.de

Abstract

In this paper, we describe a system to map hardware-software systems specified with statechart models on an ASIP architecture based on FPGAs. The architecture consists of a reusable CPU core with enhancements to execute the behavior of statecharts correctly. Our codesign system generates an application-specific hardware control block, an application-specific set of registers, and an instruction stream. The instruction stream consists of a static set of core instructions, and a set of custom instructions for performance enhancements. In contrast to previous approaches, the presented method supports extended statecharts. The system also assists designers during space/time tradeoff optimizations. The benefits of the approach are demonstrated with an industrial control application comparing two different timing schemes.

1. Introduction

The preferred way to develop application-specific reactive systems is either hardware-oriented or software-oriented. The first approach is overall expensive, includes the design of custom hardware, and is usually found in low-volume markets such as military applications. The second approach involves the low-level programming of standard microcontrollers, and is geared towards inexpensive solutions for high-volume markets, such as electronic consumer goods.

The approach presented here is suitable for hardware-software implementations. It is based on Application Specific Instruction Processors (ASIPs) [17], whose recent popularity relies on results in the area of partitioning [14] and instruction set selection methods [10]. ASIPs are microprocessors with flexible architectures. ASIP research focuses on optimized instruction sets that depend on constraints and application requirements. One of the most active areas in Codesign research is the partitioning of system models into hardware and software modules, and to generate appropriate interfaces between them. ASIP research and partitioning research overlap, because instruction sets imply partitions, and influence the size and complexity of controllers and datapaths. In turn, they affect compilers, and the instruction streams they generate on the software side. Hardware-software partitioning implies moving functionality between software and hardware, which is similar to defining a „meta instruction set„ encapsulating hardware operations.

Our design method is based on extended statechart models. Statecharts constitute a widely accepted formalism for the specification of concurrent reactive systems [2]. The central issue of our work is the efficient mapping of extended statechart models to hardware structures. In contrast to the previous approaches, our system generates an application-specific hardware structure and an instruction stream that is tailored to it. The system is built on results presented in [13], where a hardware-software design method based on statechart/activity-chart combinations was presented. However, the approach was found to be too inefficient. Also, the chosen buses and protocols made it difficult to support parallelism. The new approach overcomes most shortcomings, and enables parallel implementations of statecharts in a natural way.

Throughout the research we studied the works of Drusinsky and Harel [5], who describe a method for mapping induced statechart trees to programmable devices using tree layout techniques. In [6], Drusinsky-Yoresh describes a method to implement statecharts with conventional logic blocks. The method is based on an exclusivity encoding state assignment procedure, an extended version of which is part of our own statechart synthesis method. Both previous methods, however, can only handle basic statecharts. The available methods to synthesize extended statecharts need VHDL as an intermediate step: The Statemate™ code generator produces behavioral VHDL, which is often not synthesizable. Narayan, Vahid, and Gajski [8] describe the SpecCharts language, which is a statechart-based front end to VHDL. The SpecCharts

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00 ©1996 IEEE

language is as powerful as extended statecharts, and also contains communication mechanisms such as ports and channels. However, the implementation of SpecCharts is limited by the current state of behavioral VHDL synthesis.

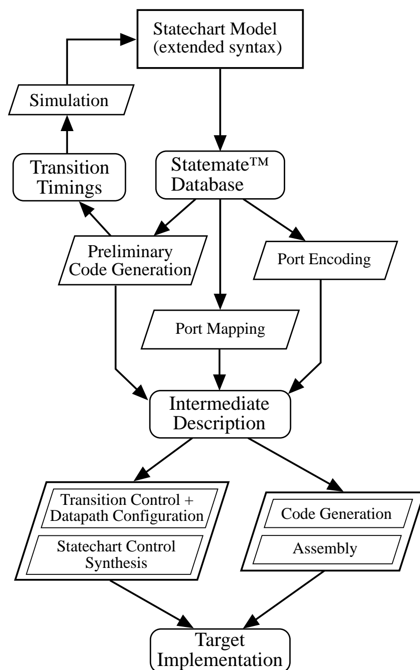

Figure 1. Codesign system architecture.

There is a rich body of literature covering the specification, design, and implementation of reactive systems. In the telecommunication market, SDL [11] is widely used, which has been recently enhanced with object-oriented features. There are SDL toolkits featuring a wide range of code generators. However, to our knowledge no method exists for hardware/software generation from SDL models. In [18], Selic et al. describe ROOM, an object-oriented methodology based on statecharts. ROOM models consist of concurrent actors, whose behavior is described with non-concurrent statecharts. Code generators are available for various target platforms and real-time operating system kernels. Although applicable to embedded applications, both these methods aim at distributed systems. The synchrony hypothesis that statecharts are based on was introduced by Berry, who introduced Esterel [1], an imperative programming language for reactive systems.

The paper is organized as follows: In section 2, we describe the design flow in our system, and present the underlying ASIP architecture. Section 3 contains our approach to the generation of hardware structures from statechart descriptions. In section 4 we present an example, and discuss the results we obtained. Section 5 summarizes our approach, and proposes future work.

2. Design flow and ASIP architecture

In the area of ASIPs, there has been tremendous progress: In [17], Sato et. al. describe a codesign system for ASIP development, which generates ASIP CPU cores along with compilers and simulators. The system accepts a set of C programs as input, which are profiled. The instruction set architecture is based on GNU‘s abstract machine model. Specific instruction sets are generated using integer programming, as described in [10]. Liem et. al. [16] use pattern matching to generate instruction set architectures for DSPs. Van Praet et. al. [19] and Leupers, Marwedel [15] present new approaches in the area of retargetable compilers for ASIPs, where instruction sets are extracted from existing architecture descriptions.

2.1. Input models and internal representation

An overview of the ASIP generation system is shown in Figure 1. It is based on specifications expressed as extended statechart models. The statechart formalism was introduced to overcome the limitations of conventional finite state machines (FSMs): they are sequential in nature and lack structuring elements. The state set of statecharts is separated into basic states, AND-states and OR-states. With AND-states, designers specify concurrency; with OR-states, they specify hierarchical models. In statecharts, there are transitions between basic states and also high-level transitions between hierarchical elements in the statechart. Figure 2 shows the top-level statechart of a process control system used as a running example in this paper. Dashed lines indicate concurrently active AND-states; states containing other states are called OR-states.

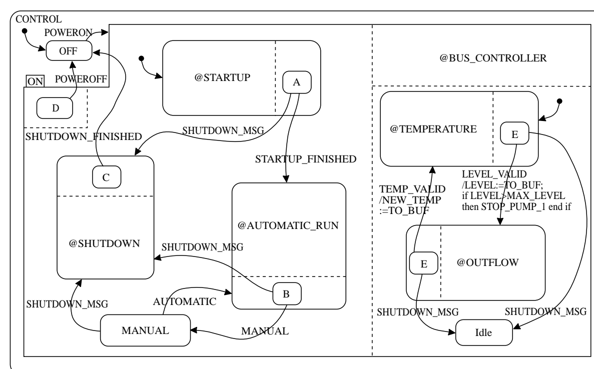

Figure 2. Process control statechart.

Conventional statecharts have no designated ports to the outside world. Consider the process control system depicted in Figure 7. It consists of pumps, valves, and a fluid-filled tank, whose level is to be held at a certain

mark. To control actuators, we introduced external ports for events, conditions, and data items into the model. The port names are kept as attributes in statecharts. A set of elements with the same port name form the protocol of the port. The element type of the port (event, condition, or data), the direction of the port (input, output, bidirectional), its protocol, and its width all together are called the type of the port. Conventional statecharts treat events and conditions as „things that occur„ and „things that persist„ without consideration for their physical implementation. Therefore, we added bitwidth attributes for events and conditions to the formalism. The encoding of events and conditions will be discussed in section 3.

An interface tool reads the elements of statechart models, and builds an internal representation. The internal model is implemented as an attributed tree, as every statechart induces a tree representation [5,6]. The leaves of the tree correspond to basic states, internal nodes represent AND-states and OR-states. Nodes are attributed by lists representing transition sources and destinations. The transitions are stored in a list structure. List elements point to a symbol table containing the symbolic events, conditions, and data items used in the model.

2.2. SCP ASIP architecture

The ASIP architecture SCP (StateChart Processor) presented here is based on a flexible CPU core architecture that communicates with statechart controllers, implemented as statechart logic arrays (SLA). An overview of the ASIP architecture is shown in Figure 3. It is implemented on a single FPGA with additional RAM. The CPU core, called TEP (Transition Execution Processor), consists of a control unit, a configurable datapath and register file, and RAM for instructions and data. The instruction stream is derived from the transitions in the statechart. For every transition, a function with a specific start address is generated. For degenerate transitions, the code consists of a single return statement. The size of the register file varies from three to 16 general-purpose registers plus zero to 64 constant registers.

The inputs of the SLA are called configuration register (CR), shown below the SLA in Figure 3. The CR holds the configuration of the system, consisting of conditions, current events, and states. The SLA produces two types of outputs: feedback and addresses. Feedback is gated to the event and state parts of the CR. Conditions are not fed back, but updated in software. For all active transitions, the SLA generates the corresponding starting addresses, and updates the array of address registers. The number of address registers corresponds to the maximum concurrency found in the statechart model. In our example, the maximum concurrency equals six.

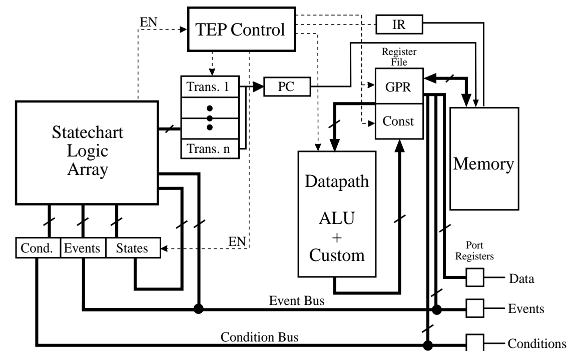

Figure 3. ASIP architecture.

The instructions of the CPU core are divided into basic and custom functionality, and are distinguished by the two highest bits in the opcode. Basic functionality includes load/store operations, branch instructions, shifting, and arithmetic/logical operations. The custom instructions are determined in an interactive process using a combination of simulation and analysis. Custom instructions are implemented as separate elements in the datapath. For custom instructions, any combination of arithmetic, logical, and shifting operations is allowed. Designs are only limited by the sizes of target FPGAs. The datapath of the TEP could be built exclusively from custom elements to speed up the system. On the other hand, a minimal solution may be generated, if the timing characteristics are consistent with the requirements.

An important part of the ASIP architecture are port instructions. Our architecture supports up to 64 ports, of which 16 may be external. There are data ports, and also ports for events, conditions, and states, which are all available to the TEP. These ports are connected to the CR. Thus, it is possible to alter statechart configurations per hardware (using the SLA) or per software (generating port instructions for the TEP). Event and condition ports are bidirectional. State ports are input ports for the TEP, because only the SLA can write to the state parts of the CR.

The SCP operates in so-called „configuration cycles„. A cycle starts with the TEP reading external events and conditions into the corresponding parts of the CR, and then enabling the SLA. The SLA produces the next configuration, the addresses of the active transitions, and enables the TEP. If no transition can be taken, all address registers are reset. Depending on the contents of the address registers, the TEP either executes the active transitions, or starts a new configuration cycle. Active transitions will often generate a new configuration. Therefore, the TEP enables the SLA after their execution by enabling the CR flip-flops. These „micro-steps„ are part of the semantics of statecharts, and described in [3,7]. Note that the transitions of a micro-step are executed serially,

although they are described as being concurrent in the model. This is legal, because the execution of concurrent transitions may not depend on an order according to the semantics of statecharts (if it did this would imply the presence of race conditions). With the execution of micro steps, it is possible to run into endless loops, which no amount of static analysis is able to detect. To our knowledge, data flow analysis of statecharts is unsolved, even more so for the extended statecharts of our approach. Thus, simulation of the model is the only recourse.

2.3. Instruction selection and code generation

After building the internal tree representation, we perform a preliminary code generation step, which provides timing information for the simulation of the model. For every transition, the code generator produces a code sequence assuming a minimal architecture with only three registers, no constant registers, and no custom instructions. In our current implementation, our timing model is very conservative, and assigns six cycles, the length of the longest instruction, to every instruction except memory operations, which take nine cycles. Additional penalties are not required, as the system runs at 8 MHz. Thus, the TEP is not slowed down by the memory bus. During preliminary code generation, states, events, and conditions are mapped to eight-bit wide symbolic ports. To illustrate the use of the ports, consider the ten conditions used in our example. They are mapped to two ports, the first fully used, in the second only the lower two bits being used (Figure 4). To set a condition in the first set, the port is loaded, an OR-operation is performed with the desired condition bit high, and the port is written back.

Using the timing information gained during preliminary code generation, the most time-critical configuration cycles can be found by simulating the model. Note that the length of a configuration cycle is the sum of the length of its individual micro-steps. Static analysis techniques would be preferred, but this problem has been shown to be NP-complete [4]. The critical cycles can be sped up by generating custom instructions, and by moving variables and constants to registers. The system points out the data items and operations that need consideration. Currently, there is no data flow analysis in our system. Therefore, designers must improve transitions similar to assigning „register„ declarations to variables in the C language. When all timing requirements are fulfilled, we perform the final code generation. Register variables and constants are assigned a final GPR or constant register. The remaining variables and constants are mapped to memory locations, and desired custom instructions are inserted. All ports (data, events, conditions, states) are mapped to port addresses, which completes the generated assembler code.

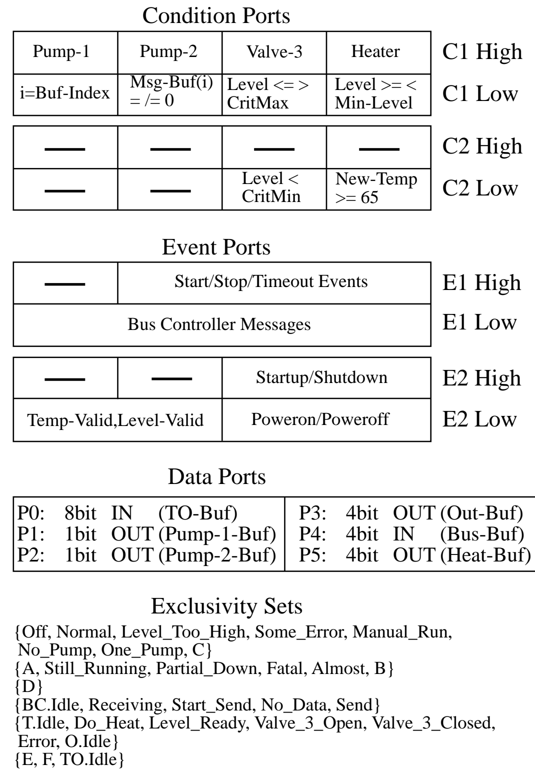

Figure 4. Port layout, exclusivity sets.

3. Statechart synthesis

Previous attempts at synthesizing statecharts only supported basic statecharts that consume and produce primitive events similar to conventional FSMs. The tree implementation method [5] poses difficult communication and timing problems, as pointed out in [6] and [13]. The PLA-style implementation of [6] introduces the problem of unspecified inputs which result in illegal transitions. The presented method extends the results of [6] by supporting conditions, encoded events, and complex actions. It also adds extra hardware to avoid problems caused by unspecified inputs. High-level transitions are handled by transformations as illustrated in Fig. 5. After the transformations, there are only transitions from and to basic states.

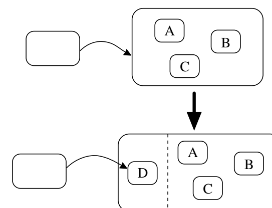

Figure 5. High-level statechart transformations.

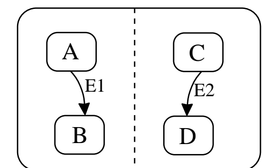

Figure 6. Unspecified inputs.

The implementation of statechart controllers is similar to the implementation of controllers based on FSMs. It consists of a state register - the configuration register CR - and a logic array (SLA). The SLA is synthesized using SIS [9] and TOS [12]. The remaining question is how to encode the CR. The problem of finding efficient and admissible encodings for the states of a configuration in a statechart has been discussed in [6]. The problem is twofold: An encoding of a configuration must uniquely identify the concurrent states it is composed of. On the other hand, combining the code words of individual states must produce a unique configuration encoding. The general problem of optimal statechart configuration assignments amounts to reachability analysis of statecharts, which is infeasible. It is obvious that concurrent states must be represented in non-overlapping parts of the configuration word. States that can never be concurrent, however, can share a portion of the configuration word. Therefore, it is possible to split the states into concurrent exclusivity sets, where the elements of each exclusivity set are pairwise non-concurrent. A statechart with a maximum concurrency of n requires n exclusivity sets. Our example has a maximum concurrency of six requiring six exclusivity sets (Figure 4). In [6], an algorithm is given to find the largest exclusivity set in a statechart. We propose a simpler greedy algorithm, which adds states to exclusivity sets as it encounters them. Drusinsky-Yoresh presents an encoding of 10 bit total length for his example. For comparison, our algorithm produces an encoding of 12 bit total length for that example. For our example, the state encoding step results in a code word of 4+3+1+3+3+2 = 16 bit.

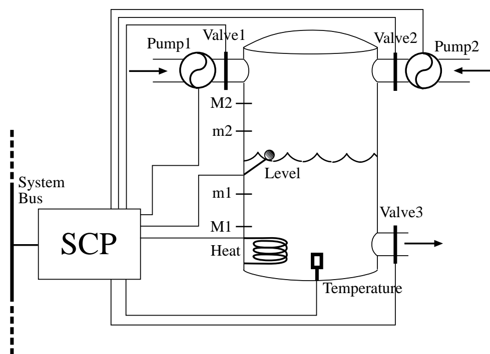

Figure 7. Process control application

Conditions are encoded by allocating one bit for each basic condition. This results in 10 bit for our example (Figure 4). The method to encode events uses exclusivity properties, similar to the state encoding: The events on a port can share the same portion of the event code word. Internal events of the statechart are exclusive to external events, because they occur in micro-steps. Concurrent internal events can only be generated by states in different exclusivity sets. For our example, the results of the event encoding algorithm are shown in Figure 4.

In [6], the problem of „unspecified inputs„ for statecharts is mentioned. Consider the example depicted in Figure 6, which has two exclusivity sets {A,B,Dummy} and {C,D,Dummy}. If E1 is received in configuration A-C, we expect B-C to be the next configuration. However, the SLA will generate B-Dummy, because no input was specified for the second exclusivity set. For statecharts, the detection of unspecified inputs is NP-complete in the number of inputs [4]. The trick here is not to try to determine a priori which transitions have to be activated by the SLA, but to detect after the fact which transitions actually were activated. For that purpose, we introduce a set of enable signals in the SLA. As every transition leads from and to basic states, every transition „influences„ one or more exclusivity sets. For the influenced exclusivity sets, the enable signals are set, for the others, they are reset. In the example, the transition on E1 generates an enable signal for the first exclusivity set, and the transition on E2 for the second. Thus, no enable signal will be generated for the second set. Therefore, the falsely generated dummy state will not be gated to the CR, and its original content, C, will remain intact. The enable signals can be computed easily from the statechart tree.

4. Example and results

As an example, we built a process control system for chemical engineering. Two chemicals are mixed in a tank and heated to a certain temperature. The product leaves the tank through an outflow valve. An overview of the application is shown in Figure 7. A part of the adjunct statechart model is presented in Figure 2. The SCP‘s ports control two pumps, three valves, measure the temperature, control the heater, and measure the level of fluid in the tank. Normally, the system runs in automatic mode, in which it keeps the level of liquid between the marks m1 and m2, and the temperature at 65 degrees Celsius. In emergencies, the operator sends a command to the controller to gain complete control over the system until the situation is corrected or the system is shut down.

The application´s SLA has 39 inputs and 71 outputs; the CR is therefore 39 bits wide. There are 10 condition inputs, 13 event inputs and 16 state inputs. The layout of the CR is shown in Figure 4. The SLA has 29 feedback outputs (13 events, 16 states), six enable outputs to the state parts of the CR, and 36 address outputs for six addresses. There are 51 transitions in the application, therefore six bits are required to encode the transitions. The implementation of the SLA takes 78 CLBs on a Xilinx®

XC4000. We generated a fast and a slow version for the implementation of the example´s TEP. Thus, the total size of the SCP varies between 324 and 375 CLBs, depending on the desired number of registers and custom instructions. Fig. 8 shows the schematic of a custom instruction generated for the fast implementation. The timing results of the application are summarized in Table 1.

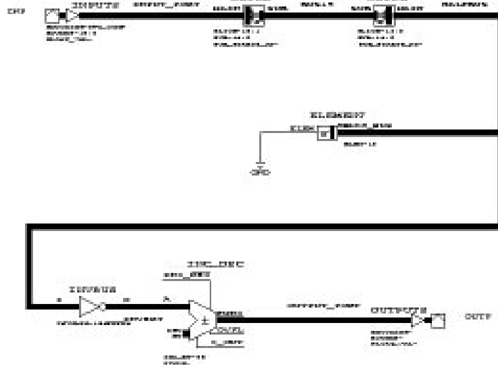

Figure 8. Custom instruction (-1/2*X+65).

Implementation	Fast	Slow
Shortest Transition (cycles)	9	21
Longest Transition (cycles)	330	441
Average Transition Length (cycles)	29	43
Total Instruction Length (cycles)	1473	2178
Critical Configur. Length (cycles)	854	1364
Hardware Size (CLBs)	375	324

Table 1. Process control system results.

5. Conclusion and future work

In this paper, we presented an approach to map hardware-software systems specified with extended statechart models onto a single-chip FPGA-based ASIP architecture. We found that our codesign method is suitable for application-specific control systems with real-time constraints. The results show that industrial applications, such as the control of a chemical process, fit on a single-chip FPGA. This is a significant result, because many industrial controllers can be replaced by a single-chip solution with our approach. We have enhanced existing statechart synthesis methods to support extended statecharts, and to produce semantically correct implementations. Currently, we work on supporting the history mechanism of statecharts. Future work also includes parallel implementations of the TEP, and improved timing analysis and area estimation.

6. References

[1] G. Berry, P. Couronne, G. Gonthier: Synchronous Programming of Reactive Systems: An Introduction to ESTEREL, Report Inst. Nat. Recherche Inf. Autom., Le Chesnay, France, March 1987

[2] D. Harel: Statecharts: A Visual Formalism for Complex Systems. Sci. Comp. Prog., vol. 8, pp 231-274, 1987.

[3] D. Harel, A. Pnueli, J.P. Schmidt, R. Sherman: On the Formal Semantics of Statecharts. Proc. Symp. on Logic in Comp. Sci., p. 54-64, 1987.

[4] D. Drusinsky: On Synchronized Statecharts. PhD Thesis, Dept. of Comp. Sci., The Weizmann Inst. of Sci., 1988.

[5] D. Drusinsky, D. Harel: Using Statecharts for Hardware Description and Synthesis. IEEE Trans. on CAD, vol. 8, pp 798-807, July 1989.

[6] D. Drusinsky-Yoresh: A State Assignment Procedure for Single-Block Implementation of State-charts. IEEE Trans. on CAD, vol. 10, No 12, December 1991.

[7] A. Pnueli, M. Shalev: What is in a Step: On the Semantics of Statecharts, Proc. Int. Conf. on Theor. Aspects of Comp. Softw., 1991.

[8] S. Narayan, F. Vahid, D. Gajski: System Specification with the SpecCharts Language, IEEE Design and Test of Computers, December 1992.

[9] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, A. Sangiovanni-Vincentelli: Sequential Circuit Design Using Synthesis and Optimization, Proc. ICCD´92, pp 328-333, 1992.

[10] A.Y. Alomari, M. Imai, J. Sato, N. Hikichi: An integer programming approach to the instruction set selection problem, IEICE Trans. Fundam. Electr. Commun. Comput. Sci. (Japan), vol. E76-A, no. 10, p. 1849-57, Oct. 1993.

[11] O. Haugen, R. Braek: Engineering Real-Time Systems. An Object-Oriented Methodology Using SDL, Prentice-Hall 1993

[12] M. Pilsl: TOS 1.5 Reference Manual, Siemens AG, Germany, November 1993.

[13] K. Buchenrieder, C. Veith: A Prototyping Environment for Control-Oriented HW/SW Systems Using Statecharts, Activity-Charts and FPGA‘s, Proc. Euro-DAC, 1994.

[14] D. Gajski, F. Vahid, J. Gong: A Binary-Constraint Search Algorithm for Minimizing Hardware During Hardware-Software partitioning, Proc. Euro-DAC, 1994.

[15] R. Leupers, P. Marwedel: Instruction set extraction from programmable Structures, Proceedings EURO-DAC '94 with EURO-VHDL '94, p. 156-6.

[16] C. Liem, T. May, P. Paulin: Instruction set matching and selection for DSP and ASIP code generation, Proc. of the EDAC-ETC-EUROASIC, Paris, France, 1994, p. 31-37.

[17] J. Sato, A.Y. Alomari, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, M. Imai: PEAS-I: A hardware/software codesign system for ASIP development, IEICE Trans. Fundam. Electron. Commun. Comput. Sci. (Japan), vol. E77-A, no. 3, p. 483-91, March 1994.

[18] B. Selic, G. Gullekson, P. T. Ward: Real-Time Object-Oriented Modeling, Wiley, 1994.

[19] J. Van Praet, G. Goossens, D. Lanneer, H. De Man: Instruction set definition and instruction selection for ASIPs, Proc. of the Seventh International Symposium on High-Level Synthesis, p. 11-16, 1994.