Journal of Visual Languages and Computing (2000) , 000–000

Visual Temporal Logic as a Rapid Prototyping Tool [†]

Martin Fränzle & Karsten Lüth

Carl von Ossietzky Universität Oldenburg
Department of Computer Science
P.O. Box 2503, D-26111 Oldenburg, Germany

Within this survey article, we explain real-time symbolic timing diagrams and the ICOS tool-box supporting timing-diagram-based requirements capture and rapid prototyping. Real-time symbolic timing diagrams are a full-fledged metric-time temporal logic, but with a graphical syntax reminiscent of the informal timing diagrams widely used in electrical engineering. ICOS integrates a variety of tools, ranging from graphical specification editors over tautology checking and counterexample generation to code generators emitting C or VHDL, thus bridging the gap from formal specification to rapid prototype generation.

1. Introduction

Due to rapidly dropping cost and the increasing power and flexibility of embedded digital hardware, digital control is becoming ubiquitous in technical systems encountered in everyday life. Often, such embedded systems can hardly be altered once they have been shipped out due to extremely large quantities being shipped, or they even have to be right first time due to their crucial impact on safety of human life. For example, modern means of transport rely on digital hardware even in vital sub-systems like anti-locking brakes, fly-by-wire systems, or signaling hardware, as does medical equipment even in such critical applications as life-support systems or radiation treatment. As, furthermore, many embedded systems are developed under tight time-to-market constraints, early availability of unambiguous specification of their intended behaviour has become an important factor for quality and timely delivery.

Such a specification is, however, always a compromise between various demands: ought it to be operational, in order to guide developers and programmers in the implementation phase, or ought it to be declarative in order to allow straightforward formulation of safety requirements, thus supporting safety analysis? Ought it to be formal, thus facilitating formal analysis and hence providing correctness guarantees that are not otherwise available, or may it be informal if this enhances comprehen-

[†] This article reflects work that has been partially funded by the German Research Council DFG under grants no. Da 205/5-1 and -2.

sibility, thus simplifying traditional approaches for ensuring quality of software, like testing and certification by code inspection?

Due to their prospects for reconciling some of these seemingly contradictory demands, graphical specifications have recently attracted much interest. Their prospects for gaining comprehensibility of even complex specification are deemed so high that it should be possible to simultaneously further the level of formality without sacrificing understandability. However, even with graphical specifications, the global interaction patterns of complex (for example, distributed) systems remain complex and, furthermore, it seems that graphical idioms also tend to hide some of the fine-grain semantics from the user, at least from the non-expert one. Thus, while being a big step ahead, graphical specification formalisms are not per se a means for ensuring that "what you specify is what you mean".

The problem of misconceptions in early development phases is, however, well-known in software engineering. A traditional remedy is rapid prototyping, where a partially developed product is brought into executable shape in order to assess its compliance with expectations. We suggest to take over the paradigm of "rapid prototyping" to the realm of formal, graphical, and declarative (i.e., intrinsicly non-operational) specifications such that these become executable, thereby facilitating early evaluation of specifications on an operational model. If such a prototyping process is based on an unambiguous semantics of specifications and applies rigorous rules for deriving executables from specifications then it can, furthermore, be made sure that the prototype obtained is in strict correspondence with the specification such that the evaluation is faithful.

In this article, we propose a method of that kind: it builds upon a fully formal semantics of specifications and applies automata-theoretic constructions for fully automatically deriving operational systems that represent the specification in a "what you specify is what you get" way. The method has been developed and implemented by the computer architecture group of Oldenburg University, which has dedicated part of its rapid prototyping project 'EVENTS' [16] towards automatic prototyping of embedded control hardware from fully formal specifications given as real-time symbolic timing diagrams. Real-time symbolic timing diagrams (RTSTDs, for short), as introduced in [7], are a graphical formalism for specifying behavioural requirements on hard real-time embedded systems. They are a full-fledged metric-time temporal logic, but with a graphical syntax reminiscent of the informal timing diagrams widely used in electrical engineering.

In Section 2 and 3, we introduce real-time symbolic timing diagrams and give an overview over the ICOS tool-box supporting requirements capture and rapid prototyping using RTSTDs. As the ICOS approach to rapid prototyping is based on synthesis of embedded control hardware — most frequently field-programmable gate arrays — satisfying the specification, we then turn to game-theoretic methods of controller synthesis. Section 4 introduces the controller synthesis problem, and Sect. 4.1 outlines a classical controller synthesis framework based on the effective correspondence of propositional temporal logic to finite automata on infinite words and on the theory of ω-regular games [21]. A compositional variant of this approach, which is more suitable for rapid prototyping purposes due to its reduced complexity, is shown in Sect. 4.2. This is our current synthesis method, which has been fully implemented in the ICOS tools [13, 15]. The results obtained using this method on e.g. the FZI production

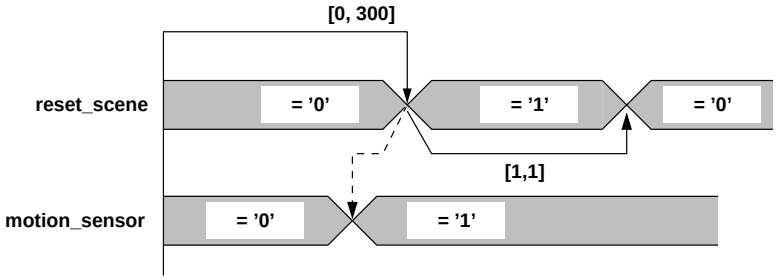


Figure 1. A requirement for a light controller, specified as an RTSTD. The solid arcs represent strong constraints, while weak constraints (i.e., assumptions on the environment) are printed as dashed arcs. The perpendicular line to the far left indicates the activation mode of the diagram, which here is the so-called invariant mode, meaning that the diagram has to apply whenever its activation condition fires. The activation condition is the state condition crossed by the perpendicular line, i.e. `reset_scene = '0'` $\wedge$ `motion_sensor = '0'`.

cell [14] indicate that the compositional extension yields a significant enhancement for reactive systems, yet a further compositional treatment of timing is necessary for real-time systems. Section 4.3 sketches the basic design decisions underlying such an extension which is currently being implemented for a new release of ICOS, while Sect. 5 compares this to the state of the art.

2. Real-time symbolic timing diagrams

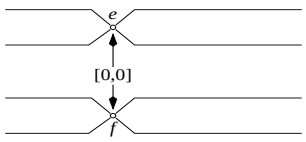
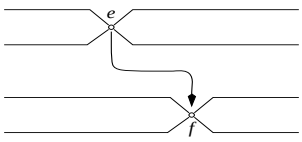
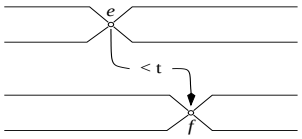
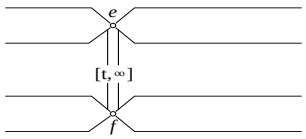
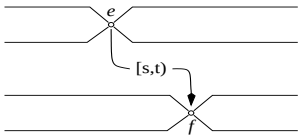
The RTSTD language [7] is a metric discrete-time temporal logic with an intuitive graphical syntax reminiscent of the informal timing diagrams widely used in electrical engineering, and with a declarative semantics which was originally formalized through a mapping to propositional temporal logic (PTL). In contrast to some other approaches using timing diagrams, e.g. those described in [3, 12], symbolic timing diagrams do not have any imperative control structure like iteration or sequencing. Instead, the recurrence structure of RTSTDs is expressed in terms of the modalities of linear-time temporal logic, thus providing a direct logical interpretation. In fact, RTSTDs provide a declarative rather than an operational specification style, even despite their intuitive syntax: an RTSTD is interpreted as a constraint on the admissible behaviours of a component, and a typical specification consists of several small RTSTDs, with the individual constraints being conjoined. The main consequence is that RTSTDs are well-suited for incremental development of requirements specifications. However, they pose harder problems than more operational timing diagrams when used as source language for code generation. Fig. 1 shows an example RTSTD, specifying a requirement for a light controller.

A basic [1] RTSTD consists of the following parts:

- An entity declaration defining the interface of the component (not shown in Fig. 1). It specifies the signals (e.g. `reset_scene, motion_sensor`), their

[1] Additionally, some syntactic sugar for making large specifications more concise is offered.

Table 1. The basic strong constraint types of RTSTDs and some compound constraint.

(a) Simultaneity constraint  If any of the events e or f occurs then the other one has to happen simultaneously.	(b) Precedence constraint  Event f does not occur before e. Once e has taken place, f may or may not happen.	(c) Leads-to$_t$ constraint  If event e happens then f occurred beforehand or it follows strictly less than t time-units after e, where $t \in \mathbb{N} \cup \{\infty\}$.
(d) Conflict$_t$ constraint  e and f do not occur less than t time-units apart, with $t \in \mathbb{N}_{>0}$.	The compound constraint type used in Fig. 1  A conjunction of precedence, conflict$_s$, and leads-to$_t$ constraints, enforcing that f follows e within time window $[s, t)$.	

data types (e.g. `Bit`) and access modes (`in` or `out`, i.e. being driven by the environment or the component, resp.).

- A set of waveforms (here `reset_scene, motion_sensor`). A waveform defines a sequence of Boolean expressions associated with a signal (e.g. `reset_scene ='0'`, then `reset_scene ='1'`, then `reset_scene ='0'` in the upper waveform of Fig. 1). The point of change from validity of one expression to another is called an event. A distinguished activation event, which specifies a precondition for applicability of the diagram, is located to the left over the waveforms.
- A set of constraints, denoted by constraint arcs, which define a partial order on events. We distinguish between strong constraints and weak constraints. Strong constraints are those which have to be satisfied by the system under development, while weak constraints denote assumptions on the behaviour of the environment. Violation of a weak constraint implies that the remainder of the timing diagram is preempted and consequently poses no further design obligations. Table 1 summarizes the different kinds of strong constraints. Each of these has a weak counterpart, denoted by a shaded instead of a solid constraint arc.

  The constraints, which are more closely examined in the next section, are akin to negation-free flat formulae of PTL [6], i.e. correspond to until- or unless-formulae with propositional left-hand side, yet add expressiveness beyond the flat fragment of PTL by subsuming the timed variants of until and unless also. [2]
- An activation mode. Initial diagrams describe requirements on the initial system behaviour whereas an invariant diagrams expresses requirements which must be satisfied at any time during system lifetime. Invariant mode corresponds to the 'always' modality of linear-time temporal logic, implying that — in contrast to timing diagram formalisms employing iteration — multiple incarnations of the

[2] Flat PTL, as defined in [6], cannot simulate the timed variants as it does not feature a next operator.

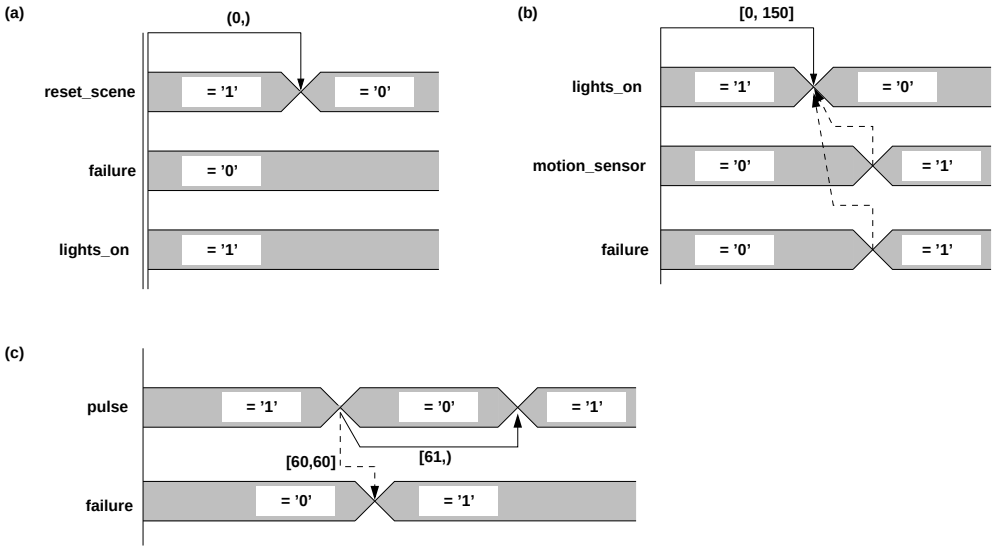


Figure 2. Light controller specification.

diagram body may be simultaneously active if the activation condition fires again before the end of the body has been reached.

2.1. Example

This section contains a simple specification to illustrate the features of the RTSTD language. The presented example was inspired by the Light Control System case study [2], and describes a light controller for a meeting room. The light controller is equipped with the following in- and outputs:

- an output `lights_on`, that switches the lights on and off.
- A person in the room can modify the light scene. An output `reset_scene` is used to reset the scene to the default settings.
- The controller is connected to a motion sensor. The input `motion_sensor` indicates, if a person was detected by the sensor.
- The input `pulse` is used to check if the motion sensor is working correctly. If must be set to a "high" (`= '1'`) value by the motion sensor every 60 seconds.
- The output `failure` indicates, that there is a problem with the motion sensor.

The initial diagram in figure 2(a) specifies the initial values of the outports. Initial diagrams are distinguished by a doubled perpendicular line at the far left side. All other diagrams are invariant diagrams, which will become "activated" whenever their activation condition holds. The diagrams in figures 2(b) and 1 describe the behavior of the controller if the motion sensor does not detect any person: in figure 1 the light scene settings will be reseted after 300s if no motion can be detected. A weak constraint (the dashed line) is used to express the assumption, that no motion sensor event occurs prior to the reset. In 2(b), the lights will be turned off after 150s, provided, there is no motion and the controller is not in the `failure` state. The

diagram in figure 2(c) shows a safety property: it signals a failure, if no pulse can be detected during a period of 60 seconds.

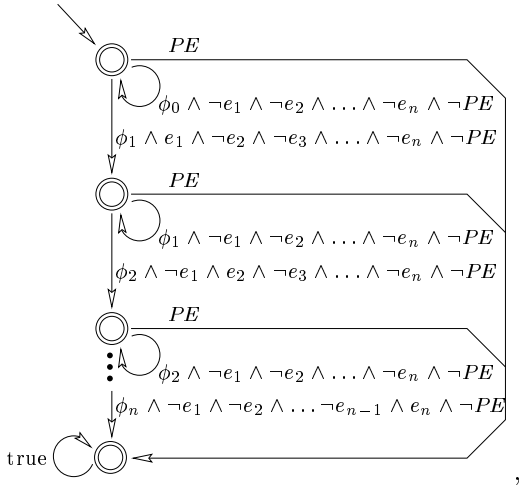
2.2. Detailed semantics

We will now provide a detailed semantics of RTSTDs by mapping them to timed automata [1], thereby employing the discrete-time interpretation of timed automata. While a dense-time interpretation would also make sense, we prefer the discrete-time one, as it renders timed automata an effective Boolean algebra, whereas timed automata are not even closed under complementation in the dense-time setting [1].

We start by giving semantics to waveforms. A waveform w, graphically depicted by

is a sequence of propositional predicates $\phi_0, \ldots, \phi_n$ over the values of a signal or port, interspersed by events $e_1, \ldots, e_n$. We assume the latter to be uniquely named throughout the whole RTSTD. [3] A waveform is called deterministic iff any two adjacent ϕ_i are mutually exclusive, i.e. iff $\phi_i \wedge \phi_{i+1}$ is unsatisfiable for each $i \in \{0, \ldots, n-1\}$.

We assign to w a timed Büchi automaton communicating port valuations as well as Boolean valuations for the event names, i.e. having alphabet $\alpha = \Sigma \times (\mathit{Event} \to \mathbb{B})$. Here, Σ is the set of possible port valuations, as stated in the entity declaration. *Event* is the set of all event names in the timing diagram, extended by a special preemption event *PE* invoked upon violation of some weak constraint, and for each weak constraint a further distinguished event $\mathit{violation}_i$ signaling violation of that weak constraint. Valuations of events will only be used internally for synchronizing constraints to waveforms and will be finally hidden from the semantics such that the overall semantics of timing diagrams is in terms of sequences of port valuations only. The Büchi automaton assigned to w is the automaton $A_w =$



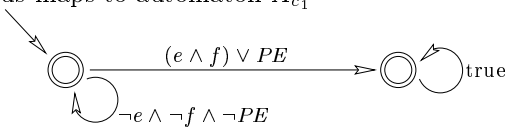
,

where doubly circled states denote accepting states (in this case, each state) and the predicates on the transition arcs mean that any alphabet entry satisfying the

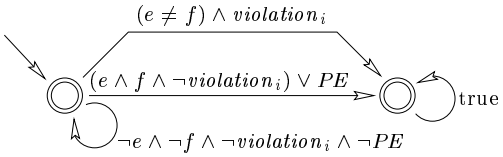
[3] Within our tool-box, unique naming is in fact ensured by the editor; the user need not even name events.

predicate may give rise to the corresponding transition. This automaton construction ensures that the port behaviour is compatible with the sequence $\phi_0, \phi_1, \ldots, \phi_n$ of predicates and that occurrences of the events $e_1, \ldots, e_n$ are confined to happen at the time instants corresponding to appropriate changes in the port valuation. All events not belonging to $e_1, \ldots, e_n$ may occur at any time, as they are not related to waveform w. Furthermore, A_w (and similarly any of the remaining automata) accepts the preemption event PE at any time, thereafter imposing no further constraints on port behaviour and event occurrences.

Constraint arcs impose extra constraints on the occurrence of events. The strong simultaneity constraint c_1 from Table 1(a) confines the two events e and f to happen simultaneously and thus maps to automaton $A_{c_1} =$

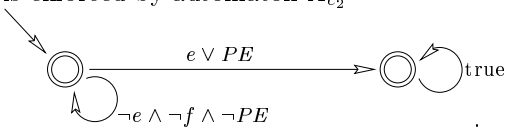


In contrast, the weak counterpart of above simultaneity constraint maps to automaton $A_{c_{1w}} =$



which does not impose any constraints, but signals constraint violation through the distinguished event $violation_i$. The $violation_i$ events from all automata implementing weak constraints, where the index i distinguishes the events used in the different automata, are later collected and or-ed together to form the preemption event PE. This causes the timing diagram to be preempted and thus to impose no further constraints on the temporal behaviour as soon as any weak constraint arc becomes violated. Weak variants of the other constraint types are derived analogously; we will therefore not elaborate on them in the remainder.

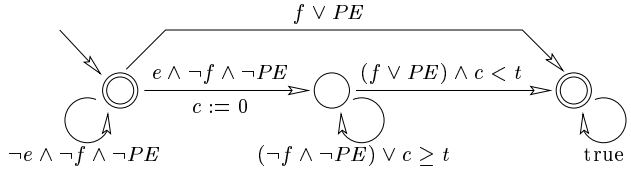
The strong precedence constraint from Table 1(b) requires that event f may not occur before e, which is enforced by automaton $A_{c_2} =$



.

Note that f need not happen, even if e has already been observed.

A strong timed leads-to constraint, as in Table 1(c), is implemented by automaton $A_{c_3} =$

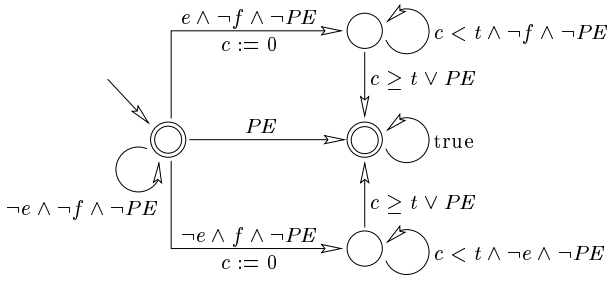


,

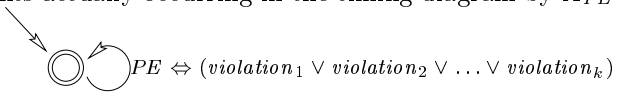
where c is a clock that is local to this automaton. Being motivated by the needs of synchronous system design, we slightly deviate from the usual interpretation of clocks in timed automata [1] and assume that clocks count transitions rather than

an independent system time.[4] Through use of the clock, above automaton forces f to take place less than t time units after e has occurred. Note that in case $t = \infty$, which denotes an unbounded liveness constraint, proper behaviour is enforced by Büchi acceptance and the middle state being non-accepting, causing f to eventually happen.

Finally, a strong conflict-t constraint, as in Table 1(d), asks for e and f to keep a temporal distance of at least t time units, which is implemented by $A_{c_4} =$



Now it remains to issue a preemption event PE whenever some weak constraint has been violated. Therefore, we "or together" the violation signals $violation_1, violation_2, \ldots$ of all weak constraints actually occurring in the timing diagram by $A_{PE} =$



The semantics of a complete timing diagram body is the conjunction of the semantics of its parts, yet projected to the behaviour on the ports only, thus eliminating all event names. This can be achieved by first taking the automaton product of the automata encoding the waveforms and constraints of the timing diagram, then applying morphism $\phi : \alpha \to \Sigma$ defined by $\phi((\sigma, e)) = \sigma$:

$$A_{TD} = \phi \left(\prod_{w \in w(TD)} A_w \times \prod_{c \in c(TD)} A_c \times A_{PE} \right) ,$$

where $w(TD)$ is the set of waveforms in TD and $c(TD)$ is the set of constraints in TD. It is easy to see that the resulting Büchi automaton is deterministic if all waveforms in TD are deterministic.

2.2.1. Initial and invariant diagrams.

The semantics of an initial diagram is simply that of its body: for any run $r \in \Sigma^\omega$, i.e. for any infinite sequence r of port valuations, we define satisfaction of a timing diagram TD by r, denoted $r \models TD$, by

$$r \models TD \quad \text{iff} \quad r \in \mathcal{L}_{A_{TD}}, \text{ provided } TD \text{ is initial.}$$

Here, $\mathcal{L}_A$ denotes the ω-regular language recognized by A under the convention that one transition is taken every time instant. Consequently, the set $\mathcal{M}[\![TD]\!]$ of models of

[4] Note that this is in fact equivalent to assuming that one transition takes place every time instant, which can be enforced by using an additional clock in the standard model of timed automata. Our convention merely saves this clock.

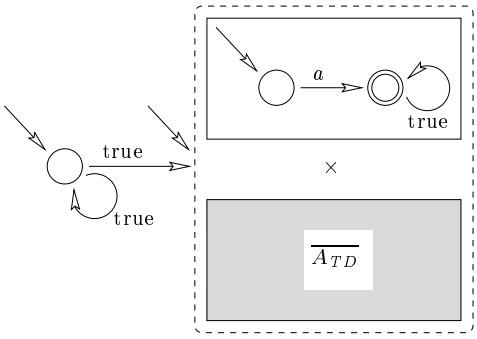
an initial diagram TD is an ω-regular language, and the construction of A_{TD} effectively yields an automaton recognizing $\mathcal{M}[\![TD]\!]$. The expressiveness of initial diagrams is that of negation-free flat PTL formulae [6], yet extended by a next operator; however, much conciseness is gained by being able to directly specify metric time bounds instead of having to expand them into an appropriate cascade of next operators.

Invariant diagrams add further expressiveness by embedding the semantics of the diagram body into a single, outermost always operator: for any run $r \in \Sigma^\omega$,

$$r \models TD \quad \text{iff} \quad \forall i \in \mathbb{N} : \left(\begin{array}{c} r_i \text{ satisfies } a \\ \Rightarrow\ r[i,\ldots] \in \mathcal{L}_{A_{TD}} \end{array} \right), \text{ provided } TD \text{ is invariant.}$$

Here, r_i denotes the i-th position in r, a is the activation condition of TD, which is the conjunction of the leftmost state predicates of all waveforms in TD, and $r[i,\ldots]$ denotes r from position i onwards, i.e. is the infinite sequence $\langle r_i, r_{i+1}, r_{i+2}, \ldots \rangle$ of port valuations. Thus, $r \models TD$ iff starting in any time instant that satisfies the activation condition a, the timing diagram body is satisfied by the remaining behaviour. Intuitively, one may think of a new incarnation of the timing diagram body being created whenever the activation condition fires.

Please note that — as discrete-time timed automata are only more concise, yet not more expressive than untimed Büchi automata — an effective automata-theoretic construction for (untimed) Büchi automata recognizing the set $\mathcal{M}[\![TD]\!]$ of models of an invariant diagram TD is in principle available: it consists of first unraveling the timed automaton A_{TD} into an untimed Büchi automaton, then complementing it into an automaton $\overline{A_{TD}}$, thereafter adding an initial loop of shape



which non-deterministically waits until starting $\overline{A_{TD}}$ in a situation satisfying activation condition a, and finally complementing the resulting device again. However, this construction is extremely expensive due to the double complementation of Büchi automata involved. This is the reason for in practice confining ourselves to deterministic timing diagrams. These yield timed Büchi automata A_{TD} which are extremely simple and cheap to complement, as they are deterministic and the only kind of loops occurring in above construction is self-loops, where a state directly loops back to itself. Under these circumstances, complementation of A_{TD} can be achieved by simply complementing the set of accepting states,[5] even although we are dealing with Büchi acceptance.

[5] Clearly, one has to make the partial transition relation depicted in above automata diagrams complete by introducing non-accepting sink states.

As a last step in giving semantics to timing diagrams, the semantics of a set of timing diagrams is defined to simply be the conjunction (i.e. language intersection, corresponding to automaton product) of the semantics of the individual timing diagrams.

3. The ICOS tool-box

Despite the appealing graphical syntax of RTSTDs, specification of reactive systems using RTSTDs remains a challenging task. The ICOS tool-box, as shown in Fig. 3, is a comprehensive set of tools that supports this process [8, 13, 15]. ICOS is built

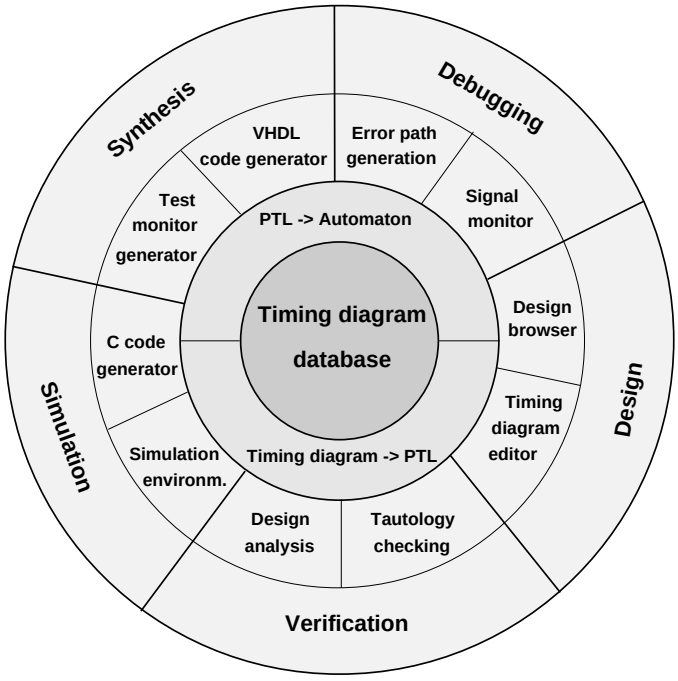


Figure 3. The ICOS system

around a timing diagram database that holds all specification clauses belonging to the current project. The database is augmented by a layer of procedures for compiling RTSTDs to other effective representations of reactive behaviour, like propositional temporal logic (PTL) [17] or Büchi automata [21]. These procedures, as well as the corresponding derived representations of reactive behaviour, are in general hidden from the user. Instead, they are invoked (resp., constructed) on the fly whenever a particular representation is needed for some user-selected task (e.g., for code generation).

User-selectable activities are concerned with design, verification, simulation, synthesis, and debugging. Design-related activities involve creating and editing RTSTDs with a graphical editor, as well as browsing the specification database. Verification entails static analysis, for checking whether interfaces are consistently used, and automatic tautology checking, for formally verifying that some refined specification satisfies

the original commitments of the design task. Simulation is performed upon a keystroke by first generating a finite automaton faithfully reflecting the semantics of the specification, then encoding it in the C programming language, and finally linking it to a simulation environment, thus facilitating interactive simulation of specifications. The intermediate steps involved do, however, not require any user interaction.

Simulation shares most of the basic technology with synthesis. Synthesis goes however a bit further in that it actually delivers code for programming a field-programmable gate array (FPGA), thus yielding an embedded control device that can be plugged into the application context rather than just an interactive simulation. The steps involved are, first, the generation of a set of interacting Moore automata satisfying the specification, second, their translation to synthesizable VHDL code, and third, synthesis of a corresponding FPGA. We will expand on the techniques underlying synthesis in the next section, as embedded controller synthesis forms the backbone of our rapid prototyping method.

Test monitor generation is a minor variant of this synthesis process which yields an FPGA that does not control its application context, but instead may be plugged into the application context as an online monitor, then monitoring the running system for violations of the specification.

Debugging, finally, yields error paths pinpointing the problem whenever synthesis fails due to a contradictory (and thus unimplementable) specification, or if tautology checking finds a non-tautology.

4. Synthesis

Automatic synthesis of FPGA-based controllers is the core of our rapid prototyping method for RTSTD specifications. The application scenario is that the user invokes an fully automatic synthesis procedure once the requirement set, being formalised through RTSTDs, is deemed sufficiently complete. The synthesis algorithm then tries to construct an FPGA-based embedded controller satisfying the specification — i.e., an operational prototype — or, if it detects non-implementability of the specification — delivers diagnostic information.

In fact, the synthesis method is a three-step procedure: first, the synthesis algorithm tries to construct a Moore automaton satisfying all stated requirements. However, no appropriate Moore automaton need exist due to a contradictory specification in which case the algorithm creates error paths to help the programmer refine the specification. Otherwise, the second step can commence, where the generated Moore automaton is automatically translated to VHDL code. The subset of VHDL used as target language is such that the final step of actually implementing the automaton by a given target technology (e.g. FPGAs) can be done through compilation (sometimes called "high-level synthesis" by the tool vendors) of the VHDL description, and we have indeed integrated our tools with the Synopsys [10] and some Xilinx tools to achieve this.

As high-level synthesis is by now an industrially available technology, we will in the remainder of this article concentrate on the first step and will sketch different specification-level synthesis procedures yielding sets of interacting Moore machines from RTSTD specifications. The procedures differ in the methods used for dealing with timing constraints and in the number and shapes of the interacting Moore machines

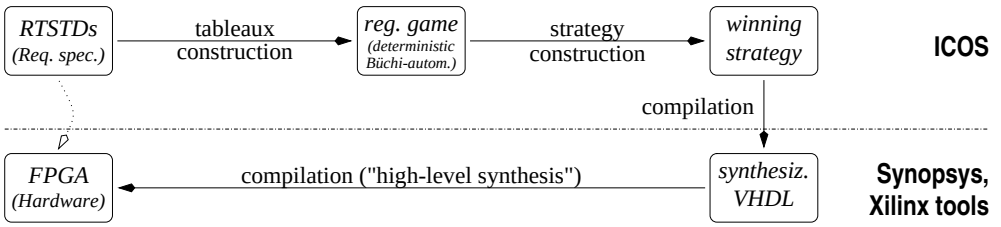


Figure 4. The basic synthesis chain.

generated, which affects the average-case complexity of the synthesis procedure and the size of the hardware devices delivered.

Given a specification ϕ, the problem of constructing a Moore automaton satisfying the specification, called the synthesis problem, is to find a Moore automaton A_ϕ which is

(1) well-typed wrt. ϕ, i.e. has the inputs and outputs that the entity declaration of ϕ requires, and accepts any input at any time (i.e., only the outputs are constrained),

(2) correct wrt. ϕ, i.e. the possible behaviours of A_ϕ are allowed by the specification ϕ, formally $\mathcal{L}_{A_\phi} \subseteq \mathcal{M}[\![\phi]\!]$, where $\mathcal{M}[\![\phi]\!]$ is the set of behaviours satisfying ϕ and $\mathcal{L}_{A_\phi}$ is the set of possible behaviours of A_ϕ.

By adoption of logical standard terminology, an algorithm solving the synthesis problem is called sound iff, given any specification ϕ, it either delivers an automaton that is well-typed and correct wrt. the specification ϕ, or no automaton at all. It is called complete iff it delivers an automaton whenever a well-typed and correct automaton wrt. the specification exists.

4.1. Classical controller synthesis and RTSTDs

For control problems specified by ω-regular games, e.g. finite game graphs augmented with the Büchi acceptance condition, there is a firmly developed theory of winning strategy construction [21]. This theory provides fully automatic procedures for generating finite-state winning strategies, i.e. finite-state controllers, from infinite regular games specifying the allowable behaviour. As the semantics exposed in Sect. 2.2 provides an effective mapping of deterministic RTSTDs to deterministic Büchi automata, this approach can be extended to deal with deterministic RTSTDs also. Soundness and completeness of the synthesis method then is directly inherited from the corresponding properties of winning strategy construction in regular games. In fact, this chain of algorithms, depicted in Fig. 4, forms the backbone of the ICOS tool set [8, 15].

However, this basic synthesis chain suffers from complexity problems if the specification is large, i.e. is a conjunction of multiple timing diagrams, as the regular games constructed then tend to suffer from state explosion. With the basic method, game graphs grow exponentially in the number of timing diagrams due to the automaton product involved in dealing with conjunction. As this would render application for rapid prototyping impractical, the ICOS tools offer modified procedures which reduce the complexity of dealing with large specifications. Obviously, such extensions cannot

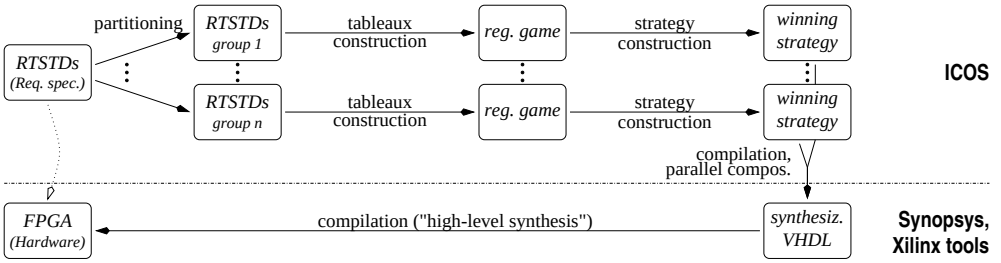


Figure 5. Compositional synthesis.

deal efficiently with arbitrary RTSTD specifications, but they are, however, carefully designed to cover the typical specification patterns.

4.2. Compositional synthesis

The first such variant is a compositional extension of the basic algorithm. Within this approach, which is sketched in Fig. 5, the specification is first partitioned into a maximal set of groups of formulae $\mathcal{G}_1, ..., \mathcal{G}_n$ such that each output is constrained by the formulae of at most one group. Then synthesis down to a winning strategy is performed for each group individually, yielding for each group $\mathcal{G}_i$ a Moore automaton A_i that has just the outputs constrained by $\mathcal{G}_i$ as outputs, and all other ports as inputs. The individual Moore automata are then compiled to synthesizable VHDL and composed by parallel composition.

With this compositional method, growth of the game graph is linear in the number of groups, and exponential growth is only encountered in the maximal size of the individual groups. Table 2 provides empiric results from [8], obtained by Feyerabend and Schlör when using both the non-compositional and the compositional synthesis procedure of ICOS for synthesizing a controller for the FZI production cell [14]. Overall, Feyerabend and Schlör have found the compositional approach to save over 99% of the transitions in the game graphs of the major components.

Soundness of the compositional synthesis technique is easily established:

Theorem 1. Let $\mathcal{G}_1, ..., \mathcal{G}_n$ be groups of RTSTDs with O_i being the outputs constrained by $\mathcal{G}_i$, and with $O_i \cap O_j = \emptyset$ for $i \neq j$. Let A_i be the Moore automaton synthesized for $\mathcal{G}_i$. Then $A_1 \parallel \ldots \parallel A_n$ is well-typed and correct wrt. $\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_n$, where $\parallel$ denotes parallel composition of Moore automata.

Proof. Correctness of $A_1 \parallel \ldots \parallel A_n$ wrt. $\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_n$ is straightforward, as parallel composition of Moore automata with disjoint output alphabets semantically yields language intersection, as does conjunction of RTSTDs. Thus, soundness of the basic synthesis algorithm, which yields $\mathcal{L}_{A_i} \subseteq \mathcal{M}[\![\mathcal{G}_i]\!]$ for the individual groups, implies $\mathcal{L}_{A_1 \parallel \ldots \parallel A_n} = \mathcal{L}_{A_1} \cap \ldots \cap \mathcal{L}_{A_n} \subseteq \mathcal{M}[\![\mathcal{G}_1]\!] \cap \ldots \cap \mathcal{M}[\![\mathcal{G}_n]\!] = \mathcal{M}[\![\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_n]\!]$.
Similarly, well-typedness of $A_1 \parallel \ldots \parallel A_n$ wrt. $\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_n$ follows from soundness of the basic synthesis procedure since the composition rules for interfaces agree for Moore automata and RTSTDs if outputs occur at most once in a parallel composition. □

Table 2.Performance of compositional synthesis (after [8]).

Non-compositional synthesis of a controller for the FZI production cell [14]: System has 8 components. One controller per component is synthesized, the components posing the most complex synthesis problems being:

Component	No. of STDs	Game graph states	Game graph transitions	Controller states	Controller transitions	Time (s)
Press	14	1703	115897	82	435	4483
Crane	16	1574	76041	190	1303	1035

Compositional synthesis of a controller for the FZI production cell: Specification is automatically partitioned into 26 groups. One controller per group is synthesized. Most complex group deals with controlling vertical movement of the press:

	No. of STDs	Game graph states	Game graph transitions	Controller states	Controller transitions	Time (s)
Press, vertical movement	5	27	193	6	26	8

Completeness is, however, lost using compositional synthesis. The problem is that a certain group may not be synthesizable without knowledge about the behaviour of another group. Such problems are regularly encountered within compositional methods, and we propose to solve them by just the same techniques that proved to be helpful in compositional verification: the necessary information on the other components can be formalized via assumptions (for example, through weak constraint arcs). It should be noted that ICOS helps in finding adequate assumptions, as an error path is supplied whenever synthesis of a component fails.

4.3. Synthesizing hardware clocks

However, there still is some reason for dissatisfaction: as timing annotations have to be unwound to an according number of transitions in the game graph by the translation of RTSTDs to ω-regular games, the modular synthesis method remains exponential in the number of potentially overlapping timing constraints. This makes dealing with timing constraints of more than a handful time units hardly affordable — realistic real-time system programming cannot be done with such code generation methods. Therefore, we are heading for an algorithm that is of linear complexity in the number of timing constraints, even though completeness is thereby necessarily sacrificed. What is thus needed is a synthesis method that separates generation of timers controlling the allowable passage of time from synthesis of an untimed control skeleton. In the remainder, we sketch a synthesis technique currently being integrated into ICOS, which offers the desired separation. In this hybrid approach, small — and thus harmless wrt. the state explosion problem — time constants are treated by purely ω-automata-theoretic means, whereas large-scale timing constraints, if found to be sufficiently independent, are directly mapped to hardware timers.

The new approach starts by using timed automata for representing the semantics of real-time symbolic timing diagrams: every RTSTD ϕ is assigned a timed automaton

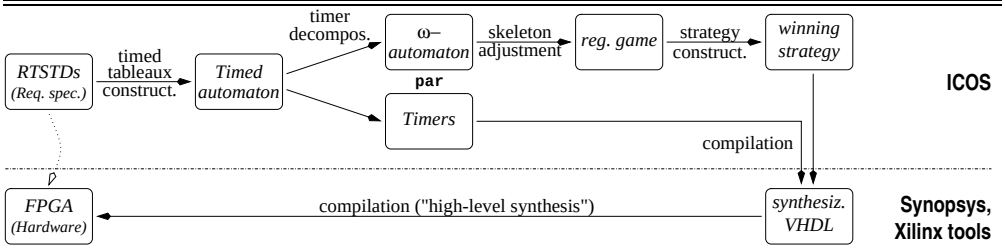


Figure 6. Synthesis chain employing timer decomposition.

A_ϕ that accepts exactly the counterexamples of ϕ, i.e. all those traces that are not models of ϕ. An example of a timed automaton recognizing the counterexamples of an RTSTD can be seen on the left hand side of Fig. 7, where the conjunction of a precedence and a leads-to$_t$ constraint between events e and f is dealt with.

Once a timed automaton accepting counterexamples to the specification is constructed, synthesis of a controller satisfying the specification can commence: in a first step, all clocks implementing delays of more than a handful time units are removed from the timed automaton and replaced by external timer components acting in parallel, as shown in Fig. 7. Thereafter, the remaining clocks are removed by expanding their effect to an appropriate number of unit-delay transitions. This results in an untimed automaton, called sequential skeleton in the remainder, which communicates with the environment and with the timer components. The novelty of our approach is that from then on, synthesis will treat the timers similar to environment components, which means that the behaviour of these components is left untouched during synthesis. The advantages are twofold: first of all, the fixed behavioural description of timers allows for the use of pre-fabricated, optimized hardware components, and second, controller synthesis can concentrate on solving the control problem described by the small, untimed automaton that remains.

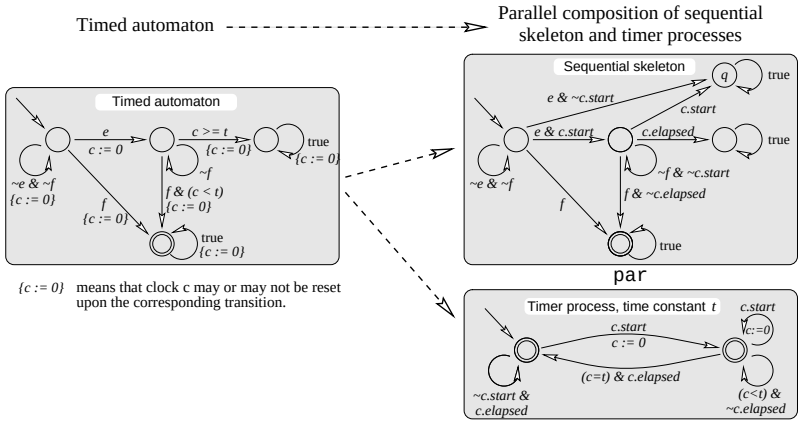


Figure 7. Converting clocks to timer components.

Note that the parallel composition (alas automaton product, thus yielding language intersection) of the sequential skeleton and the timer processes derived thus far recognizes the counterexamples to the specification. Now, we would like to implement

the timers in hardware and to remove them from the synthesis problem, i.e. we want to synthesize wrt. the skeleton only without, however, risking erroneous behaviour of the synthesized controller. As the timer communications are only internal to the controller, the correctness criterion involved is

$$(\mathcal{L}_C \cap \mathcal{L}_{timer}) \setminus [t.start, t.elapsed] \subseteq \overline{(\mathcal{L} \cap \mathcal{L}_t) \setminus [t.start, t.elapsed]} \ , \qquad (1)$$

where $\mathcal{L}_C$ and $\mathcal{L}_{timer}$ are the trace sets of the controller and the timers, resp., $\mathcal{L}$ is the language accepted by the skeleton automaton, '$\setminus[t.start, t.elapsed]$' denotes hiding of the timer communications, and the overbar denotes language complement.

It might seem that straightforward synthesis wrt. the complement of the skeleton, which yields a controller C satisfying the language inclusion property $\mathcal{L}_C \subseteq \overline{\mathcal{L}}$, suffices. Unfortunately, $\mathcal{L}_C \subseteq \overline{\mathcal{L}}$ is in general not a sufficient condition for (1) due to the existential quantification involved in hiding, [6] which changes to universal quantification under the complementation involved in (1).

However, this can be repaired by synthesizing wrt. an appropriately adjusted variant *skel′* of the skeleton automaton that enforces a certain usage of timers. The key issue is that the synthesized controller is forced to start a timer and not interfere its run whenever a violation of the corresponding timing constraint could possibly occur. The new skeleton *skel′* is generated by taking the same state set, same initial states, and same transition relation as in the original skeleton, yet expanding the set of accepting states by states like state q in Fig. 7, which is entered if the timer signaling the possible violation of the leads-to$_t$ constraint is not properly activated. The detailed construction, which we cannot provide due to lack of space, is s.t. if a timer action sequence ts exists with $w \oplus ts \in \mathcal{L} \cap \mathcal{L}_{timer}$ then $w \oplus ts' \in \mathcal{L}_{skel'}$ for each $ts' \in \mathcal{L}_{timer}$. Note that furthermore $\mathcal{L}_{skel'} \supseteq \mathcal{L}$ by construction.

If we now synthesize a controller with inputs $I \cup \{t.elapsed\}$, outputs $O \cup \{t.start\}$ (where I and O are the original in- and outputs) that is correct wrt. the adjusted control problem, i.e. satisfies $\mathcal{L}_C \subseteq \overline{\mathcal{L}_{skel'}}$, then we have obtained a correct solution:

Lemma 2. Correctness of C wrt. $\overline{\mathcal{L}_{skel'}}$, i.e. $\mathcal{L}_C \subseteq \overline{\mathcal{L}_{skel'}}$, implies (1).

Proof. Assume that $\mathcal{L}_C \subseteq \overline{\mathcal{L}_{skel'}}$ holds and (1) is false, i.e. there exists some $w \in (\mathcal{L}_C \cap \mathcal{L}_{timer}) \setminus [t.start, t.elapsed] \cap (\mathcal{L} \cap \mathcal{L}_{timer}) \setminus [t.start, t.elapsed]$. By definition of hiding this implies that there are two timer action sequences ts_1, ts_2 with

$$w \oplus ts_1 \in \mathcal{L}_C \cap \mathcal{L}_{timer} \quad \wedge \quad w \oplus ts_2 \in \mathcal{L} \cap \mathcal{L}_{timer} \ .$$

Then, by construction of *skel′*, $w \oplus ts_1 \in \mathcal{L}_{skel'}$. But on the other hand $w \oplus ts_1 \in \mathcal{L}_C$ and $\mathcal{L}_C \subseteq \overline{\mathcal{L}_{skel'}}$, which yields a contradiction. □

Consequently, synthesis wrt. the adjusted control problem is sound. Furthermore, by directly mapping all timing constraints of significant size to hardware timers, this method is linear in the number of timing constraints. If it is furthermore combined with modular synthesis then super-linear blow-up can only occur through individual

[6] For a language $\mathcal{L} \subseteq (\alpha \times (\{t.start, t.elapsed\} \to \mathbb{B}))^\omega$ and some string $w \in \alpha^\omega$, $w \in \mathcal{L} \setminus [t.start, t.elapsed]$ holds if some sequence $ts \in (\{t.start, t.elapsed\} \to \mathbb{B})^\omega$ of timer actions exists with $w \oplus ts \in \mathcal{L}$.

timing diagrams containing unusually large numbers of events or through large groups of formulae controlling the same outputs. These situations are, however, atypical, rendering the new method a practical rapid-prototyping tool for real-time embedded controllers.

Similar methods are, btw., currently explored by the second author for dealing with data in an essentially uninterpreted way. The idea is that thus the traditional separation between control and data can be incorporated into game-theoretic synthesis methods. The net effect is that a large data path no longer yields a blow-up of the game-graph. The complexity of automatic synthesis then predominantly depends on the control part, making automatic synthesis applicable to much larger, control-dominated systems.

5. Discussion

We have presented a rapid-prototyping framework for requirements specifications that are formalised through real-time symbolic timing diagrams, a metric-time temporal logic with a graphical syntax akin to the informal timing diagrams used in electrical engineering [7]. The underlying core technology is fully automatic synthesis of embedded control hardware from requirements specifications. This involves two major steps: first, the generation of Moore automata satisfying the specification, and second, the actual implementation of these automata by embedded control hardware. For the latter step, we rely on industrially available tools from Synopsys and Xilinx that perform FPGA-based implementation of VHDL-coded Moore-automata [10]. In contrast, the former step is based on procedures that have been specifically implemented for the ICOS tool-box [8, 13, 15]. While the underlying algorithms have been derived from the well-established theory of winning-strategy construction in ω-regular games, the overall approach is — being targeted towards rapid prototyping — mostly pragmatic, weighing efficiency higher than completeness. Two key issues have been identified in this respect: first, the necessity of compositional synthesis of parallel components and second, early decomposition of timing issues from the synthesis problem. The result is a synthesis method that is essentially linear in the size of the specification and thus suitable as a development tool in rapid prototyping contexts.

Within the ICOS tool-box, these algorithms are closely integrated with a graphical specification editor supporting the specification phase and all compilation and mapping algorithms necessary for mechanically translating down to automatically synthesizable VHDL code such that implementation is truly automatic from the specification level all the way down to actual hardware. ICOS furthermore comprises — under a common user interface — tools for browsing and manipulating a specification database, as well as for interactively simulating and for verifying specifications. It is thus a comprehensive environment for the incremental development of RTSTD-based requirements specifications. Furthermore, within the rapid-prototyping project "EVENTS" [16], ICOS is conjoined with Statemate-based code generation techniques such that rapid prototyping of hybridly specified systems, where some components have a declarative formulation through RTSTDs while others are operationally described by Statecharts [11], becomes feasible. While this involves integration of differ-

ent rapid-prototyping tools through automatic interface generation for the generated components, other extensions to the source language accepted can be accomplished within just the ICOS tool: ICOS is modularly built such that while the current version is dedicated towards RTSTD-based specification, an adaptation to other declarative specification formalisms is possible. A possible candidate formalism, for which front-end tools like graphical editors are already under development at our department, is that of Live Sequence Charts (LSCs), an extension of Message Sequence Charts (MSCs) recently proposed by Damm and Harel [5].

On the algorithmic side, we think that the main contribution of the ICOS tools to game-theoretic synthesis are methods for dealing with timing or with the data path in an uninterpreted way. Due to the obvious necessity of treating timing mostly independent from algorithmic aspects within any reasonably efficient synthesis method for hard real-time controllers, quite a few other research groups work on this theme. However, most approaches are based on primarily operational rather than declarative specification styles (e.g. [22]). Closest to our approach is [18], where Dierks and Olderog detail a direct mechanism for deriving timer actions from specifications formalised through a very restrictive subset of Duration Calculus [23], the so-called DC-implementables [19]. Dierks' algorithm is extremely efficient, but at the price of a very restrictive specification format: the processable specifications are confined to be 'phased designs' in the sense of the ProCoS projects [4], which are akin to RTSTDs featuring just three events. While our formalism is more expressive in this respect, Dierks and Olderog do, on the other hand, go ahead by dealing with a dense-time logic and analyzing certain kinds of switching latency.

References

1. R. Alur & D. L. Dill (1994) A theory of timed automata. Theoretical Computer Science 126(2): 183–235.
2. Egon Börger and Reinhard Gotzhein. Requirements Engineering - The Light Control Case Study. Journal of Universal Computer Science, 6(7), 2000.
3. G. Borriello (1992) Formalized timing diagrams. In: The European Conference on Design Automation 1992, IEEE Computer Society Press, pp. 372–377.
4. J. P. Bowen, M. Fränzle, E.-R. Olderog & A. P. Ravn (1993) Developing correct systems. In: Proc. 5th Euromicro Workshop on Real-Time Systems, IEEE Computer Society Press, pp. 176–189.
5. W. Damm & D. Harel (1999) LSCs: Breathing Life into Message Sequence Charts. In: FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, ????? Ergänzen!!
6. D. R. Dams (1999) Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. Logic Journal of the IGPL 7(1): 55–78.
7. K. Feyerabend & B. Josko (1997) A visual formalism for real time requirement specification. In: Transformation-Based Reactive System Development, LNCS 1231, Springer Verlag, pp. 156–168.
8. K. Feyerabend & R. Schlör (1996) Hardware synthesis from requirement specifications. In: Proceedings of EURO-DAC'96 with EURO-VHDL'96, IEEE Computer Society Press.
9. M. Fränzle & K. Lüth (1998) Compiling graphical real-time specifications into silicon. In [20], pp. 272–281.
10. S. Golsen (1994) State Machine Design Techniques for Verilog and VHDL. Synopsys Journal of High Level Design, Sept. 1994.
11. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman & A. Shtul-Trauring (1988) STATEMATE; a working environment for the development of complex reactive

systems. In: Proceedings of the 10th International Conference on Software Engineering, IEEE Computer Society Press, pp. 396–406.
12. P. Khordoc, M. Dufresne & E. Czerny (1991) A Stimulus/Response System based on Hierarchical Timing Diagrams. Technical report, publication no. 770, Universite de Montreal.
13. F. Korf (1997) System-Level Synthesewerkzeuge: Von der Theorie zur Anwendung. Dissertation, Fachbereich Informatik, Carl von Ossietzky Universität Oldenburg, Germany.
14. C. Lewerentz & T. Lindner, eds. (1995) Formal Development of Reactive Systems: Case Study Production Cell. LNCS 891, Springer-Verlag.
15. K. Lüth (1998) The ICOS synthesis environment. In [20], pp. 294–297.
16. K. Lüth, A. Metzner, T. Peikenkamp & J. Risau (1997) The EVENTS Approach to Rapid Prototyping for Embedded Control Systems. In: Zielarchitekturen eingebetteter Systeme (ZES '97), 14. ITG/GI Fachtagung Architektur von Rechnersystemen (ARCS'97), Rostock, Germany, September 1997.
17. Z. Manna & A. Pnueli (1992) The Temporal Logic of Reactive and Concurrent Systems, volume 1. Springer-Verlag.
18. E.-R. Olderog & H. Dierks (1998) Decomposing real-time specifications. In: Compositionality: The Significant Difference (H. Langmaack, W. P. de Roever & A. Pnueli, eds.), LNCS 1536, Springer-Verlag.
19. A. P. Ravn (1995) Design of Embedded Real-Time Computing Systems. Doctoral dissertation, Department of Computer Science, Danish Technical University, Lyngby, DK.
20. A. P. Ravn and H. Rischel, eds. (1998) Formal Techniques in Real-Time and Fault-Tolerant Systems. LNCS 1486, Springer-Verlag.
21. W. Thomas (1990) Automata on infinite objects. In: Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics (J. v. Leeuwen, ed.). North-Holland, pp. 133–191.
22. P. Vanbekbergen, G. Gossens & B. Lin (1994) Modeling and synthesis of timed asynchronous circuits. In: Proceedings EURO-DAC with EURO-VHDL 94, IEEE Comp. Soc. Press.
23. Zhou Chaochen, C. A. R. Hoare & A. P. Ravn (1991) A calculus of durations. Information Processing Letters 40(5):269–276.