Hardware Synthesis from Requirement Specifications

Konrad Feyerabend
Dept. of Computer Science
Oldenburg University
D-26111 Oldenburg
Feyerabend@Informatik.Uni-Oldenburg.DE

Rainer Schlör
OFFIS
D-26121 Oldenburg
Rainer.Schloer@OFFIS.Uni-Oldenburg.DE

Abstract

This paper describes the theory and implementation of a novel system for hardware synthesis from requirement specifications expressed in a graphical specification language called Symbolic Timing Diagrams (STD). The system can be used together with an existing formal–verification environment for VHDL leading to a novel methodology based on the combination of synthesis and formal verification. We show the feasibility of the approach and experimental results obtained with the system on the well known example of an industrial production cell, where both FPGA and ASIC hardware implementations were successfully synthesized.

1. Introduction

The correct design of complex systems has become an increasingly important issue. One major research topic in this field is formal verification of system properties based on a technique called (symbolic) model checking which requires two independent levels of specification: (1) an operational specification, written e.g. in VHDL or Statecharts, and (2) an abstract, typically declarative specification, written e.g. as temporal–logic specification. The abstract specification is used to express critical *system requirements* (referred to as requirement specification).

Within the FORMAT project [7] a research team at the institute OFFIS has developed a collection of tools, which allow to verify (independently developed) VHDL designs against requirement specifications, written in a graphical language called Symbolic Timing Diagrams (STD).

While this tool suite constitutes a powerful prover for the a–posteriori verification of critical properties of existing VHDL designs, a companion project has been targeted towards high–level synthesis of prototype–implementations directly out of the graphical specification language STD (**i**nterface **co**ntroller **s**ynthesis and verification system, ICOS). The first goal of the ICOS–project was to be able to handle the special case of control–dominated specifications; a case where such specifications are exclusively used is the well known example "production cell" [11].

The main theoretical results underlying the ICOS–system have been reported in [9], and an evaluation report about the application of ICOS on the "production cell" case study appeared in [11].

This paper reports the main results of [4]. This work has developed both the theoretical basis and the implementation for the synthesis of VHDL code in ICOS. From a given STD requirement specification, VHDL code in a particular style can be generated and synthesized by standard silicon compilers (e.g. Synopsys).

Besides this achievement, the work of [4] bridges the gap between ICOS and the VHDL prover described above: A synthesized VHDL controller can be combined with manually designed ones, and the combined result can be verified against STD–requirements. Furthermore, VHDL components synthesized from STD–specifications can be transformed (e.g. to obtain particular optimizations or generalizations), and the transformation can be verified against the initial specification of the synthesis process.

Related work. All known approaches to use timing diagrams in a formal sense ([1, 6, 12, 5]) differ from our approach in that they have built-in means to specify control structures such as iteration and concatenation (sequencing), while the declarative semantics of the STD language associates with each timing diagram a *constraint* on the set of admissible behaviors of a component. The main consequence is that STD is ideally suited for requirement specification and allows an incremental development of specifications.

The work done on synthesis from Propositional Temporal Logic using ω-automata [13] differs from our approach in that we restrict ourselves to specifications expressible by *deterministic* Büchi automata yielding a much better time and space complexity of our algorithm. Moreover, we prohibit outputs from being dependent from the actual inputs which allows powerful partitioning techniques (see below).

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

The rest of the paper is organized as follows: Section 2 explains the requirement specification language STD. Section 3 describes the complete synthesis–path from requirement specification down to VHDL code. Section 4 is a detailed technical description of the semantic foundation and the generation of optimized Moore–automata, which is the intermediate format from which synthesizable VHDL code is generated. Section 5 gives the experimental results obtained for the case study "production cell", which clearly demonstrates the feasibility of our approach. Finally, section 6 summarizes the achievements and identifies directions of future work.

2. Requirement Specification Language

We now briefly recall the case–study "production cell" (PC) and the graphical specification language STD (the reader familiar with [11] or [10] may skip the rest of this section). The PC is composed of a feed belt that transports metal blanks to an elevating rotary table which brings the blanks into place for the first arm of a robot. The robot picks up the blanks from the table and moves them to a press where they are forged, put out by the robot's second arm, and deposited from the PC by a deposit belt and a crane (figure 1).

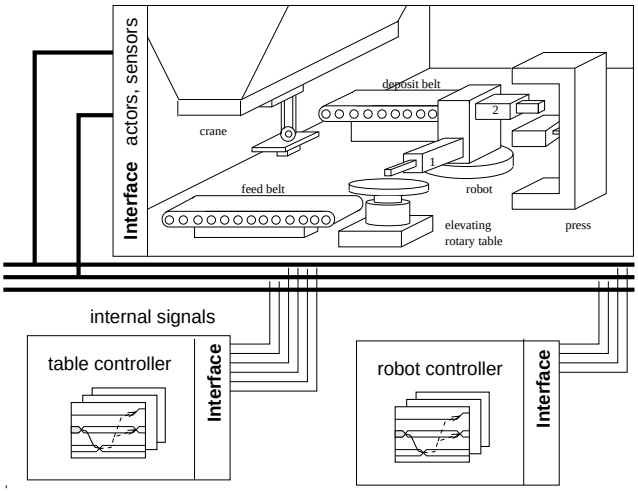

Figure 1. Outline of the production cell.

The STD–specification describes the behaviour of a *distributed controller* of the PC. Given the natural decomposition of the PC into separate entities the specification can be partitioned into according sub-specifications (for the feed belt, the table, etc.). This is possible since in our specification style we are able to distinguish between constraints to be guaranteed by the controller and assumptions made on the behaviour of other components or the physical environment. The controllers have to react on signals from sensors that observe the state of the PC and on internal signals from the controllers of the other components.

An STD–specification consists of a collection of individual (STD–)diagrams, interpreted as a collection of constraints with standard conjunctive interpretation (see [2] for an in–depth introduction to the language STD and its formal semantics).

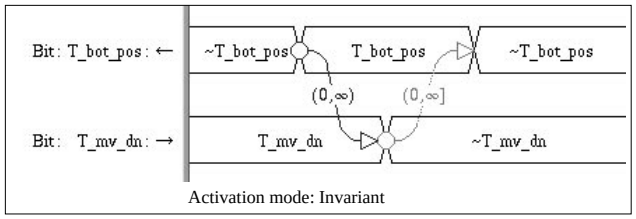

Figure 2. An example STD

An example diagram is given in figure 2. It states that *whenever* the table is not in its bottom position (denoted by the - negated - atomic proposition *T_bot_pos*, set by a sensor attached to the table) and is moving down (denoted by the atomic proposition *T_mv_dn*, which is an actuator–signal controlling the vertical table movement) it should keep on moving down until the sensor signals that the bottom position is reached. Then the movement of the table must be stopped, expecting that the sensor signal *T_bot_pos persists* to hold until the movement stops.

Note that this specification style does not permit explicit definition of real time constraints. The controller has to react "fast enough" to stop the movement of the table before it continues to move down too far (i.e. leave the range where the sensor signals *T_bot_pos*). In the hardware implementation we guarantee this behaviour by an *as soon as possible scheduling* of pending reactions, which can be further analyzed to obtain rigid timing information (in this case about the delay between recognition of the sensor–value and the actuator–response).

3. Synthesis Path

The following steps are taken during synthesis (cf. figure 3; steps 1 to 5 (extended FSM) are discussed in [9] in detail).

1. Each STD of the specification ($STD_1 \ldots STD_n$) is compiled into an ω-automaton A_i that accepts the requirements expressed in the STD. We use Büchi automata [14] for internal representation of the STD. A Büchi automaton $A = (Q, ed, \Delta, q_0, F)$ is a finite automaton on infinite words over valuations V_{ed} of the VHDL-alike entity declaration[1] $ed = (I, O, typedecl)$. Q is a finite set of states and q_0 the

[1] I and O are the in- and outports of the entity declaration; $typedecl : I \cup O \rightarrow DOM_{VHDL}$ maps each port to a finite type

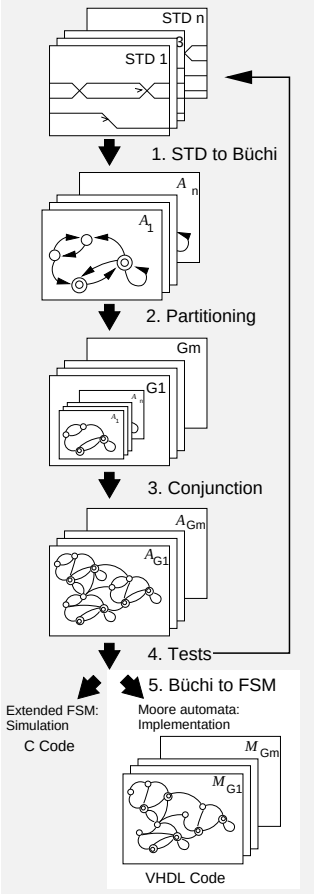

Figure 3. Synthesis path

start state of the automaton. Each transition from the transition relation $\Delta \subseteq Q \times V_{ed} \times Q$ is labeled with a valuation of the ports defined in *ed*. The automaton accepts an word α from $(V_{ed})^\omega$ ($\alpha \in [\![A]\!]$) if there exists a run for α that reaches a state from the set of accepting states $F \subseteq Q$ infinitely often. Figure 4 shows the Büchi automaton that represents the STD of figure 2.

2. The specified component must satisfy the requirements of all its STD. To overcome a state explosion problem that would arise if we compute the conjunction of the automata A_i directly we partition the automata into independent groups. The outports O of the component are partitioned into groups $O_1 \ldots O_k$ such that each automaton from $A_1 \ldots A_n$ only restricts outports from one group O_i. The automata restricting the same set of outports are then grouped together ([9] describes partitioning in detail).

3. Using the standard operation for the *cross product* of Büchi automata we generate a group automaton A_{G_i} for each group G_i such that $A_{G_i} = \bigcap_{A \in G_i} A$

4. This step checks for consistency and completeness of the specification. The specification should be satisfiable, the

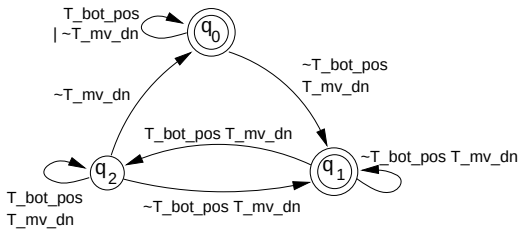

Figure 4. The Büchi automaton compiled from the STD depicted in figure 2

values assigned to the outports should not depend on the values read from the inports at the same moment, and the specification may not make assumptions on the future valuation of the inports at any time. [9] gives the formal criteria to check and shows that consistency of the overall specification can be tested by examining the group automata.

5. In this last step the group automata are transformed into different kinds of FSM, depending on the purpose of the synthesis. These FSM are then executed in parallel.

(A) For simulation each group automaton is transformed to an "extended FSM". These are FSM F_{G_i} that may choose between several possible outputs in every state and that respects the acceptance condition of the group automaton A_{G_i}.

(B) For implementation each group automaton $A_{G_i} = (Q, ed, \Delta, q_0, F)$ if reduced to a Moore automaton $M_{G_i} = (Q_M, ed, \delta, \lambda, q_0)$ that implements an *as soon as possible scheduling* of outport signals.

Again, Q_M is a finite set of states and q_0 the start state of the automaton. The transition relation $\delta \subseteq (Q_M \times V_I \times Q_M)$ is labeled with values assigned to the inports of the automaton. The output function $\lambda : Q_M \to V_O$ maps every state of the automaton to one outport valuation. There is no acceptance condition; a word $\alpha \in (V_{ed})^\omega$ is accepted by the automaton ($\alpha \in [\![M]\!]$) if there is a run $r(0)r(1)\cdots \in (Q_M)^\omega$ of the automaton such that for all i in ω it holds: $\alpha(i) \restriction V_O = \lambda(r(i))$ and $(r(i), \alpha(i) \restriction V_I, r(i+1)) \in \delta$.

The Moore automata generated satisfy the following properties:-

soundness The behaviour implemented is valid with respect to the specification:

$$[\![M_{G_i}]\!] \subseteq [\![A_{G_i}]\!]$$

responsiveness During execution the Moore automaton can react on any input value at any time:

$$\forall v_i \in V_I : \forall q \in Q_M : \exists q' : (q, v_i, q') \in \delta$$

(Note that these conditions ensure that a nontrivial subset of the specification is implemented: $[\![A_{G_i}]\!] \neq \emptyset \implies [\![M_{G_i}]\!] \neq \emptyset$.)

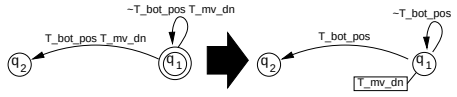

Figure 5. Translation of q_1

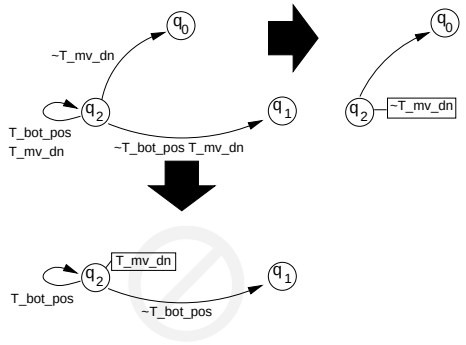

Figure 6. (Not) allowed translations of q_2

The resulting FSM M_{G_i} are then coded in synthesizable VHDL such that they execute in parallel. To this means we add a clock *clk* to the entity declaration of the component that synchronizes the controllers. In the next section we present this last step of synthesis for implementation in detail.

4. Synthesis of Moore automata

The idea underlying our approach for the Moore automaton synthesis is to translate each state q_b of the Büchi automaton to one state q_m of the Moore automaton. The outports are extracted from the transition relation and inserted into the output function λ of the Moore automaton. Thus the translation of state q_1 in figure 4 is straightforward as depicted in figure 5.

But translation is not always that easy, since we have to guarantee soundness and responsiveness of the Moore automaton by construction. For example, in state q_2 there are two possible valuations of the outports. We may either deassert *T_mv_dn* or keep it asserted (figure 6). But if we keep the signal asserted we violate the soundness requirement, since the controller then implements behaviour that is forbidden by the specification.[2]

Accordingly, we have to choose the outport valuation in a way that guarantees reaching an accepting state of the Büchi automaton after a finite number of transition steps for every valuation of the inports.

[2]If we never deassert *T_mv_dn* the Büchi automaton does not leave q_2 which is a non-accepting state. Thus this run is not accepted.

4.1. Cost function

We formalize this criterion by a function $st\_cst : Q \to \mathbb{N} \cup \bot$ on the states of the Büchi automaton. It measures the maximal number of transition steps until we can guarantee reaching an accepting state. $st\_cst(q) = \bot$ indicates that we can't warrant acceptance starting from q. In this case, no matter how we choose the outport valuation v_o there is always an input valuation v_i that prevents us from reaching an accepting state. This may either be the case if there is no transition labeled with $v_i v_o$ leaving q ($\nexists q' : (q, v_i v_o, q') \in \Delta$; this violates responsiveness) or if the transition reaches a state that has undefined costs as well ($\forall q' : (q, v_i v_o, q') \in \Delta : st\_cst(q') = \bot$), violating soundness).

Function st_cst is defined recursively as depicted in figure 7. First, we need to collect all the states already visited during computation to detect loops on non-accepting states (such loops are not permitted and thus count as $\bot$). This collection is done in the second parameter to function $stc\_ext : Q \times 2^Q \to \mathbb{N} \cup \bot$. The cost of a state is calculated as the minimum cost of all output values possible ($out\_cst : Q \times V_O \times 2^Q \to \mathbb{N} \cup \bot$). Finally, the cost in a state q for a given outport valuation v_o is defined as the maximum of the costs of all transitions from q that are labeled with v_o and any inport valuation ($tr\_cst : Q \times V_{ed} \times 2^Q \to \mathbb{N} \times \bot$).

The function $ok\_outs : Q \to 2^{V_O}$ gives the valuations of the outports permitted to guarantee a minimal number of steps until reaching an accepting state.

$$ok\_outs(q) \stackrel{\Delta}{=} \{ v_o \in V_O \mid out\_cst(q, v_o, \{q\}) = st\_cst(q) \}$$

4.2. Synthesis function

The synthesis function $b2m$ that maps Büchi automata to Moore automata is defined as $b2m : (Q, ed, \Delta, q_0, F) \mapsto (Q', ed', \delta, \lambda, q_0')$ with

- Neither set of states nor the entity declaration is modified: $Q' = Q$, $ed' = ed$, $q_0' = q_0$
- For the output function we choose a valuation from *ok_outs*: $\forall q \in Q' : \lambda(q) \in ok\_outs(q)$
- Only the transitions of the Büchi automaton that are labeled with the outport valuation chosen are translated: $(q, v_i, q') \in \delta \Leftrightarrow \exists (q, v_i v_o, q') \in \Delta : \lambda(q) = v_o$

Provided all states of the Büchi automaton have finite cost this translation guarantees soundness and responsiveness of the Moore automaton generated:

THEOREM 1 (CORRECTNESS BY CONSTRUCTION)
Given a Büchi automaton $A = (Q, ed, \Delta, q_0, F)$ such that $\forall q \in Q : st\_cst(q) \in \mathbb{N}$. Then $b2m$ guarantees soundness

$$st\_cst(q) \triangleq stc\_ext(q,\{q\})$$

$$stc\_ext(q,Q_1) \triangleq \min_{\perp}\{n \mid n = out\_cst(q,v_o,Q_1), v_o \in V_O\}$$

$$out\_cst(q,v_o,Q_1) \triangleq \begin{cases} \max\{tr\_cst(q,v_iv_o,Q_1) \mid v_i \in V_I\} \\ \quad \forall v_i \in V_I : tr\_cst(q,v_iv_o,Q_1) \in \mathbb{N} \\ \perp \quad otherwise \end{cases}$$

$$tr\_cst(q,l,Q_1) \triangleq \begin{cases} 1 & \exists q' : (q,l,q') \in \Delta, q' \in F \\ n & \exists q' : (q,l,q') \in \Delta, q' \notin F \cup Q_1 : \\ & \quad stc\_ext(q',Q_1 \cup \{q'\}) = n-1 \\ \perp & otherwise \end{cases}$$

where $\min_{\perp} : 2^{\mathbb{N}\cup\perp} \to \mathbb{N} \cup \perp$ is defined as

$$\min_{\perp}(A) = \begin{cases} \perp & A = \perp \\ \min(A \restriction \mathbb{N}) & otherwise \end{cases}$$

Figure 7. Recursive definition of the cost function

and responsiveness by construction:

$$[\![(b2m(A))]\!] \subseteq [\![A]\!]$$
$$\forall q \in states(b2m(A)) : \forall v_i \in V_I : \exists q' : (q,v_i,q') \in \delta$$

4.3. Coding in VHDL and execution

Before coding the Moore automata in VHDL they are minimized using the standard operations on FSM to eliminate equivalent states. This may cause an enormous reduction of states and transitions as shown below.

The coding of the Moore automata into VHDL is straightforward. We use a coding of the FSM in three processes. Both one-hot and binary encoding of the state register is supported. The FSM M_{G_i} generated from the group automata A_{G_i} are grouped into one VHDL model, each FSM in one VHDL block statement, and executed in parallel. This parallel execution is sound with respect to the overall specification.

THEOREM 2 (CORRECTNESS OF PARALLEL EXECUTION) Given a requirement specification by a set $STD_1 \ldots STD_n$ that is compiled to Büchi automata $A_1 \ldots A_n$ and grouped into $A_{G_1} \ldots A_{G_m}$ by the synthesis steps 1 to 4 introduced above. Then the parallel execution[3] of the Moore automata $M_{G_i} = a2m(A_{G_i})$, $i \in 1 \ldots m$ is correct with respect to the specification.

$$[\![\underset{i\in 1\ldots m}{\text{par}} (M_{G_i})]\!] \subseteq [\![\bigcap_{j\in 1\ldots n} A_j]\!]$$

$$[\![\bigcap_{j\in 1\ldots n} A_j]\!] \neq \emptyset \implies [\![\underset{i\in 1\ldots m}{\text{par}} (M_{G_i})]\!] \neq \emptyset$$

[3]The parallel execution "par" of Moore automata is defined as obvious: Each automaton starts in its start state, transitions are performed in parallel. Cf. [4] for details.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity T_Vmove is
  port( clk, rst : in std_logic;
        T_load_loaded : in  std_logic ;
        T_pick_picked : in  std_logic ;
        T_V_bot_pos : in  std_logic ;
        T_V_top_pos : in  std_logic ;
        R_A1_safe4T : in  std_logic ;
        R_A1H_Un : in  std_logic ;
        T_V_Up : out  std_logic := '0'  ;
        T_V_Dn : out  std_logic := '0'  );
end T_Vmove ;

architecture behaviour of T_Vmove is
begin
fsm_0 : block
  type states is (S0,S1,S2,S3,S4);
  signal state, next_state : states;
begin
  -- state machine:
  process (state,R_A1H_Un,R_A1_safe4T,T_V_bot_pos,
           T_V_top_pos, T_load_loaded, T_pick_picked)
  begin
    -- default value prevents latch on state
    next_state <= state;

    case state is
      when S4 =>
        if  (((( T_V_top_pos = '0')))) then
          next_state <= S3 ;
        end if;
      when S3 =>
        [...]
    end case;
  end process;

  -- outputs
  process (state)
  begin
    -- default values are '0'
    T_V_Dn <= '0';
    T_V_Up <= '0';
    case state is
      when S4 =>
        T_V_Up <= '1';
      when S3 =>
        [...]
      when others => -- no output is set to '1'
    end case;
  end process;
  -- sequential circuit: build flip flops
  process (clk, rst)
  begin
    if rst = '0' then
      state <= S0;
    elsif clk ='1' and clk'event then
      state <= next_state;
    end if;
  end process;
end block fsm_0 ;

end behaviour;
```

Figure 8. Automatically synthesized controller of the table (vertical movement)

The VHDL generated may be used with commercial RT level synthesizers to generate hardware. Figure 8 shows an (abridged) example VHDL model as created by the ICOS tools, the controller for the vertical movement of the table, with binary coded state register. The ports *clk* and *rst* are added to the component's entity declaration to control the state register.

5. Experimental results

The theory presented in the last section has been implemented and integrated into ICOS. The tools have been tested on different specifications. Using the partitioning techniques described above the specification of the production cell is partitioned into 26 sub-specifications. The ICOS tools generate behavioural VHDL models from these sub-specifications and a structural description that links the components of the specified system.

The biggest group automaton (the vertical movement of the press) is generated from 5 STD and has 27 states and 193 transitions. It is synthesized in 8 seconds to a Moore automaton with 6 states and 26 transitions. The resulting VHDL models were then synthesized using the Synopsys tools and mapped to an Altera FLEX 800 FPGA using Altera MAX+plus II. This controller of the production cell runs at a clock speed of 14 MHz. Alternatively, we synthesized and mapped the models with the Cadence tools on

a 1,0μ ES2 library to generate an ASIC consisting of 599 standard cells.

To get a feeling on how the tools will perform on more complex specifications we did a synthesis of the production cell components without partitioning by skipping synthesis step 2. Figure 9 shows the results of this synthesis. The

Entity	# STD	# ports	Büchi # states	Büchi # trans.	Moore # states	Moore # trans.	mem (MB)	time (s)
Feed belt	9	15	57	483	20	64	< 2	9,36
Table	16	18	425	12613	45	181	24,48	220,13
Press	14	17	1703	115897	82	435	169,44	4483,43
Robot								
Arm1	8	15	61	663	15	60	4,46	29,37
Arm2	8	17	56	557	15	60	4,31	31,96
Rotation	5	14	11	43	5	11	< 2	4,1
Dep. belt	15	17	198	2861	43	162	8,91	77,12
Crane	16	14	1574	76041	190	1303	81,16	1035,07

Figure 9. Synthesis results for the production cell specification (not partitioned)

reduction gained is enormous (over 99% of the transitions are eliminated for the bigger components). This shows the necessity to reduce the complexity of the specification before implementing it and the feasibility of out approach to hardware synthesis from requirement specifications.

6. Conclusion and future work

The work reported in this paper has implemented a generator for synthesizable VHDL code out of requirement specifications written in a graphical language (STD). The result is a rapid–prototyping environment for control–dominated applications, using a graphical language based on timing diagrams, which is familiar to the hardware designer. Furthermore, the work opens a new methodology, since the approach can be combined with existing work on the formal verification of VHDL designs against STD–specifications.

The work also leads to a new line of research: While the specification language STD makes only qualitative timing assertions, more rigid (quantitative) timing information can be obtained using standard analysis techniques for the synthesized controller. This allows to develop with this method even prototype controllers which have to meet rigid timing constraints, which is a topic of actual interest (cf. [3]). A formal framework for the sound transition from abstract (causality–based) specification to concrete (rigid–timed) implementation is yet to be developed.

Acknowledgement. The ICOS–project was initiated by F. Korf who also provided the implementation of the original kernel of ICOS as reported in [8] and supported our work through many discussions.

References

[1] G. Boriello. Formalized timing diagrams. In *Proceedings, The European Conference on Design Automation*, pages 372–377, Brussels, Belgium, Mar. 1992.

[2] W. Damm, B. Josko, and R. Schlör. Specification and verification of VHDL-based system-level hardware designs. In E. Börger, editor, *Specification and Validation Methods for Programming Languages and Systems*. Oxford University Press, 1995.

[3] Deutsche Forschungsgemeinschaft. Schwerpunktprogramm 'Rapid Prototyping für integrierte Steuerungssysteme mit harten Zeitbedingungen, 1996.

[4] K. Feyerabend. Schaltungssynthese aus Zeitdiagrammen: Eine Kopplung des ICOS–Systems mit Synopsys. Diplomarbeit, Universität Oldenburg, 1996.

[5] W. Grass, C. Grobe, S. Lenk, W. Tiedemann, C. D. Kloos, A. Marin, and T. Robles. Transformation of timing diagram specifications into VHDL code. In *Conference on Hardware Description Languages and their Applications*, September 1995.

[6] K. Khordoc, M. Dufresne, E. Cerny, P. Babkine, and A. Silburt. Integrating behavior and timing in executable specifications. In *Conference on Hardware Description Languages and their Applications*, pages 385 – 402. OCRI Publications, April 1993.

[7] C. Kloos, J. Goicolea, and W. Damm, editors. *Formal Methods for Hardware Verification*. Springer-Verlag, 1996.

[8] F. Korf. *System-level Synthesewerkzeuge : von der Theorie zur Anwendung*. Dissertation (to appear), Universität Oldenburg, 1996.

[9] F. Korf and R. Schlör. Interface controller synthesis from requirement specifications. In *Proceedings, The European Conference on Design Automation*, pages 385–394, Paris, France, feb. 1994. IEEE Computer Society Press.

[10] F. Korf and R. Schlör. Symbolic timing diagrams / synthesis of a production cell controller using symbolic timing diagrams. In C. Lewerenz and T. Lindner, editors, *Formal Development of Reactive Systems, LNCS 891*. Springer Verlag, 1994.

[11] C. Lewerenz and T. Lindner, editors. *Formal Development of Reactive Systems, LNCS 891*. Springer Verlag, 1994.

[12] P. Moeschler, H. Amann, and F. Pellandini. High-level modelling using extended timing diagrams. In *proceedings of EURO-DAC'93*, pages 494–499, Sept. 1993.

[13] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.

[14] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B (Formal Models and Semantics), chapter 4. Elsevier, 2. edition, 1992.