A Prototyping Environment for Control-Oriented HW/SW Systems Using State-Charts, Activity-Charts and FPGA´s

Dr. Klaus Buchenrieder
Christian Veith

Siemens Corporate R&D, Munich, Germany

Abstract

A method for quickly designing and implementing HW/SW systems is described. The method is based on the model of a flexible target architecture consisting of a dedicated controller that interacts with one or more processing elements. From a Statemate™ specification, the dedicated controller is extracted, and mapped onto a Xilinx™ FPGA. The software part of the specification is translated to C code. The C routines are modified by a set of tools to provide the interface between the software part and the hardware part of the system. A sample application illustrates the feasibility of the approach.

1: Introduction

State-charts [3, 7], a variation of the classical finite-state machine (FSM) model, are widely used for the specification of systems. Like conventional FSM´s, state-charts are based on states, events, and transitions. They differ from conventional FSM´s in several ways: They introduce hierarchical and concurrent states, and the formalism describing transitions between states is more elaborate than that of FSM´s. One notable difference between state-charts and FSM´s is the ability to reenter a previously visited state via a history mechanism. This feature actually goes beyond the realm of finite-state computational models, and will not be used in this paper.

The graphical representation of FSM´s – state-transition diagrams – is well known [6, 10]. State-charts were originally created as a visual formalism to overcome the sequential and flat nature of state-transition diagrams [7]. The state chart formalism became the basis for Statemate™, a tool from i-Logix, Inc., for the development of systems [8, 11]. As state-charts are primarily control-oriented, Statemate features activity-charts, a language similar to state charts, for the specification of functional and data-flow aspects of systems.

Statemate is routinely used by engineers to model reactive or embedded systems. Often the implementation of such systems consists of a mixture of hardware and software modules. Statemate descriptions are abstract, and also sufficiently implementation-independent. Statemate is therefore a natural choice for the specification part of a HW/SW Codesign process (furthermore, there are a variety of simulation and analysis methods available in Statemate, which we will not review in this paper).

On the implementation side, Statemate at first sight appears to be sufficiently well-equipped: It offers code generators for Ada, C, and VHDL. This is sufficient as long as pure software (C or Ada code) or pure hardware solutions (VHDL code, to be mapped to a hardware structure) are created. The result of the C and the Ada code generator needs a Unix runtime environment to function. The VHDL code generator produces algorithmic VHDL code that can be mapped to hardware using synthesis tools. Mixed HW/SW implementations, however, cannot be generated without considerable additional effort by designers, as Statemate is unable to generate the necessary interfaces between the two implementation domains.

Statemate´s code generators are primarily used to create visual interactive prototypes, which simulate the behavior of the system. They can be enriched with visual elements (panels) that give users and developers the "look and feel" of the specified behavior. Still, the goal of HW/SW Codesign is the creation of HW/SW systems, and based on our experience converting a Statemate prototype to a final HW/SW implementation is very time-consuming. Fortunately, Statemate is not a closed system, and allows the extraction of information from its data base. With the *Dataport* interface parts of the code generation step of Statemate can be replaced with custom tools, while retaining the possibility to create visual prototypes.

In this paper we propose a direct and automated way to generate the three major components of a mixed HW/SW implementation based on Statemate specifications: the hardware subsystem, the software subsystem, and the interface between the two. The HW subsystem is produced circumventing the code generation of Statemate. For the rest of the specification, C code is generated using the Statemate code generator. Additional tools provide the interface between the SW and HW subsystems. An FPGA

is synthesized from the description of the HW subsystem. The software runs on standard microprocessors; the processors and the controller speak to each other over a bus. The synchronization between the individual elements is achieved via interrupts. An overview of our method is shown in Fig. 1.

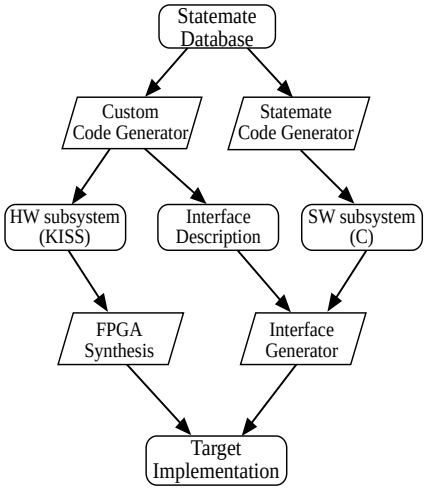


Fig. 1: Overview of the method

We aim at the development of embedded control systems, consisting of sensors, actuators, a dedicated controller, and one or more processing elements. Such control systems are applied in car electronics, medical equipment, or household appliances. To demonstrate our method, we have created an experimental platform that is identical to our architecture, except that the sensors and actuators are replaced with a serial I/O mechanism. The behavior of the environment of the control system is simulated by a PC. As an example, we built an automatic switchboard, which can handle several callers in parallel.

In section 2, we briefly discuss state-chart principles and describe the generation of FPGA-based controllers from Statemate descriptions. Section 3 gives an overview of our target architecture, and describes the generation of the HW/SW interface. In section 4 we present an example, and discuss the results we obtained. Section 5 summarizes our approach, and proposes future work.

2: From state-charts to FPGA´s

2.1: A short review of state-chart principles

Finite-state machines constitute the most successful formalism for the specification of control-oriented systems. Their simple visual representation, as well as their relation to regular expressions, have made them indispensable in the development of software and hardware systems. The long history in designing embedded controllers using FSM´s has led to a rich body of literature covering all aspects of FSM synthesis [1, 17]. The only drawback of FSM´s is their lack of structure: Their inherent flat and sequential nature makes the specification of complex systems a difficult task. Those limitations have been overcome by the introducing state-charts [7].

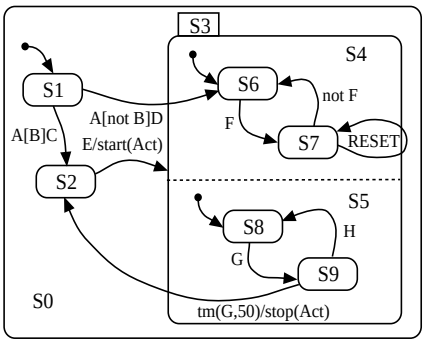


Fig. 2: A simple state-chart

State-charts are a visual formalism that lets designers specify hierarchy, concurrency, and also nondeterminisms. Consider the state-chart depicted in Fig. 2 where the top-level state S0 consists of three substates S1, S2, and S3. S1 and S2 are basic states, while S3 is further decomposed. S0 is called an OR-state; OR-states are used to structure a state-chart hierarchically. S3 is called an AND-state; AND-states are used to specify concurrency. AND-states are indicated by dashed lines dividing them into segments that run concurrently. S3, the only AND-state in the diagram, has two concurrent substates S4 and S5. Like conventional FSM´s, state-charts receive events, perform transitions, and produce outputs. The syntax and semantics of transition labels, however, has been enhanced. State-chart transitions are of the form *event[condition]/action* (e.g. the transition from S1 to S2). Apart from basic events, many special events can be used in a diagram, such as whether a state has been entered, or whether a condition has changed. The transition from S9 to S2, for instance, describes a time-out event. Hierarchical states are entered through "default connectors",which are denoted by arrows with a bubble attached to one end. When an OR-state is entered, the next state will be the one denoted by the default connector (S0 is entered through S1). When an AND-state is entered, all of its concurrent substates are entered through the default connector (S3 is entered through S6 and S8). If a concurrent substate is left, all of its concurrent states are exited, too. State-charts can be linked to the data-flow language of Statemate called activity-charts. State-charts can start, stop or resume activities via actions, and they can receive events from activities.

2.2: State-chart synthesis

From a Codesign point of view, a Statemate specification, consisting of state-charts and activity-charts, is almost perfect for the specification of systems. However, control is obviously separated from data. This somewhat violates the Codesign principle of completely implementation-independent specifications [2], because such a separation usually induces an implementation of control and data in distinct modules. In practical applications, however, this charateristic of Statemate specifications does not seem to affect the quality of implementations.

The Statemate code generators are not sufficient to create complete HW/SW implementations [13], because they do not synthesize HW/SW interfaces. We therefore developed a backend to Statemate that maps state-/activity-chart combinations to a simple target architecture shown in Fig. 5. It consists of a processor, a controller implemented as an FPGA, memory elements, and I/O compoents, which are connected by a bus. Since such architectures are based on the concept of separating control and data, we can exploit Statemate´s control/data separation by cutting out the control portion of a specification, and mapping it to an FPGA. This flexible architecture is suitable for many control applications that can be generated using a set of EDA tools, as shown in Fig. 3.

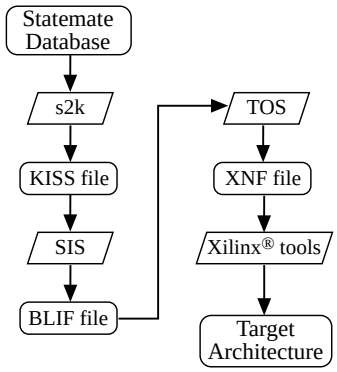


Fig. 3: State-chart synthesis process

For the extraction of the controller, we have developed *s2k*, a tool that uses Statemate´s *Dataport* interface [12]. a library of functions to query Statemate´s database. As the state-chart formalism induces a tree structure in a natural way, *s2k* recursively stores the state information in an internal tree format. For every node in the tree, a list is created representing the transitions leaving the node. The tree is successively flattened by replacing abstract AND- and OR-nodes with their respective substates. The result of the process is a list of basic (non-hierarchic, non-concurrent) states. The substitution of OR-states is straightforward (Fig. 4 (a)). The replacement of AND-states is more complex, as – according to state-charts semantics – entering an AND-state means entering all of its concurrent substates simultaneously. It is therefore necessary to replace AND-states by Cartesian products of their substates, as depicted in Fig. 4 (b).

When flattening the tree, transitions must be carefully rerouted to their appropriate new targets. When replacing OR-states, the process is straightforward. For AND-states, the sources and targets of transitions have to be modified to represent the compound states that originated from the original sources and targets of those transitions. Transitions leaving the AND-state must also be attached to the appropriate new compound states. In our current implementation, the tree is traversed to find all transitions leading into an AND-state, and those transitions must be modified accordingly. After the replacement of all AND- and OR-states, *s2k* removes any remaining nondeterminisms according to standard algorithms [10, 18].

The presented approach may suffer from state-explosion for highly concurrent state-charts with additional nondeterminisms. This problem is well-known, and the synthesis of hierarchically structured state-charts has been investigated in [4, 5], where the idea of flattening FSMs was dismissed. Pathological cases can be constructed [3], but for smaller examples the straightforward approach can be successful, as shown in our example. In [4], the direct mapping of the state-chart tree to a network of communicating FSM´s is proposed, but in [5], which builds upon the work of [4] the obvious timing problems that would result from such an approach are mentioned. The solution proposed in [5] resolves some of the problems, and thus suggests a baseline for future research.

From the resulting list of states, *s2k* (**s**tate **t**o **K**ISS) creates a table in KISS format [15, 16]. It uses a simple binary encoding for triggers and actions, and a one-hot encoding for conditions. The codes of triggers and conditions are combined to form larger words, which represent the actual inputs of the state table. For the states, *s2k* generates a one-hot encoding, which turned out to produce the best results in all our applications. Along with the state table, all codes are stored in a separate interface description file, which provides the basis for the generation of the HW/SW interface. The KISS file is converted to a BLIF (Berkeley Logic Interchange Format) [15] file using SIS, a tool developed at UCB [15, 16]. SIS interactively synthesizes and optimizes sequential circuits. The BLIF output is converted to an XNF file by TOS, a synthesis tool developed at the Technical University of Munich, Germany [14]. TOS has been especially designed for the synthesis of FPGA´s. It provides scripts to generate and optimize implementations for Xilinx™´s XC4000 family. TOS produces XNF netlists, which can be directly used by Xilinx´s XACT™ system[19, 20, 21].

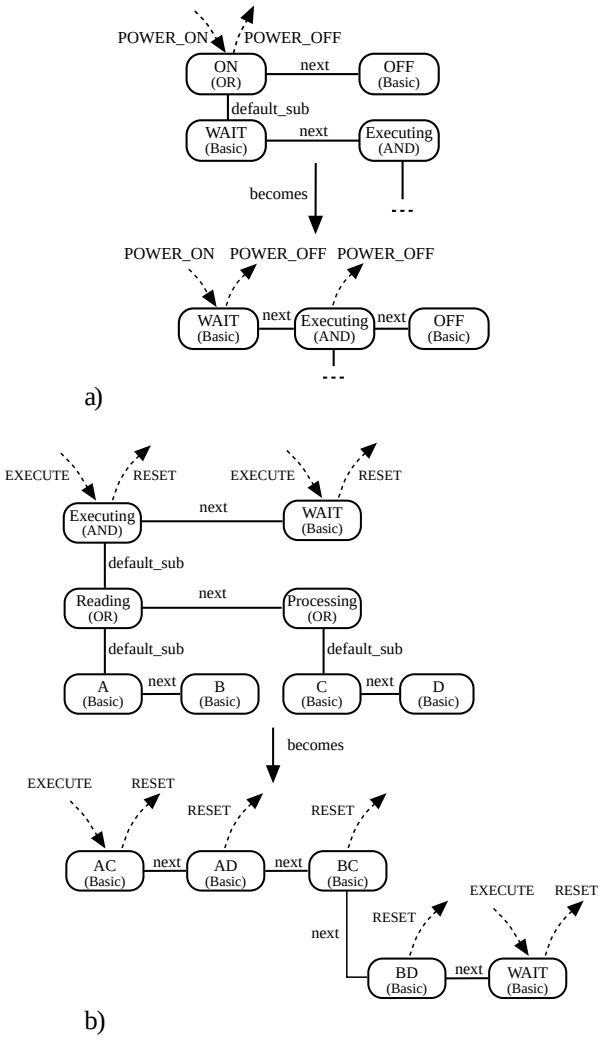


Fig. 4: Replacement of a) OR-states, b) AND-states

3: HW/SW aspects

3.1: The target architecture

An overview of the target architecture is shown in Fig. 5 a), its implementation is depicted in Fig. 5 b). It consists of a Xilinx™ XC4000 FPGA, which is the system´s controller, a Motorola 68008 processor [9], ROM and RAM memory banks, and serial and parallel I/O components. The components are connected by an eight-bit bus.

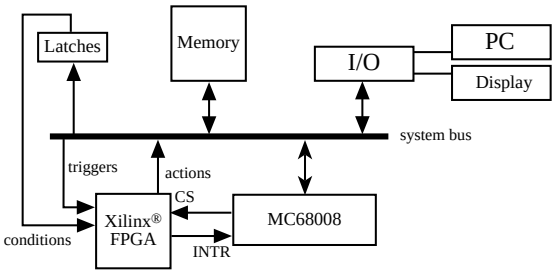
The controller has two sets of inputs; one is connected to the system bus, the other is connected to a special area in memory. The first set represents the triggers (events) of the system; they are valid for only one control cycle. The triggers are generated by software routines executed by the processor. When an action is finished, the processor enables the FPGA, and writes data on the bus. The other set of inputs represents conditions that are valid indefinitely. In software, conditions are implemented as boolean variables with fixed addresses; in hardware, conditions are implemented with latches that are also connected to the second set of inputs of the FPGA. The actual inputs of the controller are formed by concatenating elements of the two sets of inputs. Conditions are updated by software routines, and the asynchronous inputs change accordingly. As conditions are encoded one-hot, any compound condition consisting of basic conditions connected by *not*-, *and*-, and *or*-operators can be represented.

The output of the FPGA is eight bit wide, and – like the triggers – connected to the system bus. The output words represent the actions of the underlying state-chart. When it sends an action code on the bus, the controller interrupts the processor. An interrupt request service routine switches to the appropriate action routine. The action routine performs the desired action, and returns the next valid word for the controller. Our architecture uses two levels of interrupts, one for I/O, the other for the HW/SW interface. During the execution, conditions might be updated. They are immediately available to the FPGA, but are not consumed before the next control cycle.

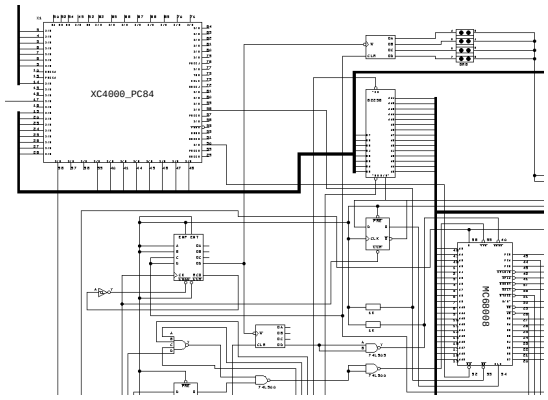
In its current form our target architecture only supports eight different basic conditions, which limits the size of state-charts that can be transformed to FPGA´s considerably. The number of latches can, however, be easily incremented. It is only limited by the available pins on the FPGA. For small packages (84 pins, of which 16 are used for power supply; three more reserved for chip-select, processor interrupt, and a data-valid signal), 64 pins can be used for I/O. Of these, eight could be used for triggers, and eight for outputs (giving a total of each 255 representable triggers and actions). This leaves a maximum of 48 basic conditions that could be used in state-charts, certainly enough for many embedded control systems.

3.2: The HW/SW interface

The FPGA synthesis process results in a fully functional Xilinx XC4000 chip. External input and output signals are connected via tri-state buffers defined by *s2k*. It assigns unique binary encodings to all events, conditions, and actions of the specification. The encodings with their symbolic names are stored in an interface description file.For the activity-chart(s) of the specification, we use the Statemate code generator [13]. Code for activities can be generated in the form of independently running tasks, or in the form of (self-terminating) procedures. Our target architecture obviously implies the second method. Therefore, actions that stop or resume activities are prohibited. Statemate generates a header file containing data definitions and C prototypes for every routine, and C source files for the actual implementation of activities.



a) Overview



b) Implementation

Fig. 5: Target architecture

Parsing the description file with the interface generator *ig*, we separate the HW/SW interface into a HW-to-SW and a SW-to-HW part. *ig* first generates the SW-to-HW part, producing a boolean variable for every condition it encounters. For the triggers, an enumeration data type is created. Further, a code segment in the form of a CASE-statement is generated that maps the symbolic triggers to corresponding binary codes. A predefined routine writes binary sequences (code words) on the system bus, where they are read by the controller. Every action routine the processor performs produces a trigger, which can be one of the triggers defined in the description file, or a null trigger. Our current implementation is restricted to just one resulting trigger per action routine, as the generation of several events "at once" would inevitably enforce a queuing mechanism to avoid losing triggers. If a null trigger occurs, the controller can still perform transitions based on the set of inputs representing conditions. For those transitions, the trigger inputs are treated as don't-cares. Finishing the SW-to-HW part, return sequences for the action routines produced by Statemate are generated.

In the second part of the process, *ig* possibly has to generate additional routines. These originate from basic actions in the state-chart, such as changing the value of a condition. As the controller cannot provide this functionality, the processor must carry out a corresponding routine. Then every routine is associated with its binary code in the description file. These codes are used to interrupt the processor. When the processor receives an interrupt, it retrieves the code of the current action from the bus, and interprets it as the address of an interrupt routine. In addition, *ig* generates a ´main routine´ that runs when the system is idle otherwise, and listens for external events. This routine is also used for the handling of I/O interrupts.

4: Applying the method

To demonstrate our method, we selected an automatic switchboard as an example. It routes external callers to recipients on an internal phone network consisting of, e.g., technical help desks, an information desk, and a marketing desk. The switchboard can handle several callers in parallel. It automatically responds to external calls, and possibly puts callers on hold if it is just busy handling another call. During heavy-load times, several callers might be queued that way. The other part of the switchboard plays a message asking the caller to enter another digit, where every digit represents a different internal connection. Based on that key, an internal connection is established. Fig. 6 shows the model of the switchboard. The two main parallel states represent the two basic functions of the system, answering external calls, and making internal connections. The model is simplified in that an infinite supply of internal connections is assumed.

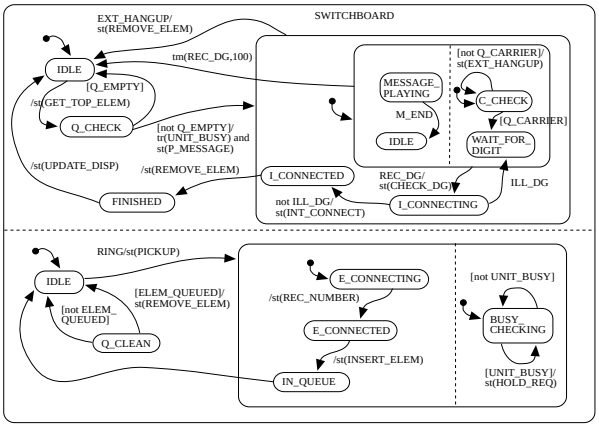


Fig. 6: State-chart of Switchboard example

The implementation of the model had to be simplified: As we cannot build a real phone network, we simulated it with a PC. Random requests of a simple program on the PC simulate external callers. Internal connections are simulated by updating a bar chart on a simple display that is connected to the parallel port of the target architecture. The bar chart shows the number of callers connected to the various internal desks.

Converting the state-chart of Fig. 6 to KISS format, *s2k* generated an FSM with 45 states, 9 input bits and four output bits. The nine input bits are the result of five basic conditions and the encoding of seven events plus the null event. The four output pins originate from the encoding of 12 actions plus the null action. From the KISS file, SIS [16] produced a network of 168 nodes and 766 literals. TOS [14] converted the resulting BLIF file to an XNF netlist consisting of 115 CLB´s. Both SIS and TOS achieved considerable optimizations of the network.

Based on the associate activity-chart (which is not shown here), the Statemate code generator produced 12 routines corresponding to the activities that are started from the state-chart. *ig* generated five boolean variables for the basic conditions, the code to handle triggers in both binary and symbolic form, and the return sequences for the routines generated by Statemate. The state-chart shows that no additional routine had to be generated, as the only action not starting an activity, tr(UNIT_BUSY), is associated with starting an activity, and therefore a routine had been generated already.

5: Conclusion and future work

We have shown that it is possible to generate HW/SW systems from a specification consisting of state- and activity-charts. In certain aspects, our method is more powerful than the currently available code generators for state-/activity-charts, as it not only generates hardware *and* software implementations, but also the HW/SW interface. On the other hand, it is less flexible than the software prototypes of Statemate, and less efficient than the state-chart synthesis method proposed in [5]. The method of [5], however, can – in contrast to our method – not represent conditions in state-charts.

Regarding future work, we will further investigate whether additional state-chart features not considered in out current implementation can be mapped to FPGA´s. As our method suffers from state explosion for complex state-charts, we will also look for more efficient ways to synthesize state-charts. We will also enhance our method to cover MIMD multi-processor systems by adding a queuing mechanism for triggers to our target architecture.

6: References

[1] R. K. Brayton, G. De Micheli, A. Sangiovanni-Vincentelli: Optimal State Assignment for Finite State Machines, IEEE Trans. on CAD, vol. CAD-4, pp 269-285, July 1985

[2] K. Buchenrieder, G. de Micheli, P. A. Subrah-manyam: Hot Topics Section, IEEE Computer, vol. 26, No 1, pp 84-87, January 1993

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

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

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

[6] E. M. Gurari: Introduction to the Theory of Computation; Computer Science Press, 1989

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

[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring:: STATEMATE: A Working Environment for the Development of Complex Reactive Systems, Proc. 10th Int. Conf. on Soft. Eng., New York, IEEE Press, April 1988, pp 396-406.

[9] W. Hilf, M. Nausch: The M68000 Family, Vol. 1&2, te-wi Publishers, Germany, 1990 (in German)

[10] J. E. Hopcroft, J. D. Ullman: Introduction to Automata Theory, Languages, and Computation; Addison-Wesley, 1979

[11] i-logix, Inc.: Statemate 5.0 User and Reference Manual. Burlington, MA, June 1993.

[12] i-logix, Inc.: Statemate 5.0 Dataport Reference Manual. Burlington, MA, June 1993.

[13] i-logix, Inc.: Statemate 5.0 Code Generator User Reference Manual. Burlington, MA, June 1993.

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

[15] 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.

[16] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, r. Murgai, A. Saldanha, H. Savoj, P. R. Stephan R. K. Brayton, A. Sangiovanni-Vincentelli: SIS: A System for Sequential Circuit Synthesis, Memorandum UCB/ERL M92/41, UCB, May 1992.

[17] J. D. Ullman: Computational Aspects of VLSI. Comp. Science Press, Rockville, MD, 1984.

[18] D. Wood: Theory of Computation. Harper & Row Publishers, New York, 1987.

[19] Xilinx, Inc.: XACT Development System: Design Interface and User Guide, San Jose, CA, Aug. 1991.

[20] Xilinx, Inc.: XACT Reference Guide, Vol. 1 & 2, San Jose, CA, October 1992.

[21] Xilinx, Inc.: The XC 4000 Data Book, San Jose, CA, August 1992.