New Results on Timed Specifications*

Timothy Bourke[2], Alexandre David[1], Kim. G. Larsen[1], Axel Legay[2],
Didier Lime[3], Ulrik Nyman[1], Andrzej Wąsowski[4]

[1] Computer Science, Aalborg University, Denmark
[2] INRIA/IRISA, Rennes Cedex, France
[3] IRCCyN/Ecole Centrale de Nantes, France
[4] IT University of Copenhagen, Denmark

Abstract. Recently, we have proposed a new design theory for timed systems. This theory, building on Timed I/O Automata with game semantics, includes classical operators like satisfaction, consistency, logical composition and structural composition. This paper presents a new efficient algorithm for checking Büchi objectives of timed games. This new algorithm can be used to enforce liveness in an interface, or to guarantee that the interface can indeed be implemented. We illustrate the framework with an infrared sensor case study.

1 Introduction and State of The Art

Several authors have proposed frameworks for reasoning about interfaces of independently developed components (e.g. [20, 13, 9, 12]). Most of these works have, however, devoted little attention to real-time aspects. Recently, we proposed a new specification theory for Timed Systems (TS) [11]. Syntactically, our specifications are represented as Timed I/O Automata (TIOAs) [19], i.e., timed automata whose discrete transitions are labeled by *Input* and *Output* modalities. In contrast to most existing frameworks based on this model, we view TIOAs as games between two players: Input and Output, which allows for an optimistic treatment of operations on specifications [13].

Our theory is equipped with features typical of a compositional design framework: a *satisfaction relation* (to decide whether a TS is an implementation of a specification), a *consistency check* (whether the specification admits an implementation), and a refinement (to compare specifications in terms of inclusion of sets of implementations). Moreover, the model is also equipped with *logical composition* (to compute the intersection of sets of implementations), *structural composition* (to combine specifications) and its dual operator *quotient*. Our framework also supports incremental design [14].

Refinement, Satisfaction, and Consistency problems can be reduced to solving timed-games. For example, if inconsistent states are states that cannot be implemented, since they violate assumptions of the abstraction, then deciding whether an interface is consistent is equivalent to checking if a strategy that avoids inconsistent states exists.

Our theory is implemented in ECDAR [17], a tool that leverages the game engine UPPAAL-TIGA [4], as well as the model editor and the simulator of the UPPAAL model

* Work partially supported by VKR Centre of Excellence – MT-LAB and by an "Action de Recherche Collaborative" ARC (TP)I

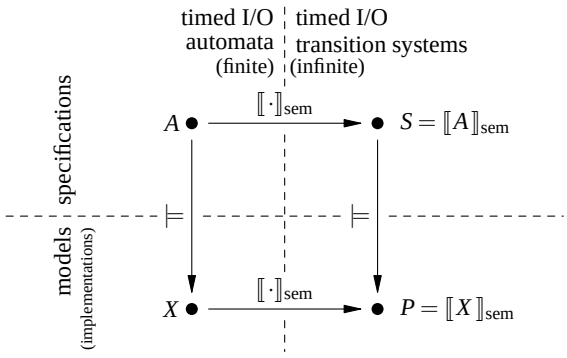


Fig. 1: Structure of our specification theory for real-time systems.

checker [5]. The purpose of this paper is to describe enrichments to our theory, and to report on the evaluation of the tool on a concrete case study. Our contributions are:

1. *An on-the-fly algorithm for checking Büchi objectives of two-player timed games.* The algorithm builds on an existing, efficient method for solving reachability objectives [8, 4], but it uses zones as a symbolic representation. We show how the method can be combined with a safety objective. This allows, for example, to guarantee that a player has a strategy to stay within a set of states without blocking the progress of time. Similar results were proposed by de Alfaro et al. [16] but for a restricted class of timed interfaces and without an implementation for the continuous case.
2. *A realistic case study.* Most existing interface theories have not been implemented and evaluated on concrete applications. We use ECDAR to show that our interface theory is indeed a feasible solution for the design of potentially complex timed systems. More precisely, we specify an infrared sensor for measuring short distances and for detecting obstructions. This extensive case study reveals both the advantages and disadvantages of our theory, which are summarized in this paper.

2 Background: Real Time Specifications as Games

Following [11], we now introduce the basic objects of this paper. Our specifications and models (implementations) are taken from the same class, timed games. They both exist in two flavors: infinite and finite. Fig. 1 summarizes this structure. The top–bottom division goes across the notion of satisfaction (models and specifications) and the left–right one across syntax-semantics (Timed I/O Transition Systems and Timed I/O Automata). This orthogonality is exploited to treat the intricacies of continuous time behaviour separately from those of algorithms. Roughly, the infinite models have been used to develop the theory, while the finite symbolic representations are used in the implementation.

Definition 1. *A Timed I/O Transition System (TIOTS) is a tuple* $S = (St^S, s_0, \Sigma^S, \rightarrow^S)$*, where* St^S *is an infinite set of states,* $s_0 \in St$ *is the initial state,* $\Sigma^S = \Sigma^S_{\text{i}} \oplus \Sigma^S_{\text{o}}$ *is a finite set of actions partitioned into inputs and outputs, and* $\rightarrow^S : St^S \times (\Sigma^S \cup \mathcal{R}_{\geq 0}) \times St^S$

is a transition relation. We write $s \xrightarrow{a}^S s'$ *instead of* $(s, a, s') \in \rightarrow^S$ *and use* $i?$, $o!$ *and* d *to range over inputs, outputs and* $\mathcal{R}_{\geq 0}$ *respectively. Also for any TIOTS we require:*

[time determinism] whenever $s \xrightarrow{d}^S s'$ *and* $s \xrightarrow{d}^S s''$ *then* $s' = s''$,

[time reflexivity] $s \xrightarrow{0}^S s$ *for all* $s \in St^S$, *and,*

[time additivity] for all $s, s'' \in St^S$ *and all* $d_1, d_2 \in \mathcal{R}_{\geq 0}$ *we have* $s \xrightarrow{d_1+d_2}^S s''$ *iff* $s \xrightarrow{d_1}^S s'$ *and* $s' \xrightarrow{d_2}^S s''$ *for some* $s' \in St^S$.

We write $s \xrightarrow{a}^S$ *meaning that there exists a state* s' *such that* $s \xrightarrow{a}^S s'$.

TIOTSs are abstract representations of real time behaviour. We use *Timed I/O Automata* (TIOAs) to represent them symbolically using finite syntax.

Let *Clk* be a finite set of *clocks*. $[Clk \mapsto \mathcal{R}_{\geq 0}]$ denotes the set of mappings from *Clk* to $\mathcal{R}_{\geq 0}$. A *valuation* over *Clk* is an element u of $[Clk \mapsto \mathcal{R}_{\geq 0}]$. Given $d \in \mathcal{R}_{\geq 0}$, we write $u+d$ to denote a valuation such that for any clock $r \in Clk$ we have $(u+d)(r) = x+d$ iff $u(r) = x$. We write $u[r \mapsto 0]_{r \in c}$ for a valuation which agrees with u on all values for clocks not in c, and gives 0 for all clocks in $c \subseteq Clk$. Let op be the set of relational operators: $\mathsf{op} = \{<, \leq, >, \geq\}$. A *guard* over *Clk* is a finite conjunction of expressions of the form $x \prec n$, where $\prec \in \mathsf{op}$ and $n \in \mathbb{N}$. We write $\mathcal{B}(Clk)$ for the set of guards over *Clk* using operators in the set op, and $\mathscr{P}(X)$ for the powerset of a set X.

Definition 2. *A* Timed I/O Automaton *(TIOA) is a tuple* $A = (Loc, q_0, Clk, E, Act, Inv)$ *where Loc is a finite set of locations,* $q_0 \in Loc$ *is the initial location, Clk is a finite set of clocks,* $E \subseteq Loc \times Act \times \mathcal{B}(Clk) \times \mathscr{P}(Clk) \times Loc$ *is a set of edges, Act is the action set* $Act = Act_{\mathrm{i}} \oplus Act_{\mathrm{o}}$, *partitioned into inputs and outputs respectively, and* $Inv : Loc \mapsto \mathcal{B}(Clk)$ *is a set of location invariants.*

If $(q, a, \varphi, c, q') \in E$ is an edge, then q is a source location, a is an action, φ is a constraint over clocks that must be satisfied when the edge is executed, c is a set of clocks to be reset, and q' is the target location. We will give examples of TIOAs in Sect. 4.

The expansion of the behaviour of a TIOA $A = (Loc, q_0, Clk, E, Act, Inv)$ is the following TIOTS $[\![A]\!]_{\text{sem}} = (Loc \times [Clk \mapsto \mathcal{R}_{\geq 0}], (q_0, \mathbf{0}), Act, \rightarrow)$, where $\mathbf{0}$ is a constant function mapping all clocks to zero, and $\rightarrow$ is generated by the two rules:

- Each $(q, a, \varphi, c, q') \in E$ gives rise to $(q, u) \xrightarrow{a} (q', u')$ for each clock valuation $u \in [Clk \mapsto \mathcal{R}_{\geq 0}]$ such that $u \models \varphi$ and $u' = u[r \mapsto 0]_{r \in c}$ and $u' \models Inv(q')$.
- Each location $q \in Loc$ with a valuation $u \in [Clk \mapsto \mathcal{R}_{\geq 0}]$ gives rise to a transition $(q, u) \xrightarrow{d} (q, u + d)$ for each delay $d \in \mathcal{R}_{\geq 0}$ such that $u + d \models Inv(q)$.

We refer to states and transitions of a TIOA, meaning the states and transitions of the underlying TIOTS. As stated above, these states are location–clock valuation pairs.

The TIOTSs induced by TIOAs conform to Def. 1. In addition, to guarantee determinism, for each action–location pair only one transition can be enabled at a time. This is a standard check. We assume that all TIOAs below are deterministic.

Implementations (models) are a subclass of specifications that are amenable to implementation. They have fixed timing behaviour (outputs occur at predictable times) and can always advance either by producing an output or delaying.

Definition 3. *A TIOA* A *is a* specification *if each state* $s \in St^{[\![A]\!]_{\text{sem}}}$ *is input-enabled:*

[input enabledness] $\forall\, i? \in \Sigma_{\mathrm{i}}^{[\![A]\!]_{\text{sem}}}.\ s \xrightarrow{i?} [\![A]\!]_{\text{sem}}$.

Definition 4. *An* implementation A *is a specification (so a suitable TIOA), where, in addition, for each state* $p \in St^{[\![A]\!]_{\text{sem}}}$ *the following two conditions hold:*

[output urgency] for each $o! \in \Sigma_{\text{o}}^{[\![A]\!]_{\text{sem}}}$ *if* $p \xrightarrow{o!} {}^{[\![A]\!]_{\text{sem}}}$ *and* $p \xrightarrow{d} {}^{[\![A]\!]_{\text{sem}}}$ *then* $d = 0$ *and,*

[independent progress] $(\forall d \geq 0.\, p \xrightarrow{d} {}^{[\![A]\!]_{\text{sem}}})$ *or*

$$(\exists\, d \in \mathcal{R}_{\geq 0}.\, \exists\, o! \in \Sigma_{\text{o}}^{[\![A]\!]_{\text{sem}}}.\, p \xrightarrow{d} {}^{[\![A]\!]_{\text{sem}}} p' \text{ and } p' \xrightarrow{o!} {}^{[\![A]\!]_{\text{sem}}})$$

Specifications are a subclass of TIOAs (the upper-left quadrant in Fig. 1) which induce TIOTSs that are input-enabled (the upper-right quadrant). Implementations are TIOAs (the lower-left quadrant) that induce both input-enabled and output-urgent TIOTSs able to progress independently (the lower-right quadrant). Although specifications and implementations are defined above by restricting their semantic properties, it is possible, although more clumsy, to rephrase these conditions syntactically and implement them in a tool. These are again standard checks.

A run ρ of a TIOTS S from its state s_1 is a sequence $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_n$ such that for all $i \in [1..n]$, $s_i \xrightarrow{a_i} s_{i+1}$ is a transition of S. We write $\mathsf{Runs}(s_1, S)$ for the set of runs of S starting in s_1, and $\mathsf{Runs}(S)$ for the set of runs starting from the initial state of S. We write $\mathsf{States}(\rho)$ for the set of states of S present in ρ and, if ρ is finite, $\mathsf{last}(\rho)$ for the last state occurring in ρ.

TIOAs are interepreted as two-player real-time games between the *output player* (the component) and the *input player* (the environment). The *input* plays with actions in Σ_{i} and the *output* plays with actions in Σ_{o}:

Definition 5. *A strategy* f *for the input (resp. output) player,* $k \in \{\text{i}, \text{o}\}$*, on the TIOA* A *is a partial function from* $\mathsf{Runs}([\![A]\!]_{\text{sem}})$ *to* $Act_{\text{i}} \cup \{\mathsf{delay}\}$ *(resp.* $Act_{\text{o}} \cup \{\mathsf{delay}\}$*) such that for every finite run* ρ*, if* $f(\rho) \in Act_k$ *then* $\mathsf{last}(\rho) \xrightarrow{f(\rho)} s'$ *for some state* s' *and if* $f(\rho) = \mathsf{delay}$*, then* $\exists d > 0.\, \exists s''$ *such that* $\mathsf{last}(\rho) \xrightarrow{d} s''$*.*

For a given strategy, we consider behaviors resulting from the application of the strategy to the TIOA, with respect to all possible strategies of the opponent:

Definition 6 (Outcome [15]). *Let* A *be a TIOA,* f *a strategy over* A *for the input player, and* s *a state of* $[\![A]\!]_{\text{sem}}$*. The* outcome $\mathsf{Outcome}_{\text{i}}(s, f)$ *of* f *from* s *is the subset of* $\mathsf{Runs}(s, [\![A]\!]_{\text{sem}})$ *defined inductively by:*

- $s \in \mathsf{Outcome}_{\text{i}}(s, f)$,
- *if* $\rho \in \mathsf{Outcome}_{\text{i}}(s, f)$ *then* $\rho' = \rho \xrightarrow{e} s' \in \mathsf{Outcome}_{\text{i}}(s, f)$ *if* $\rho' \in \mathsf{Runs}(s, [\![A]\!]_{\text{sem}})$ *and one of the following three conditions hold:*
  1. $e \in Act_{\text{o}}$,
  2. $e \in Act_{\text{i}}$ *and* $e = f(\rho)$,
  3. $e \in \mathcal{R}_{\geq 0}$ *and* $\forall\, 0 \leq e' < e.\, \exists s''.\, \mathsf{last}(\rho) \xrightarrow{e'} s''$ *and* $f(\rho \xrightarrow{e'} s'') = \mathsf{delay}$.
- $\rho \in \mathsf{Outcome}_{\text{i}}(s, f)$ *if* ρ *infinite and all its finite prefixes are in* $\mathsf{Outcome}_{\text{i}}(s, f)$

Let $\mathsf{MaxOutcome}_{\text{i}}(s, f)$ denote the maximal runs of $\mathsf{Outcome}_{\text{i}}(s, f)$, that is $\rho \in \mathsf{MaxOutcome}_{\text{i}}(s, f)$ iff $\rho \in \mathsf{Outcome}_{\text{i}}(s, f)$ and ρ has an infinite number of discrete actions, or ρ has a finite number of discrete actions, but there exist no $e \in Act \cup \mathcal{R}_{\geq 0}$ and no state s' with $\rho \xrightarrow{e} s' \in \mathsf{Outcome}_{\text{i}}(s, f)$, or the sum of the delays in ρ is infinite.

For a given TIOA A, a *winning condition* W for input is a subset of $\mathsf{Runs}([\![A]\!]_{\text{sem}})$. We say that W does not depend on the progress of the opponent (here output) iff whenever $\rho \in W$ and $\rho = \rho' \xrightarrow{e} \rho''$, with $e \in \mathit{Act}_{\text{o}}$, then either there exists $e' \in \mathit{Act}_{\text{i}}$, $d \in \mathcal{R}_{\geq 0}$, a state s and a run ρ''' such that $\rho' \xrightarrow{d} s \xrightarrow{e'} \rho''' \in W$ or there exists $d \in \mathcal{R}_{\geq 0}$ and some state s such that $\rho' \xrightarrow{d} s \in W$. This restriction means that input should always be able to ensure progress by itself and that the actions of the opponent should not be abused to advance the game, since we cannot assume that the opponent will ever make use of them. For a winning condition W, we write $\mathsf{Strip}(W)$ to denote the subset of W in which the runs not satisfying this condition are removed.

A pair (A, W) is an *input timed game*. Given a winning condition W for input, a strategy f of input is *winning* from state s if $\mathsf{MaxOutcome}(s, f) \subseteq W$. A state s is *winning* for input, if there exists a winning strategy for input from s. The game (A, W) is *winning* for input if the initial state of A is winning for it. For an input timed game (A, W), we write $\mathcal{W}_{\text{i}}(A, W)$ for the set of winning states for input and $\mathcal{F}_{\text{i}}(A, W, s)$ for all winning strategies for input from s. The winning conditions considered here are:

– Reachability objective: the input player must enforce a set Goal of "good" states. The corresponding winning condition is defined as

$$\mathit{WR}_{\text{i}}(\mathsf{Goal}) = \mathsf{Strip}\{\rho \in \mathsf{Runs}([\![A]\!]_{\text{sem}}) \mid \mathsf{States}(\rho) \cap \mathsf{Goal} \neq \emptyset\} \tag{1}$$

– Safety objective: the player must avoid a set Bad of "bad" states. The corresponding winning condition is defined as:

$$\mathit{WS}_{\text{i}}(\mathsf{Bad}) = \{\rho \in \mathsf{Runs}([\![A]\!]_{\text{sem}}) \mid \mathsf{States}(\rho) \cap \mathsf{Bad} = \emptyset\} \tag{2}$$

– Büchi objective: the player must enforce visiting Goal, a set of "good" states, infinitely often. Let $|A|$ denote the cardinality of set A. The winning condition is:

$$\mathit{WB}_{i}(\mathsf{Goal}) = \mathsf{Strip}\{\rho \in \mathsf{Runs}([\![A]\!]_{\text{sem}}) \mid \ |\mathsf{States}(\rho) \cap \mathsf{Goal}| = \infty\} \tag{3}$$

We define the outcomes $\mathsf{Outcome}_{\text{o}}(s, f)$ and $\mathsf{MaxOutcome}_{\text{o}}(s, f)$ of a strategy of the output player, as well as output timed games and all the related notions, by swapping 'i' and 'o' (for instance Act_{i} and Act_{o}) in the above definitions.

We now present discuss the *refinement relation*, which relates TIOTSs of two real time specifications, by determining which one allows more behaviour:

Definition 7. *A TIOTSs $S = (\mathit{St}^S, s_0, \Sigma, \rightarrow^S)$ refines a TIOTSs $T = (\mathit{St}^T, t_0, \Sigma, \rightarrow^T)$, written $S \leq T$, iff there exists a binary relation $R \subseteq \mathit{St}^S \times \mathit{St}^T$ containing (s_0, t_0) such that for each pair of states $(s, t) \in R$ we have:*

1. *if $t \xrightarrow{i?}{}^T t'$ for some $t' \in \mathit{St}^T$ then $s \xrightarrow{i?}{}^S s'$ and $(s', t') \in R$ for some $s' \in \mathit{St}^S$*
2. *if $s \xrightarrow{o!}{}^S s'$ for some $s' \in \mathit{St}^S$ then $t \xrightarrow{o!}{}^T t'$ and $(s', t') \in R$ for some $t' \in \mathit{St}^T$*
3. *if $s \xrightarrow{d}{}^S s'$ for $d \in \mathcal{R}_{\geq 0}$ then $t \xrightarrow{d}{}^T t'$ and $(s', t') \in R$ for some $t' \in \mathit{St}^T$*

A specification A_1 refines a specification A_2, written $A_1 \leq A_2$, iff $[\![A_1]\!]_{\text{sem}} \leq [\![A_2]\!]_{\text{sem}}$. If A_1 is an implementation then we also say that it satisfies A_2, written $A_1 \models A_2$.

Refinement between two automata may be checked by playing a safety game on the product of their two state spaces, avoiding the error states (where error states are pairs of states of S and T for which one of the above rules is violated). See details in [11, 13]. Since the product can be expressed as a TIOA itself, the refinement can be checked using the safety game as defined above.

Consider two TIOTSs $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ and $T = (St^T, s_0^T, \Sigma^T, \rightarrow^T)$. We say that they are *composable* iff their output alphabets are disjoint $\Sigma_o^S \cap \Sigma_o^T = \emptyset$. The *product* of S and T is the TIOTS $S \otimes T = (St^S \otimes St^T, (s_0^S, s_0^T), \Sigma^{S\otimes T}, \rightarrow^{S\otimes T})$, where the alphabet $\Sigma^{S\otimes T} = \Sigma^S \cup \Sigma^T$ is partitioned into inputs and outputs in the following way: $\Sigma_i^{S\otimes T} = (\Sigma_i^S \setminus \Sigma_o^T) \cup (\Sigma_i^T \setminus \Sigma_o^S)$, $\Sigma_o^{S\otimes T} = \Sigma_o^S \cup \Sigma_o^T$. The transition relation is generated by the following rules:

$$\frac{s \xrightarrow{a}{}^S s' \qquad a \in \Sigma^S \setminus \Sigma^T}{(s,t) \xrightarrow{a}{}^{S\otimes T} (s',t)}\,[\text{indep-l}] \qquad \frac{t \xrightarrow{a}{}^T t' \qquad a \in \Sigma^T \setminus \Sigma^S}{(s,t) \xrightarrow{a}{}^{S\otimes T} (s,t')}\,[\text{indep-r}]$$

$$\frac{s \xrightarrow{a}{}^S s' \qquad t \xrightarrow{a}{}^T t' \qquad a \in \mathcal{R}_{\geq 0} \cup \Sigma_i^{S\otimes T} \cup (\Sigma_i^S \cap \Sigma_o^T) \cup (\Sigma_o^S \cap \Sigma_i^S)}{(s,t) \xrightarrow{a}{}^{S\otimes T} (s',t')}\,[\text{sync}]$$

Let undesirable be a set of error states that violate a safety property (for example, an elevator engine running while its door is open). Two specifications are *useful* with respect to one another if there is an environment that can avoid undesirable states in their product. The existence of such an environment is established by finding a winning strategy in the game formed by the product automaton and the objective $WS_{\text{i}}(\text{undesirable})$.

The parallel composition of S and T is defined as $S \mid T = \mathit{prune}(S \otimes T)$, where the prune operation removes from $S \otimes T$ all states which are not winning for the input player in the game $(S \otimes T, WS_{\text{i}}(\text{undesirable}))$. Parallel composition is defined for TIOTSs induced by both specifications and implementations. A similar construction can be given directly for specifications and implementations on the syntactic level [11].

In [11] we give constructions for two other operators computed as winning strategies in timed games. For TIOAs (TIOTSs) B and C we define conjunction $B \wedge C$, which computes an automaton representing shared implementations of B and C, and also quotient $B \setminus C$, which computes a specification describing implementations that when composed with C give a specification refining B. Rather than define these operations explicitly we characterize their essential properties, and refer the reader to [11] for precise details of the constructions. Let A be an implementation. Then:

$$A \models B \wedge C \quad \text{iff} \quad A \models B \text{ and } A \models C \tag{4}$$

$$A \models B \setminus C \quad \text{iff} \quad C \mid A \leq B \tag{5}$$

3 Büchi Objectives

Symbolic On-The-Fly Timed Reachability (SOFTR) [8] is an efficient algorithm for solving two-players reachability timed games used in UPPAAL-TIGA [4]. It operates on the simulation graph induced by a TIOA representing the game. It follows an established principle: begin with all reachable states and propagate the winning states backwards. Its major contribution is the use of zones rather than regions. Zones, which

are unions of regions of Alur and Dill [3], are the most efficient representation of clock valuations known to date. In the following we recall SOTFTR, extend it to solve Büchi objectives, and provide a new algorithm to verify Büchi and safety objectives combined.

3.1 Solving Büchi Games with SOTFTR

For a TIOTS S and a set of states X, write $\mathsf{Pred}_a(X) = \{s \in St \mid \exists s' \in X.\ s \xrightarrow{a} s'\}$ for the set of all a-predecessors of states in X. We write $\mathsf{iPred}(X)$ for the set of all input predecessors, and $\mathsf{oPred}(X)$ for all the output predecessors of X, so $\mathsf{iPred}(X) = \bigcup_{a \in \Sigma_{\mathrm{i}}^S} \mathsf{Pred}_a(X)$ and $\mathsf{oPred}(X) = \bigcup_{a \in \Sigma_{\mathrm{o}}^S} \mathsf{Pred}_a(X)$. Also $\mathsf{post}_{[0,d_0]}(s)$ is the set of all time successors of a state s that can be reached by delays less than or equal to d_0: $\mathsf{post}_{[0,d_0]}(s) = \{s' \in St^S \mid \exists\, d \in [0, d_0].\ s \xrightarrow{d}{}^S s'\}$. The safe timed predecessors of a set X relative to an unsafe set Y are the states from which a state in X is reached after a delay while avoiding any of the states in Y (the subscript t in the definition of $\mathsf{cPred}_{\mathsf{t}}$ below indicates that these are timed predecessors only):

$$\mathsf{cPred}_{\mathsf{t}}(X, Y) = \{s \in St^S \mid \exists\, d_0 \in \mathcal{R}_{\geq 0}.\ \exists s' \in X. s \xrightarrow{d_0}{}^S s' \text{ and } \mathsf{post}^S_{[0,d_0]}(s) \subseteq \overline{Y}\}$$

Let A be a TIOA and G a set of "good" states in $[\![A]\!]_{\text{sem}}$ that have to be reached, that is the objective is $WR_{\mathrm{i}}(G)$. Consider the following computation [21, 8]:

$$\begin{array}{l} H_0 \leftarrow \emptyset \\ \textbf{repeat}\ H_{k+1} \leftarrow H_k \cup \pi_{\mathrm{i}}(H_k) \cup G\ \textbf{for}\ k = 0, 1, \ldots \\ \textbf{until}\ H_{k+1} = H_k \end{array}$$

where $\pi_{\mathrm{i}}(H) = \mathsf{cPred}_{\mathsf{t}}(\mathsf{iPred}(H), \mathsf{oPred}(\mathsf{States}(\mathsf{Runs}([\![A]\!]_{\text{sem}})) \setminus H))$. The π_{i} operator computes the predecessors of set H that can enforce H in one step, regardless of what the output player does. This is done by taking timed predecessors of input-predecessors of H, as long as we can avoid output predecessors of states outside H. The fixpoint of π_i is the set of states in which the input player can enforce reaching G eventually [21, 8]. SOTFTR is a symbolic zone-based implementation of the above fixpoint.

The winning states of the output player can be computed by replacing π_{i} with $\pi_{\mathrm{o}}(H) = \mathsf{cPred}_{\mathsf{t}}(\mathsf{oPred}(H), \mathsf{iPred}(\mathsf{States}(\mathsf{Runs}([\![A]\!]_{\text{sem}})) \setminus H))$. Thus, in the remainder, we focus on solving the game for the input player only.

The following algorithm for solving Büchi timed games is an adaptation of the above procedure given in [21], adjusted for a TIOA A and a Büchi objective. The set of "good" states, Goal, is to be enforced infinitely often:

$$\begin{array}{l} W_0 \leftarrow \mathsf{States}(\mathsf{Runs}([\![A]\!]_{\text{sem}})) \\ \textbf{for}\ j = 0, 1, \ldots\ \textbf{repeat} \\ \quad H_0 \leftarrow \emptyset \\ \quad \textbf{repeat}\ H_{k+1} \leftarrow H_k \cup \pi_{\mathrm{i}}(H_k) \cup (\mathsf{Goal} \cap \pi_{\mathrm{i}}(W_j))\ \textbf{for}\ k = 0, 1, \ldots \\ \quad \textbf{until}\ H_{k+1} = H_k \\ \quad W_{j+1} \leftarrow H_k \\ \textbf{until}\ W_{j+1} = W_j \end{array}$$

Observe that a Büchi objective is essentially a closure of a reachability objective: it corresponds to finding a subset of "good" Goal states, from which reachability to the

good subset again is guaranteed for the player, and then solving for reachability of that good subset. In the above computation, the inner loop finds states that can enforce a Goal state in at least one discrete step, and uses this information to determine which Goal states are actually "good" (the intersection with Goal). The outer loop removes the Goal states that are not "good" from the target set of the inner loop. In the fixpoint, we find both the subset of good Goal states and the states from which this subset can be reached regardless of what the opponent does.

SOTFTR itself computes the inner loop of this algorithm when $G = \mathsf{Goal} \cap \pi_{\mathrm{i}}(W_j)$, this observation leads to the *Symbolic Timed Büchi* games (STB) algorithm:

$W_0 \leftarrow \mathsf{States}(\mathsf{Runs}([\![A]\!]_{\mathrm{sem}}))$
repeat $W_{j+1} \leftarrow \mathrm{SOTFTR}(\mathsf{Goal} \cap \pi_{\mathrm{i}}(W_j))$ **for** $j = 0, 1, \ldots$
until $W_{j+1} = W_j$

Observe that STB uses exactly the same operations on zones as SOTFTR, which means that it can also be implemented in an efficient manner.

Theorem 1 ([8, 21]). *For any input Büchi timed game* $(A, \mathit{WB}_{\mathrm{i}}(\mathsf{Goal}))$*, STB terminates and upon termination* $W_j = \mathcal{W}_{\mathrm{i}}(A, \mathit{WB}_{\mathrm{i}}(\mathsf{Goal}))$.

The algorithm of [21] computes over infinite sets of states. Our algorithm is nothing more than a symbolic implementation of the original one. By construction and because of [8], the above correspondence is obtained directly. Termination is shown in [21].

3.2 Combining Safety and Büchi objectives

We now strengthen the Büchi objective so that not only the Goal states are visited infinitely often, but also the set of unsafe states Bad is avoided ($\mathsf{Bad} \cap \mathsf{Goal} = \emptyset$):

$$\mathit{WBS}(\mathsf{Goal}, \mathsf{Bad}) = \mathsf{Strip}\{\rho \in \mathsf{Runs}([\![A]\!]_{\mathrm{sem}}) \mid \mathsf{States}(\rho) \cap \mathsf{Bad} = \emptyset \text{ and } |\mathsf{States}(\rho) \cap \mathsf{Goal}| = \infty\} \quad (6)$$

One application of such games is ensuring that the input player has a strategy to avoid Bad while ensuring that time is elapsing [16], eliminating the so called Zeno-behaviours.

If Bad can be expressed as a finite union of pairs of locations and finite unions of zones, then this objective can be reduced to the usual Büchi objective by transforming the game in the following way: (i) add a location $B \notin \mathsf{Goal}$; (ii) add an *output* action $\mathit{err} \notin \mathit{Act}_{\mathrm{i}}$; (iii) for each pair $(q, \bigcup_{i=1..n} Z_i) \in \mathsf{Bad}$ such that q is a location of A and $\bigcup_{i=1..n} Z_i$ is a finite union of zones, add n edges E_i $(i = 0..n)$ labelled by *err* from q to B such that for all i, the guard of E_i is Z_i. Since location B has no outgoing edges and does not belong to Goal, entering B means losing the Büchi game. Suppose we want a winning strategy for the input player. Observe that the added edges belong to the opponent. By definition of outcomes, going through any state in Bad means that one of these edges can now be taken by the output player and, as $B \notin \mathsf{Goal}$, the game is lost for the input player. The following theorem expresses the correctness of our transformation.

Theorem 2. *Let* $(A, \mathit{WBS}_{\mathrm{i}}(\mathsf{Goal}, \mathsf{Bad}))$ *be a TIOA, and* A' *be its modification obtained by the above construction. Then* $\mathcal{F}_{\mathrm{i}}(A, \mathit{WBS}_{\mathrm{i}}(\mathsf{Goal}, \mathsf{Bad})) = \mathcal{F}_{\mathrm{i}}(A', \mathit{WB}_{\mathrm{i}}(\mathsf{Goal}))$

Proof. Show that $\mathcal{F}_i(A, \mathit{WBS}_{\mathrm{i}}(\mathsf{Goal}, \mathsf{Bad})) \subseteq \mathcal{F}_i(A', \mathit{WB}_{\mathrm{i}}(\mathsf{Goal}))$. Let f be a strategy in $\mathcal{F}_i(A, \mathit{WBS}_{\mathrm{i}}(\mathsf{Goal}, \mathsf{Bad}))$ and s_0 be the initial state of A and s_0' of A'. As f is winning, no run in $\mathsf{MaxOutcome}_{\mathrm{i}}(s_0, f)$ goes through a state in Bad. By construction of A' we have that no run in $\mathsf{MaxOutcome}_{\mathrm{i}}(s_0', f)$ goes through Bad and therefore the guards of the extra edges in A are never satisfied. Since, apart from these edges, A' is identical to A, and since f ensures infinite repetition of Goal in A, then it does also in A'.

Now, show $\mathcal{F}_i(A, \mathit{WBS}_{\mathrm{i}}(\mathsf{Goal}, \mathsf{Bad})) \supseteq \mathcal{F}_i(A', \mathit{WB}_{\mathrm{i}}(\mathsf{Goal}))$. Let f be a strategy in $\mathcal{F}_i(A, \mathit{WBS}_{\mathrm{i}}(\mathsf{Goal}, \mathsf{Bad}))$ and s_0 be the initial state of A, and s_0' of A'. The runs of A' that go to location B are maximal and cannot belong to $\mathit{WB}_{\mathrm{i}}(\mathsf{Goal})$ for B has no outgoing edge. Let ρ be a run in $\mathsf{MaxOutcome}_{\mathrm{i}}(s_0', f)$ and $\rho = \rho' \to s \to \rho''$, and the guard of one of the *err* edges is satisfied in s. Then $\rho' \xrightarrow{err} (B, v)$ for some valuation v is a maximal run and thus belongs to $\mathsf{MaxOutcome}_{\mathrm{i}}(s_0', f)$ and then $\mathsf{MaxOutcome}_{\mathrm{i}}(s_0', f) \not\subseteq \mathit{WB}_{\mathrm{i}}(\mathsf{Goal})$ which contradicts that f is winning. So the runs in $\mathsf{MaxOutcome}_{\mathrm{i}}(s_0', f)$ never go through states in Bad. Furthermore, since A and A' are identical except for B and its incoming edges, it must then be that $\mathsf{MaxOutcome}_{\mathrm{i}}(s_0, f) = \mathsf{MaxOutcome}_{\mathrm{i}}(s_0', f)$ and so the runs in $\mathsf{MaxOutcome}_{\mathrm{i}}(s_0, f)$ also repeat Goal infinitely often. □

An Application: eliminating Zeno Strategies. Consider a TIOA A and a set Bad of bad states. Our objective is to find the set of states from which the input player (symmetrically the output player) has a strategy to avoid Bad while letting time elapse — as opposed to, for example, taking infinitely many discrete transitions without any delays.

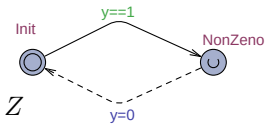
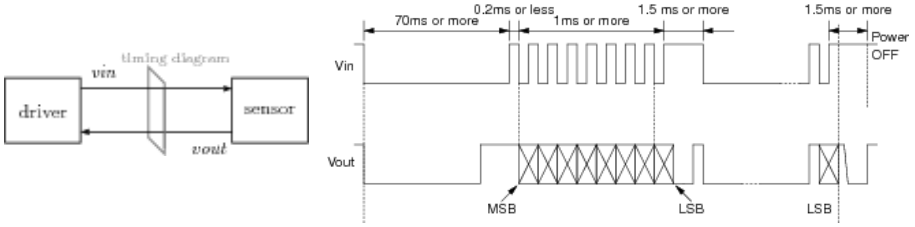


Fig. 2: Monitor for non-zeno strategies

In order to generate non-zeno strategies consider the product $A \times Z$ of A and the TIOA Z of Fig. 2. Then solve the timed game $(A \times Z, \mathit{WBS}_{\mathrm{i}}(\mathsf{Goal}, \mathsf{Bad}))$, where Goal is the set of states of $A \times Z$ in which Z is in location NonZeno. To fulfill this objective, the input player needs to avoid Bad *and* ensure that NonZeno is visited infinitely often: once in NonZeno, the only way to revisit it is to pass through Init. This loop requires that 1 time unit elapses, so repeated visits to NonZeno ensure that time progresses.

Note that this does not prevent the opposing player from using a spoiling strategy producing zeno runs to prevent fulfillment of the objective.

Remark 1. One problem with the above setup is the effect of adding self-loops. Our interface theory requires TIOA to be input-enabled. This means that, in any state of the game, the input player should always be able to react on any of the input actions. This typically means that states have implicit loops on input actions when the designer does not specify any other transition for an input. Now, assume that the output player wants to win the game and guarantee that time elapses. The input player could always play such an input-loop and hence block time. This means that the potential addition of arbitrary inputs may corrupt the game. A solution to the above problem is to blame the input player each time it plays [16]. Then, the input player loses the game if there is a point of time after which it is blamed forever. De Alfaro et al. were the first to use blames [16]. We can also add a monitor for the blame situation. Another solution, in order to avoid adding an extra automaton, is to use a counter in ECDAR to bound the number of Inputs (Outputs) that can be played successively.



(a) Logical interface (b) Timing diagram; modified slightly from [22]

Fig. 3: The driver/sensor system

4 Case Study

The ideas just presented have been implemented in the tool ECDAR [10], which supports graphical modeling of TIOAs, computing composition operators (including quotienting), and reachability analysis. For this paper, we have extended ECDAR with support for Büchi and Büchi with safety objectives. We apply it to the analysis of a simple but realistic example: a sensor component and the software required to interface with it.[5] The case study serves both to elucidate some of the technical definitions and to demonstrate their practicability.

4.1 Timing diagram model

The Sharp GP2D02 infrared sensor is a small component for measuring short distances and for detecting obstructions. Such sensors are incorporated into larger embedded systems through two communication wires which carry a protocol of rising and falling voltage levels. The four main components of a sensor subsystem are shown in Fig. 3a: an instance of the *sensor*, a *driver* component of a larger system, a *vin* wire controlled by the driver and read by the sensor, and a *vout* wire controlled by the sensor and read by the driver. The communication protocol between driver and sensor is described by the timing diagram of Fig. 3b.

The timing diagram describes the permissible interactions between a driver and a sensor. It represents a (partial) ordering of events and the timing constraints between them. With careful interpretation, against a background of engineering practice, the timing diagram can be modeled as the TIOA shown in Fig. 4 and henceforth called T. Note that constants are multiples of 0.1 ms, so the constant 0.2 ms in the timing diagram is represented by an integer constant 2 in the model. This model is the result of several choices and its fidelity can only be justified by informal argument [6, Chapter 4].

We now step through the timing diagram and the TIOA model in parallel describing the meaning of the former and justifying the latter. The interaction of driver and sensor is essentially quite simple: the driver requests a range reading, then after a brief delay the sensor signals that a reading has been made, the driver triggers the sensor to transmit

[5] See http://www.tbrk.org/papers/wadt10.tar.gz for the implementation in ECDAR

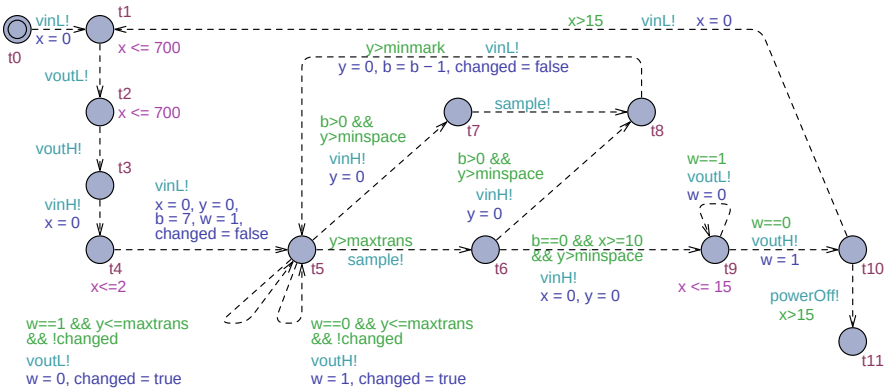


Fig. 4: TIOA model of the timing diagram: T

the reading bit by bit, and, finally, the process is repeated or the sensor is powered off. The interaction takes place solely over the two communication wires.

The signal controlled by the driver is shown in the top half of the timing diagram. Its most obvious features are the falling and rising transitions, these have been modeled in the TIOA as outputs called, respectively, vinL! and vinH!. The driver may also perform two other actions which are not entirely evident from the timing diagram. It may sample the *vout* signal to read a bit transmitted by the sensor, which we represent by an output called sample!, and it may stop using the sensor, which we represent by an output called powerOff!. The signal controlled by the sensor is shown in the bottom half of the timing diagram. The rising and falling transitions on this signal are modeled as outputs called, respectively, voutL! and voutH!. In fact, all of the actions in the model are outputs because the timing diagram describes a closed system. The model is thus trivially input-enabled and there is no need for self-looping input transitions on each state. Furthermore, the model can be simulated in isolation since all channels in ECDAR must be broadcast channels (i.e. outputs are non-blocking).

The driver requests a range reading with vinL!, i.e. by lowering the voltage level of *vin*. The sensor responds with voutL!, it then performs the necessary measurements before signaling completion with voutH!. The timing diagram guarantees that the sensor will complete a reading and respond after at least *70ms or more* have passed, after which the driver may perform a vinH!. This sequence can be seen in the model in the transitions linking states T_0–T_4. We model the timing constraint by resetting a clock x when the initial vinL! occurs, and adding the location invariant $x \leq 700$ to states T_1 and T_2. By rights this invariant should be strict, i.e. $x < 700$, but this is not currently permitted in ECDAR.For strict compliance with the timing diagram we should also add the guard $x > 700$ to the vinH! transition between T_3 and T_4, in practice, however, there are implementations that do not wait the full $700\,\text{ms}$ but rather respond to voutH!. Both possible behaviors will be examined more closely in the next subsection.

After a reading has been made, the driver transfers the eight bits of the result from the sensor, from the most (MSB) to the least (LSB) significant bit. For each bit, the sensor sets the level of *vout* according to the value being transmitted, hence the ‘crossed

blocks' in Fig. 4. The timing diagram could be more precise about the details, but in our interpretation the driver triggers the next bit value with a vinL!, the sensor responds within a bounded time, and then the sensor may sample! the value and reset *vin* with a vinH!, in any order, before the next bit is requested. The triggering vinL! appears in the model from T_4 for the first bit and from T_8 for subsequent bits. The first action must occur in *0.2ms or less*, hence the invariant on T_4. The associated transition resets two clocks: x, for enforcing the *1ms or more* constraint across cycles, and y, for conditions on response times within each cycle. It also sets three variables: b, for counting the number of bits transmitted, w, for monitoring the level of *vout*, and changed, for limiting oscillations on *vout*. We use the w variable to ensure the strict alternation of voutL! and voutH!, an alternative approach is shown later. Two other constants appear around the loop T_5–T_8: maxtrans is a limit on the time it takes for *vout* to change after a triggering vinL!, and minspace is the minimum width of pulses on *vin*. We set both constants to zero for this case study.

Finally, after transmitting eight bits, the driver and sensor return their respective wires to a high level, and, after *1.5ms or more*, either another reading is requested, or the sensor is powered off. The timing constraint is expressed as an invariant on $T9$, i.e. a guarantee on the behavior of the sensor, and guards on the transitions from $T10$, i.e. a constraint on the behavior of the driver. The invariant is right-closed and the guards are left-open for the same reasons given above for the $700\,\text{ms}$ constraint. Importantly, they do not overlap, so that time alone can be used to enforce the ordering between sensor and driver actions.

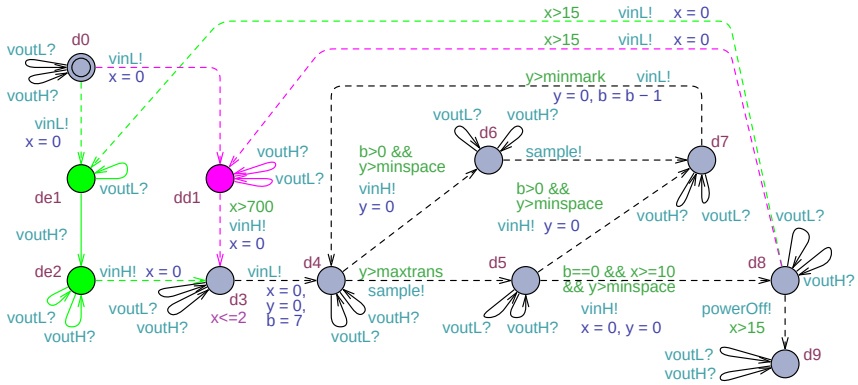
ECDAR is used to verify that the model is a valid (deterministic) specification, and also that it is consistent, i.e. that it has at least one valid implementation. We can also show two basic properties of the timing diagram model. The first, that vinL! and vinH! alternate strictly, is expressed using the automaton V^{in}, shown in Fig. 5b, and verified by the refinement $T \leq V^{in}$. The second, that voutL! and voutH! alternate strictly, is shown similarly using V^{out}, shown in Fig. 5c, and the refinement $T \leq V^{out}$. In fact, both properties can also be shown, using composition, by the single refinement $T \leq (V^{in} \mid V^{out})$.

4.2 Separate driver and sensor models

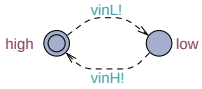
While the single automaton model of the previous section is a suitable formalization of the timing diagram, there are at least two motivations for creating separate but interacting models for the roles of driver and sensor. First, this separation emphasizes the distinct behaviors of each and clarifies their points of synchronization; each of the two wires is, in effect, modeled separately. Second, each of the models may be used in isolation. This possibility is exploited in an appendix of the full version of this paper where a separate driver model serves as the specification for a model of an implementation in assembly language.

The components of the models are shown in Fig. 5. We discuss the driver models first, then the sensor, before relating them all to the timing diagram model.

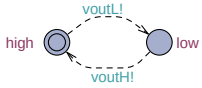
The driver model. As previously mentioned, there are two ways for a driver to behave after it has requested a range reading: it can wait for a rising transition on the *vout* wire,



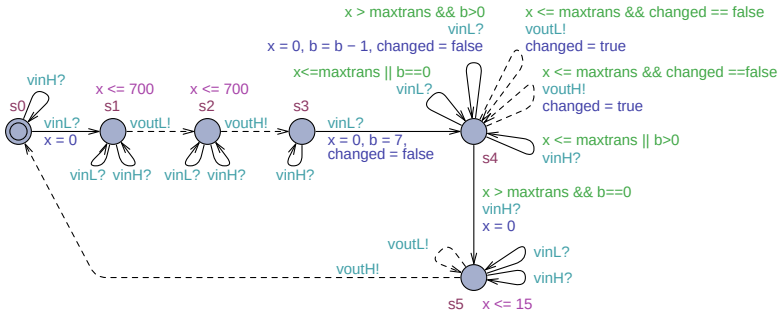
(a) Driver models: D^{ev} and D^{de}



(b) Driver property: V^{in}



(c) Auxiliary sensor model: V^{out}



(d) Main sensor model: S

Fig. 5: Sensor and driver models

or it can just wait 700 ms regardless. We model each possibility separately, both models shown in Fig. 5a. The model that responds to the sensor event is called D^{ev}, it comprises all locations except the one labeled *dd1*, which should be ignored together with all of its incoming and outgoing transitions. The model that always delays is called D^{de}, it comprises all locations except those labeled *de1* and *de2* whose connected transitions are also excluded. The models cannot be combined without introducing non-determinism.

Aside from these initial differences the two models behave identically and their structures resemble that of the timing diagram model except that events on *vout* from the sensor are now modeled as the input actions voutL? and voutH?, and a counterpart for the state T_9 is not required. We explicitly model input-enabledness by adding self-loops, which, although not mandatory, since actions occur on broadcast channels,

are necessary in ECDAR for verifying refinement. Note that both driver variants require little interaction with the sensor, relying instead on timing assumptions to ensure synchronization. In fact only D^{ev} reacts to sensor events directly, through the voutH? transition between D_1^{ev} and D_2^{ev}, though both models do sample the level of *vout*.

Refinement can be used to show a basic property of both driver models, that vinL! and vinH! alternate. This property is expressed as the automaton V^{in}, shown in Fig. 5b, and we use ECDAR to show $D^{ev} \leq V^{in}$ and $D^{de} \leq V^{in}$.

We would also like to claim that D^{de} refines D^{ev}, i.e. that $D^{de} \leq D^{ev}$, since D^{ev} can always wait after receiving voutH?, but ECDAR rejects this claim since D^{de} does not guarantee that voutH? will precede its initial vinH!. In fact, this type of refinement can only be shown in a conditional form where assumptions on the environment are made explicit. We revisit this idea after presenting a model for the sensor that embodies sufficient assumptions.

The sensor model. The sensor model S is shown in Fig. 5d. Events on the *vin* wire are now modeled as the inputs vinL? and vinH?, with additional self-loops on certain states, and the outputs sample! and powerOff! are not needed. The initial segment, S_0–S_3, mimics the corresponding part of the timing diagram model, but the clocking loop is reduced to a single location S_4 with five self-looping transitions and one outgoing transition.

In location S_4, the effect of the inputs, vinL? and voutL?, depends on the time elapsed since the last request for a bit, as measured by the clock x, and the number of bits remaining to transmit, as tracked by the counter b. The input vinL?, which requests the next bit, is ignored if it occurs (again) within the period given to the sensor to set the level of *vout*, and also when all bits have been transmitted, i.e. when $b = 0$. The input vinH! is ignored until all bits have been transmitted at which time, provided maxtrans units have elapsed since the last vinL?, it triggers an exit from S_4. The outputs voutL! and voutH! may only occur within maxtrans units of the last vinL?, and, furthermore, only at most one output may occur within any cycle, that is between any two successive and 'legal' vinL?s. The former constraint is expressed in the clause $x \leq$ maxtrans, and the latter using the variable changed.

Instead of a changed variable, an earlier model [6, Figure 4.16] has two states with three transitions from the first (changed $=$ tt) to the second (changed $=$ ff): one labeled with voutL!, another with voutH!, and the last unlabeled. This last τ-transition marks the possibility that the sensor decides not to change the voltage level, which occurs when two consecutive bits of a range reading are identical. Besides being more explicit, the two-state version is also more liberal since it is ready to accept vinH? and vinL? as soon as the value of *vout* has been set. Even with maxtrans $= 0$ there is a difference since in the current model there is always a non-zero delay after a triggering vinL! before subsequent vinL! or vinH! actions can influence the sensor. In any case, τ-steps are not permitted in TIOA and replacing them with an explicit output only makes modeling awkward, and, moreover, it is unnecessary since the driver models always wait and never respond immediately to vinL! or vinH! whose occurrence is a sufficient but not necessary indication of a stable value on *vout*.

The sensor model as it stands allows arbitrary interleaving of voutL! and voutH!. This is in contrast to the timing diagram model of Fig. 3b, where a variable, w, tracks

D^{de}.vinL!		Attacker plays outputs on left of $\leq$
	D^{ev}.vinL!	Defender's response on right of $\leq$
D^{de} waits 701 ms		Attacker may delay on left of $\leq$
	D^{ev} waits 701 ms	Defender's response on right of $\leq$
D^{de}.vinH!		Attacker plays outputs on left of $\leq$
	no response	Defender loses!

Table 1: Counterexample for $D^{de} \leq D^{ev}$

the level of *vout*, or effectively which of voutL! or voutH! occurred most recently, and is used to constrain output events. The required alternating behavior is recovered using the conjunction operator and the TIOA V^{out}, depicted in Fig. 5c, giving the complete sensor specification: $(S \wedge V^{out})$. Here, the conjunction operator obviates the need to update and query a state variable on multiple transitions. A specific constraint is expressed in a localized and obvious form and the rest of the model can be constructed under the assumption that it will hold. In ECDAR, the two automata, S and V^{out}, execute in parallel and must synchronize on voutL! and voutH!, neither of which may occur otherwise. Unlike for the timing diagram and the driver models, there is no need to separately verify the alternation of outputs—it is guaranteed by construction.

Relations between the models. Now that we have a few different models, we turn our attention to their interrelationships. It turns out that one of the driver models is more general than the other under certain assumptions. After verifying that fact, we turn our attention to validating the composition of the driver and sensor models against the timing diagram model. We also consider how the quotient operator might be applied.

The two driver models differ only in their initial interaction with the sensor, after requesting a range reading, D^{de} always waits 700 ms whereas D^{ev} may respond as soon as the sensor raises *vout*. One could thus suppose that D^{ev} is more general than D^{de}, since it can also refuse to act before 700 ms has passed even after receiving a voutH!. But, as described earlier, a first, naive attempt to show the refinement $D^{de} \leq D^{ev}$ fails! The counter-example strategy can be simulated in ECDAR, giving the results shown in Table 1. There is no guarantee that the inputs needed by D^{ev} will be provided. We must make these assumptions on the environment explicit by instead stating the relation as

$$\left(D^{de} \mid (S \wedge V^{out})\right) \leq \left(D^{ev} \mid (S \wedge V^{out})\right),$$

which is readily validated by ECDAR.[6] The verification fails if D^{de} and D^{ev} are swapped: D^{ev} can perform a vinH? when $x \leq 700$ while D^{de} cannot.

The compositions of the driver and sensor models have been proposed as alternatives to the timing diagram model. We state this, for the more general driver model, as two properties: $(D^{ev} \mid (S \wedge V^{out})) \leq T$, and $T \leq (D^{ev} \mid (S \wedge V^{out}))$. Both of which are verified almost instantaneously by ECDAR. For the similar properties with D^{de} instead of D^{ev}, only the version with T on the right of the refinement holds; as would be expected.

[6] In the current version of ECDAR, S and V^{out} must be explicitly duplicated.

Even ignoring the conjunction operator, the possibility of verifying a refinement with a composition on the right-hand side is interesting, because it is not possible in any other existing tools for checking timed automata refinement. For instance, current implementations [7] of the usual construction for checking timed trace inclusion [18, 23] require that the refined specification is an explicit automaton. The capability to address compositions is one advantage of incorporating the refinement verification into the model-checker itself.

There are limited opportunities to apply the quotient operator in this case study, perhaps because there are only a small number of models and the operators are not nested in especially complicated ways. There are, though, two types of properties that may be attempted.

The first type of property uses the quotient on the right-hand side of a refinement instead of composition on the left-hand side. For instance, we can verify $D^{ev} \leq (T \setminus (S \wedge V^{out}))$ in ECDAR. The right hand side expresses the idea of the timing diagram modulo certain assumptions on the environment. Currently the tool requires the explicit definition of universal and inconsistent states when using the quotient operator, and simulations are not possible. These issues will be addressed in future versions.

Second, we could try the quotient on the left-hand side of a refinement. For instance, to propose the property $(T \setminus D^{ev}) \leq (S \wedge V^{out})$ as a means of finding out whether the sensor model is maximal with respect to the timing diagram and driver model. This cannot work in general, however, since as soon as D^{ev} cannot do an output from a state, like vinH! from the initial state for example, the quotient will have a transition to the universal state from which any output or delay can be chosen, at any time, to challenge the other side of the refinement.

Büchi objectives. Some aspects of specifying liveness are addressed by the algorithms presented earlier, and supported in ECDAR. It is possible, for example, to determine whether a given combination of a TIOA and a liveness constraint, expressed as a Büchi objective, are consistent; i.e. whether refinement is possible. But other important aspects are not yet addressed satisfactorily. Most notably, the interaction of Büchi constraints and refinement is limited.

Büchi objectives offer a way to further constrain specifications. For example, consider adding an additional requirement to the timing diagram model T: if an initial range reading is requested, the system must eventually be powered off. We will interpret this to mean that two behaviors are allowed: 1. resting forever in T_0, or, 2. terminating in T_{11}. Our first attempt is to simply try to solve a Büchi objective for the current model: $(T, \mathit{WB}(\{T_0, T_{11}\}))$. But this is not correct, and ECDAR reports that the model is inconsistent. While the model starts in T_0, and T_{11} is always reachable, the Büchi objective is only satisfied if either of T_0 or T_{11} is reentered infinitely often. Self-looping output transitions must be added to T_0 and T_{11} to allow 'resting' in these states. If we do this—choosing an arbitrary output that will not occur in any other models—and call the modified version T', ECDAR confirms that $(T', \mathit{WB}(\{T'_0, T'_{11}\}))$ is consistent.

The modified model is easily adapted to allow a system that never stops taking range readings: $(T', \mathit{WB}(\{T'_0, T'_{10}, T'_{11}\}))$. This model is obviously consistent since increasing the set of states in the Büchi objective cannot reduce the set of possible implementations. More information can be gained by verifying the consistency of $(T', \mathit{WB}(\{T'_{10}\}))$,

which confirms that the model allows unbounded repetitions of the protocol. Compliance with the Büchi objective is achieved by pruning away the transition labelled powerOff!, so this verification does not show that the unadorned model T' does not allow termination, only that the model can choose to cycle continuously. Verifying the consistency of a model with a Büchi objective can be useful as a sanity check.

While Büchi objectives in ECDAR are quite useful for checking consistency properties, they work less well in combination with refinement. For instance, in ECDAR we can show $(T', \mathit{WB}(\{T'_0\})) \leq (T', \mathit{WB}(\{T'_{10}\}))$.

This is indeed correct, since any implementation of the left-hand side is also an implementation of the right-hand side, but it could be considered misleading, since the left-hand side specifies a system that never starts a range reading, while the right-hand side could be interpreted as specifying a system that never stops performing range readings whereas, in fact, it is a system where it is possible, but not strictly necessary, to keep performing range readings. The source of this mismatch is that the current refinement is based on partial observations rather than complete ones, which is adequate for safety but not for liveness.

The pruning of output transitions that can result from the combination of a TIOA and a Büchi objective gives models where a constraint that is supposedly on infinite behaviors also constrains finite behaviors, which, while not necessarily bad, is perhaps not completely reasonable [1]. The methodological implications for our theory are not yet clear, but we note here that this situation can be detected using refinement verification in ECDAR. The *machine closure* [2] of a TIOA A and a Büchi objective B can be checked by the refinement $A \leq (A, B)$, which will fail if a reachable output transition in A is not present in (A, B).

5 Summary and Future Work

We have shown that ECDAR and the underlying theory, are powerful enough to handle a small—in terms of the scale of systems developed by industry—but realistic case study. The input/output semantics of TIOA works well for open systems, and the game-based refinement semantics, i.e. the idea of challenging with inputs from the right-hand side and outputs or delays from the left-hand side, quickly comes to seem natural. Including refinement testing in the model checker itself is much more convenient than having to pass models through an external tool, and the concomitant feature of allowing composed models on either side of the relation is a powerful one. Finally, the conjunction operator is a very convenient modeling feature.

Still, several elements could be improved. While Büchi objectives are currently not without use, a different notion and implementation of refinement is needed to support more sophisticated applications. The quotient operator is supported by ECDAR, but its effect is not easily visualized or simulated. More work is needed to determine how it can be usefully applied to system development and verification; the sensor case study is too limited in this regard. ECDAR takes advantage of the mature UPPAAL user interface, but strategies, goals, and the effect of pruning are inherently more complicated and harder to understand than are simple traces, more work is needed to understand how best to compute and communicate this information. Furthermore, the new operators and

analyses available in ECDAR make it natural to work with multiple pairings of system declarations and properties, but this is not yet well supported by the user interface.

References

1. M. Abadi, B. Alpern, K. R. Apt, N. Francez, S. Katz, L. Lamport, and F. B. Schneider. Preserving liveness: Comments on "Safety and liveness from a methodological point of view". *Information Processing Letters*, 40(3):141–142, 1991.
2. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
3. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-Tiga: Time for playing games! In *CAV*, volume 4590 of *LNCS*. Springer, 2007.
5. G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *SFM*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.
6. T. Bourke. *Modelling and Programming Embedded Controllers with Timed Automata and Synchronous Languages*. PhD thesis, University of New South Wales, Sydney, 2009.
7. T. Bourke and A. Sowmya. Automatically transforming and relating Uppaal models of embedded systems. In *EMSOFT*, pages 59–68, 2008.
8. F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, 2005.
9. A. Chakabarti, L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *EMSOFT*, LNCS. Springer, 2003.
10. A. David, K. Larsen, A. Legay, U. Nyman, and A. Wąsowski. ECDAR: An environment for compositional design and analysis of real time systems. In *ATVA*, 2010. Accepted.
11. A. David, K. Larsen, A. Legay, U. Nyman, and A. Wąsowski. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC'10*, pages 91–100. ACM, 2010.
12. L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *FroCos*, volume 3717 of *LNCS*, pages 81–105. Springer, 2005.
13. L. de Alfaro and T. A. Henzinger. Interface automata. In *FSE*, pages 109–120, Vienna, Austria, Sept. 2001. ACM Press.
14. L. de Alfaro and T. A. Henzinger. Interface-based design. In *Marktoberdorf Summer School*. Kluwer Academic Publishers, 2004.
15. L. De Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic Algorithms for Infinite-State Games. In *CONCUR*, volume 2154 of *LNCS*, pages 536–550. Springer, 2001.
16. L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Timed interfaces. In *EMSOFT*, volume 2491 of *LNCS*, pages 108–122. Springer, 2002.
17. http://www.cs.aau.dk/~adavid/ecdar/.
18. H. E. Jensen, K. G. Larsen, and A. Skou. Scaling up Uppaal: Automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT00*, pages 19–30, 2000.
19. D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager. Timed I/O Automata: A mathematical framework for modeling and analyzing real-time systems. In *RTSS*, pages 166–177. IEEE Computer Society, 2003.
20. K. G. Larsen. Modal specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.
21. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.
22. Sharp Corp. GP2D02: Compact, high sensitive distance measuring sensor, 1997.
23. M. I. Stoelinga. *Alea Jacta est: Verification of probabilistic, real-time and parametric systems*. PhD thesis, Katholieke Universiteit Nijmegen, 2002.