Real-time specifications

David, Alexandre; Larsen, Kim Guldstrand; Legay, Axel; Nyman, Ulrik Mathias; Traonouez, Louis-Marie; Wasowski, Andrzej

Published in:
International Journal on Software Tools for Technology Transfer

DOI (link to publication from Publisher):
10.1007/s10009-013-0286-x

Publication date:
2015

Document Version
Early version, also known as pre-print

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Real-Time Specifications ⋆

Alexandre David[1] and Kim. G. Larsen[1] and Axel Legay[2] and Ulrik Nyman[1] and Louis-Marie Traonouez[2] and Andrzej Wąsowski[3]

[1] Computer Science, Aalborg University, Denmark, e-mail: adavid@cs.aau.dk, kgl@cs.aau.dk, ulrik@cs.aau.dk
[2] INRIA/IRISA, Rennes Cedex, France, e-mail: axel.legay@irisa.fr, louis-marie.traonouez@irisa.fr
[3] IT University of Copenhagen, Denmark, e-mail: wasowski@itu.dk

Received: date / Accepted: date

Abstract A specification theory combines notions of specifications and implementations with a satisfaction relation, a refinement relation, and a set of operators supporting stepwise design. We develop a specification framework for real-time systems using Timed I/O Automata as the specification formalism, with the semantics expressed in terms of Timed I/O Transition Systems. We provide constructs for refinement, consistency checking, logical and structural composition, and quotient of specifications – all indispensable ingredients of a compositional design methodology.

The theory is implemented in the new tool ECDAR. We present symbolic versions of the algorithms used in ECDAR, and demonstrate the use of the tool using a small case study in compositional verification.

Key words: Real-time systems, Stepwise-Refinement, Compositional Verification

1 Introduction

Many modern systems are big and complex assemblies of numerous components. The components are often designed by independent teams, working under a common agreement on what the interface of each component should be. Consequently, *compositional reasoning* [41], the mathematical foundations of reasoning about interfaces, is an active research area. It supports inferring properties of the global implementation from the components, or advisedly designing and reusing components.

In a logical interpretation, interfaces are specifications, while components that implement an interface are understood as models/implementations. Specification theories may support various features including (1) *refinement*, which allows us to compare specifications as well as to replace a specification by another one in a larger design, (2) *logical conjunction*, expressing the intersection of the set of requirements expressed by two or more specifications, (3) *structural composition*, which allows us to combine specifications, and (4) a *quotient operator* that is dual to structural composition. We shall see that quotient is useful to perform incremental design and to reason about assumptions and guarantees. Also, the operations have to be related by compositional reasoning theorems, guaranteeing both incremental design and independent implementability [32].

Building good specification theories is the subject of intensive studies [20,31]. One successfully direction is the theory of interface automata [31,32,45,52]. In this framework, an interface is represented by an input/output automaton [50], *i.e.* an automaton whose transitions are typed with *input* and *output*. The semantics of such an automaton is given by a two-player game: the *input* player represents the environment, and the *output* player represents the component itself. Contrary to the input/output model proposed by Lynch [50], this semantic offers an optimistic treatment of composition: two interfaces can be composed if there exists at least one environment in which they can interact together in a safe way. In [34], a timed extension of the theory of interface automata has been introduced, motivated by the fact that time can be a crucial parameter in practice, for example in embedded systems. While [34] focuses mostly on structural composition, in this paper we go one step further and build a game-based specification theory for timed systems that embeds the four features listed above.

⋆ This paper is an extended version of the work previously presented in [24,23,26]. The main additions are (1) a unified presentation, (2) a deeper link between the theory and the tool, (3) proofs of theorems, and (4) the description of case studies.

We represent specifications by timed input/output automata [42], i.e., timed automata whose sets of discrete transitions are split into input and output transitions (see Section 4). Contrary to [34] and [42], we distinguish between implementations and specifications by adding conditions on the models. This is done by assuming that the former have fixed timing behaviour and they can always advance either by producing an output or delaying. We also provide a game-based methodology to decide whether a specification is consistent, i.e. whether it has at least one implementation. The latter reduces to deciding existence of a strategy that despite the behaviour of the environment will avoid states that cannot possibly satisfy the implementation requirements.

Our theory is equipped with a refinement relation (see Section 5). Roughly speaking, a specification S_1 *refines* a specification S_2 iff it is possible to replace S_2 with S_1 in every environment and obtain an equivalent system that satisfies the same specifications. In the input/output setting, checking refinement reduces to deciding an alternating timed simulation between the two specifications [31]. In our timed extension, checking such simulation can be done with a slight modification of the theory proposed in [15]. As implementations are specifications, refinement coincides with the satisfaction relation. Our refinement operator has the *model inclusion property*, i.e., S_1 refines S_2 iff the set of implementations satisfied by S_1 is included in the set of implementations satisfied by S_2. We also propose a *logical conjunction* operator between specifications (see Section 6). Given two specifications, the operator will compute a specification whose implementations are satisfied by both operands. The operation may introduce error states that do not satisfy the implementation requirement. Those states are pruned by synthesizing a strategy for the component to avoid reaching them. We also show that conjunction coincides with shared refinement, i.e., it corresponds to the greatest specification that refines both S_1 and S_2.

Following [34], specifications interact by synchronizing on inputs and outputs. However, like in [42,50], we restrict ourselves to input-enabled systems. This makes it impossible to reach an immediate *deadlock state*, where a component proposes an output that cannot be captured by the other component. Here, in checking for compatibility of the composition of specifications, one tries to synthesize a strategy for the inputs to avoid the error states, i.e., an environment in which the components can be used together in a safe way. Our composition operator is associative and the refinement is a precongruence with respect to it (see Section 7). We propose a quotient operator dual to composition (see Section 8). Intuitively, given a global specification T of a composite system as well as the specification of an already realized component S, the *quotient* will return the most liberal specification X for the missing component, i.e. X is the largest specification such that S in parallel with X refines T.

Our methodology has been implemented in a new tool ECDAR that is an extension of UPPAAL-TIGA [9] (see Section 9). It builds on timed input/output automata, a symbolic representation for timed input/output transition systems. We show that conjunction, composition, and quotienting can be reduced to simple product constructions allowing for both consistency and compatibility checking to be solved using the zone-based algorithms for synthesizing winning strategies in timed games [51, 17]. So while our theory is clearly new, our reduction allows us to exploit well-established algorithms and implementations which makes it robust. Finally, refinement between specifications is checked using a variant of the recent efficient game-based algorithm of [15]. The potential of our tool is illustrated on two case studies, each of them showing the utility of the various features of our theory (see Sections 10 and 11).

2 Introductory Example

We will now give a rough overview of the theory using an example. Consider a vending machine that can serve tea or coffee. Its specification is shown in Fig. 1(a). We use the syntax of timed I/O automata [42]. The dashed edges represent outputs and the solid ones correspond to inputs. In the example, `tea!` is an output and `coin?` is an input. The machine waits for coins and serves either tea or coffee with different timing constraints. It can also serve free tea after two time units. A possible implementation of this machine is given in Fig. 1(b).

Our models share the following characteristics:

- Both specifications and implementations are deterministic. This assumption reflects our experience of working with engineers, who prefer to create deterministic specifications. It also allows to create a theory with good properties for compositional reasoning.
- Output transitions of the implementation Implementation must arrive at a fixed moment in time and cannot be delayed. We say that an implementation is *output-urgent*. Specifications are allowed to be imprecise about timing of outputs, while implementations have fixed timing. Intuitively, this means that not only the choice of action, but also the timing (of outputs) is deterministic. We do not restrict the timing of inputs as the environment may well be not predictable.
- In Implementation, we can observe that each time the output tea! from Idle to Idle is taken, Clock y is reset. Without this reset, the time would be stopped and the execution would be stuck in the location Idle. A desirable property is that either a component can delay or it must be able to produce some output. This property, called *independent progress*, guarantees that the progress of time can happen without relying on the environment.

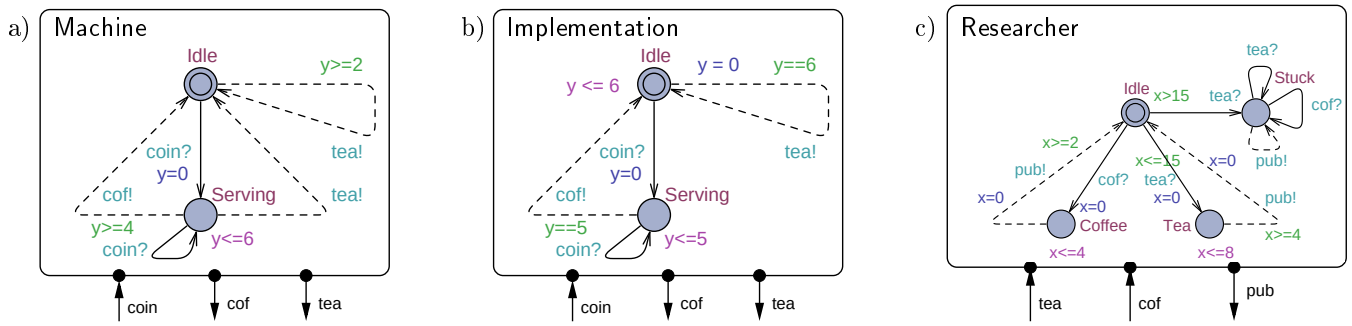

Figure 1: a) Specification of a coffee and tea Machine, b) an implementation that refines the specification and c) a Researcher that uses the Machine. Initial locations are double circled. Transition guards are written in green and clock resets in blue, while location invariants are in purple.

– Both specifications and implementations are assumed to be input-enabled. This is a natural requirement that a component cannot prevent the environment from sending an input. Instead we should be able to describe the failure of the system, when an unexpected input arrives. This assumption is made in many specification theories [49,38,56,61,53].

Implementations relate to specifications through refinement. More precisely, our implementation model Implementation *refines* our specification Machine in the sense that whenever Implementation wants to produce an output, that output is allowed by Machine, and Implementation accepts all the inputs specified by Machine. Then an implementation is reusable in any environment which accepts the specification. Also an implementation will not produce more interactions than what the specification allows in such an environment. We will see later that checking refinement reduces to a two-player game where the attacker plays delays and outputs on Implementation, and inputs on Machine, while the defender responds with outputs and delays on Machine, and inputs from Implementation.

More generally, the refinement can be used to compare specifications. Thanks to the assumptions of determinism and input-enabledness, our refinement coincides with implementation set inclusion, that is Specification A_S refines Specification A_T if and only if the set of implementations of A_S is included in the set of implementations of A_T.

Consider now the specification of UniSpec in Fig. 2. A good university produces patents as a result of receiving grants. Observe the timing constraints that constrain how often the university should produce patents. Our objective is to refine this specification by another one that is more precise regarding the behavior of the researchers and administration staff of the university. We consider researchers who will publish, if provided with tea and coffee, an administration that will turn grants into coins (to fund tea and coffee) while turning publications into patents, and a coffee machine that accepts coins and produces hot beverages for the researchers. In order to reason about each component individually, we will split

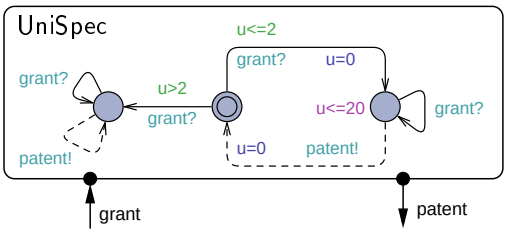

Figure 2: Specification of the university component (UniSpec).

the university specification into multiple specifications that we will combine using composition operators. The resulting specification shall then be checked against the original one using refinement.

The specifications for the coffee machine and the researcher are given in figures 1(a) and 1(c), respectively. We assume that researchers publish more efficiently if drinking coffee than when drinking tea. Furthermore, researchers dislike tea, so if tea is served after a long period of waiting (15 units of time) the subsequent behaviour is undefined—supposedly due to irritation. Publications are produced with the output `pub!`.

The case of the administration is somewhat more complicated. Indeed, administration should not only turn grants into coins, but also turn publications into patents—a conjunction of two requirements. We will model each requirement individually and then compute their conjunction, i.e, the specification that represents the set of their common implementations: Administration is the conjunction of HalfAdm1 and HalfAdm2, both presented in Fig. 3. Observe that both specifications are input enabled and allow patents and coins as outputs. Given grants (`grant?`), resp. publications (`pub?`), coins are produced within 2 time units (with `coin!`), resp. patents (with `patent!`). In general, conjunction can introduce bad behaviors in specifications, i.e, behaviors that cannot be implemented because they do not respect properties such as independent progress. In our theory such behaviors will be pruned using a game-based technique.

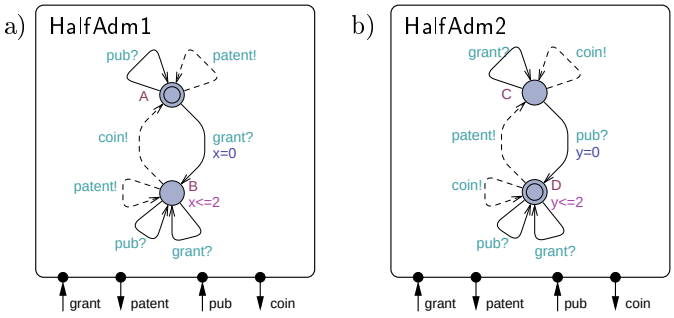

Figure 3: Two conjuncts that together model the Administration component.

We are now ready to compose our specifications in order to derive a refinement of the university model. Fig. 4 gives the overview of this refinement check. We put in parallel the components for the researcher, the coffee machine, and the administration. Our verification engine then checks if this composition refines the specification of our university. The verification is done in a compositional manner in the sense that every component is explored locally, bad behaviour is eliminated (pruned), and combined with the appropriate operator, shown in the figure.

Slightly surprisingly, the refinement check of Fig. 4 fails. It turns out that since the machine allows the researchers to get free tea, they can publish for free, which can give patents for free—a scenario that has not been anticipated in the specification.

3 Related Work

The objective of this section is mainly to survey a state-of-the art for interface theory, not to make an exhaustive list of all existing timed specification theories.

It has been argued [31,27,32] that *games* constitute a natural model for interface theories: each component is represented by an automaton whose transitions are typed with *input* and *output* modalities. The semantics of such an automaton is given by a two-player game: the *input* player represents the environment, and the *output* player represents the component. Contrary to the input/output model proposed by Lynch and Tuttle [50], this semantic offers (among many other advantages) an optimistic treatment of composition: two interfaces can be composed if there exists at least one environment in which they can interact together in a safe way. Game-based interfaces were first developed for untimed systems [32,28] and implemented in tools such as TICC [2] and CHIC [21] for both synchronous and asynchronous models. The first dense time extension of the theory of interface automata has been developed in [34], motivated by the fact that real time is a crucial parameter in some systems. The theory, which extends timed input/output automata [42], was later implemented in TICC, but using discretized real time only [29]. The idea is similar to the untimed case: components are modeled using timed input/output automata (TIOAs) with a timed game semantics [17]. The theory of [34] has never been completed, in the sense that it lacks support for conjunction and refinement (in contrast to the one presented here). The usefulness of such theories for compositional design of real time systems is thus limited. While tooling is not the focus of this paper, let us mention that, elsewhere [14], we show how the Ecdar tool and our timed interface theory can be used to solve problems that are beyond the scope of classical Uppaal timed input/automata extensions [13,11].

In [45] Larsen proposes *modal automata*, which are deterministic automata equipped with transitions of the following two types: *may* and *must*. The components that implement such interfaces are simple labeled transition systems. Roughly, a must transition is available in every component that implements the modal specification, while a may transition need not be. Recently [12] a timed extension of *modal* automata was proposed. This series of works, which generalizes an early attempt [19], embeds all the operations presented in the present paper. However, modalities are orthogonal to inputs and outputs, and it is well-known [47] that, contrary to the game-semantic approach, they cannot be used to distinguish between the behaviors of the component and those of the environment.

Among other modeling languages for specification, one find those that use logical representations such as Timed Computational Tree Logic (TCTL), Metric Temporal Logic (MTL), or duration. While such logics are generally convenient to reason on individual requirements [54], they are generally not suited for operations such as structural composition and quotient. To the best of our knowledge, the expressiveness relation between logical formalism and timed I/O automata or timed modal specifications remains unknown. There are also timed extensions of languages such as CSP. A comparison between CSP (and related process algebra languages) and interface theories can be found in [8].

Finally, let us add that numerous authors have studied interface theories and component based design. Among them, one finds a series of very practical works that do not study quotient and conjunction, but rather focus on richer composition operations and specific models of computation for interconnection and software design [1, 36,37]. Another example is the series of more recent papers that focus on composition and performance analysis or scheduling for embedded systems [40]. While our theory is certainly more general, it would be of interest to learn from those models and the case studies they handle in order to extend our composition operation.

There are of course other tools and theories for timed systems. As an example, another tool supporting refinement is PAT [57,58]. Unlike Ecdar, it builds on CSP

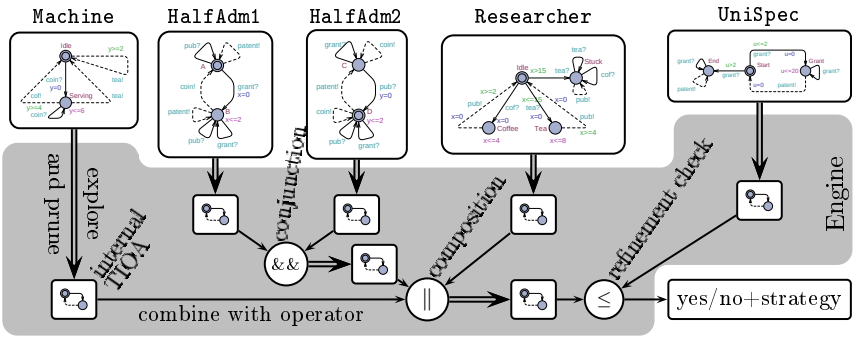

Figure 4: Illustration of the steps performed in a concrete refinement check. The gray box represents the part carried out internally by the verification engine.

with a failure, divergence, and refusal semantics, which makes a direct comparison difficult. However, the CSP theory does not support quotienting nor simple conjunction of specifications. And thus, in contrast to Ecdar, PAT does not support assume/guarantee reasoning about systems. This related work survey only the position of our work in the interface theory setting.

4 Specifications and Implementations

We use four classes of objects in our theory—specifications, and models (implementations) together with their respective behavioral semantics as transition systems. Two kinds of relations are used between the four classes: operational semantics and satisfaction. Fig. 5 shows an overview of the four classes of objects and relations between them.

We distinguish *specifications* and *models*. In the left part of Fig. 5, a specification A and a model X can be related through a satisfaction relation $\models$, relating models and specifications. The left half of Fig. 5, shows syntactic objects (specifications and implementations), while the right half shows the semantic objects (*specification semantics* and *implementation semantics*). Horizontal arrows point from syntactic objects to their semantics. Vertical arrows point from specifications downwards to their models (both in the syntactic and the semantic halves).

Traditionally specifications are logical formulas, and models are witnesses of consistency of these formulas. This is the view that most of the model-checking [22, 7] research takes. In our case, specifications are timed games [51], resembling timed automata [3]. Since these are symbolic finite representations describing continuous state behavior, it is convenient to distinguish another semantic layer, which describes this behavior operationally. Thus we will say that the semantics of a specification A (respectively of an implementation X) is given by a Timed I/O Transition System $[\![S]\!]_{\text{sem}}$ (respectively of a Timed I/O Transition System $[\![X]\!]_{\text{sem}}$). Our transition systems are very similar to those induced

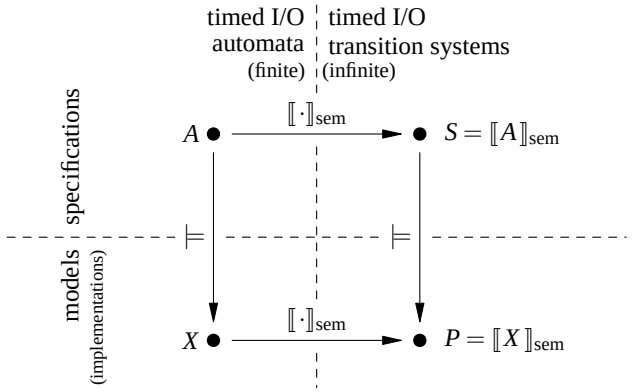

Figure 5: Semantic Layer's in our specification theory

by processes in [63], except that their discrete actions are split into inputs and outputs, like in I/O automata [49]. Unlike in I/O automata we give them a game semantics, not the language semantics.

Throughout the presentation of our specification theory, we continuously switch the mode of discussion between the semantic and syntactic levels. In general, the formal framework is developed for the semantic objects, *Timed I/O Transition Systems* (TIOTSs in short) [39], and enriched with syntactic constructions for *Timed I/O Automata* (TIOAs), which act as a symbolic and finite representation for TIOTSs. However, the theory for TIOTSs does not rely in any way on the TIOAs representation—one can build TIOTSs that cannot be represented by TIOAs, and the theory remains sound for them (although we would not know how to manipulate them symbolically).

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, $\Sigma^S = \Sigma_{\text{i}}^S \oplus \Sigma_{\text{o}}^S$ is a finite set of actions partitioned into inputs (Σ_{i}^S) and outputs (Σ_{o}^S), 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 we write $s \xrightarrow{a}{}^S$ if $\exists s'. s \xrightarrow{a}{}^S s'$, and use $i?$, $o!$ and d to

range over inputs, outputs and $\mathcal{R}_{\geq 0}$ respectively. Transitions that are labelled by actions (inputs or outputs) are called *discrete* transitions, while transitions labelled by real values are called *timed* transitions. In addition any TIOTS satisfies the following:

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

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

[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 an $s' \in St^S$.

We only work with *deterministic* TIOTSs in this paper: for all $a \in \Sigma \cup \mathcal{R}_{\geq 0}$ whenever $s \xrightarrow{a}{}^{S} s'$ and $s \xrightarrow{a}{}^{S} s''$, we have $s' = s''$ (determinism is required not only for timed transitions, but also for discrete transitions). In the rest of the paper, we often drop the adjective 'deterministic'. Of course, this definition of determinism does not prevent from issuing several actions from the same state, the only restriction is that one given action can only take the system to a deterministic location.

For a TIOTS S and a set of states X, we write:

$$\mathsf{pred}_a^S(X) = \left\{ s \in St^S \mid \exists s' \in X.\ s \xrightarrow{a}{}^{S} s' \right\} \tag{1}$$

for the set of all a-predecessors of states in X. We write $\mathsf{ipred}^S(X)$ for the set of all input predecessors, and $\mathsf{opred}^S(X)$ for all the output predecessors of X:

$$\mathsf{ipred}^S(X) = \textstyle\bigcup_{a \in \Sigma_{\mathrm{i}}^S} \mathsf{pred}_a^S(X) \tag{2}$$

$$\mathsf{opred}^S(X) = \textstyle\bigcup_{a \in \Sigma_{\mathrm{o}}^S} \mathsf{pred}_a^S(X)\ . \tag{3}$$

Also $\mathsf{post}^S_{[0,d_0]}(s)$ is the set of all time successors of a state s that can be reached by delays smaller or equal to d_0:

$$\mathsf{post}^S_{[0,d_0]}(s) = \left\{ s' \in St^S \mid \exists\, d \in [0, d_0].\ s \xrightarrow{d}{}^{S} s' \right\} \tag{4}$$

Following [51] we will later use these operators to find strategies for safety and reachability objectives imposed on TIOTSs.

We shall now introduce a finite syntactic symbolic representation for TIOTSs in terms of Timed I/O Automata (TIOAs). Let *Clk* be a finite set of *clocks*. A *clock valuation* over *Clk* is a mapping $u \in [Clk \mapsto \mathcal{R}_{\geq 0}]$. Given $d \in \mathcal{R}_{\geq 0}$, we write $u + d$ to denote a valuation such that for any clock r 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 returns 0 for all clocks in c. 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 $\prec$ is a relational operator and $n \in \mathbb{N}$. We write $\mathcal{B}(Clk)$ for the set of guards over *Clk* using operators in the set op, and $\mathcal{U}(Clk)$ for the subset of upper bound guards using only the operators $\{<, \leq\}$. We also write $\mathscr{P}(X)$ 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, $Act = Act_{\mathrm{i}} \oplus Act_{\mathrm{o}}$ is a finite set of actions, partitioned into inputs and outputs respectively, and $Inv : Loc \mapsto \mathcal{U}(Clk)$ is a set of location invariants.

If $(q, a, \varphi, c, q') \in E$ is an edge, then q is an initial location, a is an action label, 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 a target location. We denote $NextInv(q') = Inv(q') \vee \left(\bigvee_{r \in c} \{r \geq 0\} \right)$ the invariant of the next location that restrict the guard of the edge. Examples of TIOAs have been shown in the introduction.

We define the semantic of a TIOA $A = (Loc, q_0, Clk, E, Act, Inv)$ to be a TIOTS $[\![A]\!]_{\mathrm{sem}} = (Loc \times (Clk \mapsto \mathcal{R}_{\geq 0}), (q_0, \mathbf{0}), Act, \rightarrow)$, where $\mathbf{0}$ is a constant function mapping all clocks to zero, and $\rightarrow$ is the largest transition relation generated by the following rules:

$$\frac{(q, a, \varphi, c, q') \in E \quad u \in [Clk \mapsto \mathcal{R}_{\geq 0}] \quad u \models \varphi \wedge \quad u[r \mapsto 0]_{r \in c} \models Inv(q')}{(q, u) \xrightarrow{a} (q', u[r \mapsto 0]_{r \in c})}$$

$$\frac{q \in Loc \quad u \in [Clk \mapsto \mathcal{R}_{\geq 0}] \quad d \in \mathcal{R}_{\geq 0} \quad u + d \models Inv(q)}{(q, u) \xrightarrow{d} (q, u + d)}$$

The TIOTSs induced by TIOAs, according to the above rules, satisfy the axioms of Definition 1: time determinism, time reflexivity, time additivity. Moreover, in order to guarantee determinism of $[\![A]\!]_{\mathrm{sem}}$, the TIOA A has to be deterministic: for each action–location pair only one transition can be enabled at the same time.

This can be checked algorithmically with a standard check for disjointness of guards of transitions with the same action. For each location q and each action $a \in Act$, check whether all its guards are mutually exclusive. Formally, let $G_{q,a}$ be the set of strengthened guards of all a transitions leaving q:

$$G_{q,a} = \{\varphi \wedge NextInv(q') \mid \text{whenever } (q, a, \varphi, c, q') \in E\} \tag{5}$$

To guarantee determinism check for each pair $\psi_1, \psi_2 \in G_{q,a}$ whether the conjunction $Inv(q) \wedge \psi_1 \wedge \psi_2$ is inconsistent, and do that for all locations.

We assume that all TIOAs below are deterministic.

4.1 Specifications

We will now introduce our notions of specifications and implementations.

Definition 3 (Specification). A TIOTS $P = (St^P, p_0, \Sigma^P, \rightarrow^P)$ is a *specification semantics* if each state $s \in St^P$ is *input-enabled*: for each input $i? \in \Sigma_{\mathrm{i}}^P$ there exists a state $s' \in St^P$ such that $s \xrightarrow{i?}{}^{P} s'$.

A TIOA A is a *specification* iff its semantics $[\![A]\!]_{\text{sem}}$ is *input-enabled*.

The assumption of input-enabledness, also seen in many specification theories [49,38,56,61,53], reflects our belief that an input cannot be prevented from being sent to a system, but it might be unpredictable how the system behaves after receiving it. A standard way of modeling a disallowed input in such a setting is to redirect it to a special universal state, where all actions are enabled—the behaviour of the system becomes unpredictable after reaching this state.

Input-enabledness encourages explicit modeling of this unpredictability, and compositional reasoning about it; for example, it allows asking if an unpredictable behaviour of one component induces unpredictability of the entire system.

In practice, tools should not require the users to specify input-enabled automata, as this quickly becomes tedious. There are however good strategies for making automata input-enabled. First, absent inputs can be interpreted as ignored inputs, corresponding to location loops in the automaton that can be added automatically. Second, absent inputs can be interpreted as unavailable ("blocking") inputs, which are modeled by adding implicit transitions to a designated error location (for example a universal location as suggested above). Later, in Section 7, we will call such a state *strictly undesirable* and give a rationale for this name.

In order to check that a TIOA A induces an input-enabled TIOTS $[\![A]\!]_{\text{sem}}$, decide for each location $q \in Loc^A$ and each input action $i? \in Act$ if a disjunction of guards of outgoing transitions labelled by $i?$ is entailed by $Inv(q)$. Formally, if $G_{q,i?}$ is the set of strengthened guards (see (5)) of all $i?$–transitions leaving q, then in order to check if $i?$ is always enabled in location q, check

$$Inv(q) \quad \text{entails} \bigvee_{\psi \in G_{g,i?}} \psi \tag{6}$$

To check if the entire specification automaton is input-enabled just repeat the check for all location–input pairs.

4.2 Implementations

The role of specifications in a specification theory is to abstract, or underspecify, sets of possible implementations. We will assume that implementations of timed systems have fixed timing behaviour (outputs occur at predictable times) and systems can always advance either by producing an output or delaying. This is formalized using axioms of *output-urgency* and *independent-progress* below:

Definition 4 (Implementation). A TIOTS $P = (St^P, p_0, \Sigma^P, \rightarrow^P)$ is an *implementation semantics* if it is a specification semantics that fulfills the output urgency and independent progress conditions, so if for each state $p \in St^P$ we respectively have:

[output urgency] $\forall p', p'' \in St^P$ if $p \xrightarrow{o!}{}^P p'$ and $p \xrightarrow{d}{}^P p''$ then $d = 0$ (and thus, due to determinism $p = p''$)

[independent progress] either $(\forall d \geq 0.\, p \xrightarrow{d}{}^P)$ or $\exists\, d \in \mathcal{R}_{\geq 0}.\, \exists\, o! \in \Sigma^P_{\text{o}}.\, p \xrightarrow{d} p'$ and $p' \xrightarrow{o!}{}^P$.

A TIOA A is an *implementation* iff A is a specification and its semantics, $[\![A]\!]_{\text{sem}}$, fulfills independent progress and output urgency.

Independent progress is one of the central properties in our theory: it states that an implementation cannot ever get stuck in a state where it is up to the environment to induce the progress of time. So in every state there is either an output transition (which is controlled by the implementation) or an ability to delay until an output is possible. Otherwise a state can delay indefinitely. An implementation cannot wait for an input from the environment without letting time pass.

Remark 1. Our notion of implementation remains at the theory level. Generating executable code and taking robustness into account is not the topic of this paper. However, one could exploit existing works [5] to generate robust C code from a given timed automaton.

In Section 9 we describe how to check for independent progress and other important properties of specifications.

4.3 Specifications as Timed Games

Specifications are interpreted as two-player real-time games between the *output player* (the component) and the *input player* (the environment). The input player plays with actions in Act_{i} and the output player plays with actions in Act_{o}. A strategy for a player is a function that defines his move at any state (either delaying or playing a controllable action). As we will explain in the following sections, strategies for output (respectively input) can be interpreted as implementations (respectively compatible environments).

A strategy is called *memoryless* if the next move depends solely on the current state. We only consider memoryless strategies, as these suffice for safety games [30]. For simplicity, we only define strategies for the output player (i.e. output is the verifier). Definitions for the input player are obtained symmetrically.

Definition 5. A memoryless *strategy* f_o for the output player on the TIOA A is a partial function $St^{[\![A]\!]_{\text{sem}}} \mapsto Act_{\text{o}} \cup \{\mathsf{delay}\}$, such that

- If $f_o(s) \in Act_{\text{o}}$ then $\exists s'.s \xrightarrow{f_o(s)}{}^S s'$.
- If $f_o(s) = \mathsf{delay}$ then $\exists s''.s \xrightarrow{d}{}^S s''$ for some $d > 0$, and $f_o(s'') = \mathsf{delay}$.

The game proceeds as a concurrent game between the two players. Then, by applying a strategy f_o the output player restricts the set of reachable states from the semantics. This defines the *outcome* of the strategy, such that for a state $s \in St^{[\![A]\!]_{\text{sem}}}$, $\mathsf{Outcome}(s, f_o)$ is the set of states defined inductively by:

- $s \in \mathsf{Outcome}(s, f_o)$,
- if $s' \in \mathsf{Outcome}(s, f_o)$ and $s' \xrightarrow{a} s''$, then $s'' \in \mathsf{Outcome}(s, f_o)$ if one the following conditions holds:
  1. $a \in Act_i$,
  2. $a \in Act_o$ and $f_o(s') = a$,
  3. $a \in \mathcal{R}_{\geq 0}$ and $\forall d \in [0, a[\,. \exists s'''.\, s' \xrightarrow{d} s'''$ and $f_o(s''') = \mathsf{delay}$.

In a safety game, the winning condition is to avoid a set Bad of "bad" states. A strategy f_o is a *winning strategy* from state s if and only if $\mathsf{Outcome}(s, f_o) \cap \mathsf{Bad} = \emptyset$. A state s is winning if there exists a winning strategy from s, and the game is winning if and only if the initial state is winning. Solving this game is decidable [51,17,24].

5 Satisfaction, Refinement and Consistency

A notion of *refinement* allows to compare two specifications as well as to relate an implementation to a specification. Refinement should satisfy the following *substitutability* condition. If P refines Q, then it should be possible to replace Q with P in every environment and obtain an equivalent system.

We study these kind of properties in later sections. It is well known from the literature [31,32,15] that in order to give these kind of guarantees a refinement should have the flavour of *alternating (timed) simulation* [4].

Definition 6 (Refinement $\leq$). A TIOTS $S = (St^S, s_0, \Sigma, \rightarrow^S)$ refines a TIOTS $T = (St^T, t_0, \Sigma, \rightarrow^T)$, written $S \leq T$, iff there exists a binary relation $R \subseteq St^S \times St^T$ containing (s_0, t_0) such that for each pair of states $(s, t) \in R$ we have:

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

A specification automaton A_1 refines another specification automaton A_2, written $A_1 \leq A_2$, iff $[\![A_1]\!]_{\text{sem}} \leq [\![A_2]\!]_{\text{sem}}$.

It is easy to see that the refinement is reflexive and transitive, so it is a preorder on the set of all specifications (and, of course, also on the set of all specification semantics). Refinement can be checked for specification automata by reducing the problem to a specific refinement game, and using a symbolic representation to reason about it. We discuss details of this process in Section 9.

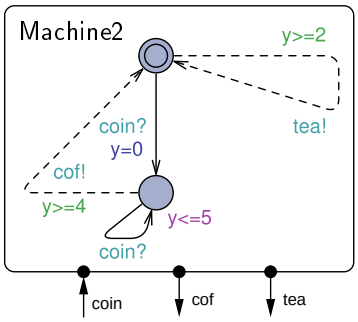

Figure 6: A coffee machine specification that refines the coffee machine in Fig. 1.

Fig. 6 shows a coffee machine that is a refinement of the one in Fig. 1. It has been refined in two ways: one output transition has been completely dropped and one state invariant has been tightened.

Since our implementations are a subclass of specifications, we simply use *refinement* as an implementation relation:

Definition 7 (Satisfaction). An implementation semantics TIOTS P *satisfies* a specification semantics S, written $P \models S$, iff $P \leq S$. An implementation I satisfies a specification A iff $[\![I]\!]_{\text{sem}} \models [\![A]\!]_{\text{sem}}$. We write $[\![A]\!]_{\text{mod}}$ for all semantic models of A, so $[\![A]\!]_{\text{mod}} = \{P \mid P \text{ is a TIOTS and } P \models [\![A]\!]_{\text{sem}}\}$.

From a logical perspective, specifications are like formulae, and implementations are their models. This analogy leads us to a classical notion of consistency, as existence of models.

Definition 8 (Consistency). A specification semantics TIOTS S is consistent if there exists an input-enabled TIOTS P such that $P \models S$, and P is an implementation semantics. A specification A is consistent if its specification semantics, $[\![A]\!]_{\text{sem}}$, is consistent.

All specifications shown until now are consistent. An example of an inconsistent specification can be found in Fig. 7: notice that the invariant in the second state ($x \leq 4$) is stronger than the guard ($x \geq 5$) on the cof! edge; therefore this state does not fulfill the independent progress condition, and it cannot be implemented.

We also define a soundly stricter, more syntactic, notion of consistency, which requires that all states are consistent:

Definition 9 (Local Consistency). A state s of a specification semantics S is *locally consistent* if it fulfills independent progress. S is locally consistent iff ev-

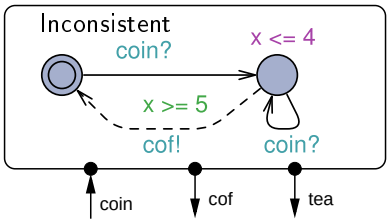

Figure 7: An inconsistent specification.

ery state $s \in St^S$ is locally consistent. A specification A is locally consistent if $[\![A]\!]_{\text{sem}}$ is locally consistent.

Lemma 1. *Every locally consistent specification semantics S is consistent in the sense of Def. 8.*

Proof (Lemma 1). Let us begin with defining an auxiliary function δ which chooses a delay and an output for every locally consistent state s:

$$\delta_s = \begin{cases} d & \text{for some } d \text{ such that } s \xrightarrow{d}{}^S s' \\ & \text{and } \exists o!.\, s' \xrightarrow{o!}{}^S \\ +\infty & \text{if } \forall d \geq 0.\, s \xrightarrow{d}{}^S \end{cases} \tag{7}$$

Note that δ is a function, so it always gives a unique value of a delay for any state s, thus in the first case we mean that an arbitrary fixed value is chosen out of uncountably many possible values. It is immaterial for the proof which of the many values is chosen. It is important however that δ is time additive in the following sense: if $s \xrightarrow{d} s'$ and $d \leq \delta_s$ then $\delta_{s'} + d = \delta_s$. It is always possible to choose such a function δ due to time additivity of $\rightarrow^S$, and local consistency of S.

We want to synthesize a TIOTS $P = (St^P, p_{s_0}, \Sigma^P, \rightarrow^P)$, where $St^P = \{p_s \mid s \in St^S\}$, $\Sigma^P = \Sigma^S$ with the same partitioning into inputs and outputs, and $\rightarrow^P$ is the largest transition relation generated by the following rules:

$$\frac{s \xrightarrow{i?}{}^S s' \qquad i? \in \Sigma^S_{\text{i}}}{p_s \xrightarrow{i?}{}^P p_{s'}} \tag{8}$$

$$\frac{s \xrightarrow{o!}{}^S s' \qquad o! \in \Sigma^S_{\text{o}} \qquad \delta_s = 0}{p_s \xrightarrow{o!}{}^P p_{s'}} \tag{9}$$

$$\frac{s \xrightarrow{d}{}^S s' \qquad d \in \mathcal{R}_{\geq 0} \qquad d \leq \delta_s}{p_s \xrightarrow{d}{}^P p_{s'}} \tag{10}$$

Since P only takes a subset of transitions of S, the determinism of S implies determinism of P. The transition relation of P is time-additive due to time additivity of $\rightarrow^{[\![A]\!]_{\text{sem}}}$ and of δ. It is also time-reflexive due to the last rule ($0 \leq \delta_s$ for every state s and $\rightarrow^S$ was time reflexive). So P is a TIOTS.

The new transition relation is also input-enabled as it inherits input transitions from A, which was input enabled. The second rule guarantees that outputs are urgent (P only outputs when no further delays are possible). Moreover P observes independent progress. Consider a state p_s. Then if $\delta_s = +\infty$ clearly p_s can delay indefinitely. If δ_s is finite, then by definition of δ and of P, the state p_s can delay and then produce an output. Thus P satisfies conditions of Def. 8.

Now, the following relation $R \subseteq St^P \times St^S$ witnesses $P \models S$:

$$R = \left\{ (p_s, s) \mid p_s \in St^P \text{ and } s \in St^{[\![A]\!]_{\text{sem}}} \right\} \tag{11}$$

This is argued using an unsurprising coinductive argument. Obviously, $(p_{s_0}, s_0) \in R$. Now for any $(p_s, s) \in R$:

- If $s \xrightarrow{i?}{}^S s'$ with $i? \in \Sigma^S_{\text{i}}$, then according to rule 8 $p_s \xrightarrow{i?}{}^P p_{s'}$.
- If $p_s \xrightarrow{o!}{}^P p_{s'}$ with $o! \in \Sigma^S_{\text{o}}$, then according to rule 9 $s \xrightarrow{o!}{}^S s'$.
- If $p_s \xrightarrow{d}{}^P p_{s'}$ with $d \in \mathcal{R}_{\geq 0}$, then according to rule 10 $s \xrightarrow{d}{}^S s'$.

This proves that R is a refinement relation. □

It follows directly that:

Corollary 1. *Every locally consistent specification is consistent (in the sense of Def.8).*

We shall see later (Figure 8) that the implication opposite to the one of Corollary 1 does not hold. To establish local consistency, or independent progress, for a TIOA, it suffices to check for each location if the supremum of all solutions of its invariant exists, whether it satisfies the invariant itself and allows at least one enabled output transition.

Prior specification theories for discrete time [45] and probabilistic [16] systems reveal two main requirements for a definition of implementation. These are the same requirements that are typically imposed on a definition of a model as a special case of a logical formula. First, implementations should be consistent specifications (logically, models correspond to some consistent formulae). Second, implementations should be fully specified (models cannot be refined by non-models), as opposed to proper specifications, which should be *underspecified*. For example, in propositional logics, a model is represented as a complete consistent term. Any implicant of such a term is also a model (in propositional logics, it is actually equivalent to it).

Our definition of implementation satisfies both requirements, and to the best of our knowledge, is the first example of a proper notion of implementation for timed specifications. As the refinement is reflexive we get $P \models P$ for any implementation and thus each implementation is consistent as per Def. 8. Furthermore each implementation cannot be refined anymore by any underspecified specifications:

Lemma 2. *Any locally consistent specification semantics S refining an implementation semantics P is an implementation semantics as per Def. 4.*

Proof (Lemma 2). Observe first that S is already locally consistent, so all states of S warrant independent progress. We only need to argue that they also verify output urgency.

Without loss of generality, assume that $[\![S]\!]_{\text{sem}}$ only contains states that are reachable by (sequences of) discrete or timed transitions.

If S only contains reachable states, every state of S has to be related to some state of P in a relation R witnessing $S \leq P$ (output and delay transitions need to be matched in the refinement; input transitions also need to be matched as P is input enabled and S is deterministic). This can be argued for using a standard, though slightly lengthy argument, by formalizing reachable states as a fixpoint of a monotonic operator.

Now, that we know that every state of S is related to some state of P consider an arbitrary $s \in St^S$ and let $p \in St^P$ be such that $(s,p) \in R$. Then if $s \xrightarrow{o!}{}^S s'$ for some state $s' \in St^S$ and an output $o! \in \Sigma_{\text{o}}^S$, it must be that also $p \xrightarrow{o!} p'$ for some state $p' \in St^P$ (and $(s',p') \in R$). But since P is an implementation, its outputs must be urgent, so $p \not\xrightarrow{d}{}^P$ for all $d > 0$, and consequently $s \not\xrightarrow{d}{}^S$ for all $s > 0$. We have shown that all states of S have urgent outputs (if any) and thus S is an implementation. □

Corollary 2. *Any locally consistent specification S refining an implementation P is an implementation itself.*

We conclude the section with the first major theorem. Observe that every preorder $\preceq$ is intrinsically complete in the following sense: $S \preceq T$ iff for every smaller element $P \preceq S$ also $P \preceq T$. This means that a refinement of two specifications coincides with inclusion of sets of all the specifications refining each of them:

$$S \leq T \quad \text{iff} \quad \{P \mid P \leq S\} \subseteq \{P \mid P \leq T\} \tag{12}$$

However, since out of all specifications only the implementations correspond to real world objects, another completeness question is more relevant: does the refinement coincide with the inclusion of implementation sets? This property, which does not hold for preorders in general, turns out to hold for our refinement:

Theorem 1 (Refinement Is Thorough). *For any two locally consistent specifications A, B we have that*

$$A \leq B \text{ iff } [\![A]\!]_{\text{mod}} \subseteq [\![B]\!]_{\text{mod}} \tag{13}$$

We split the proof of Theorem 1 into two lemmas.

Lemma 3 (Soundness). *For all locally consistent specification semantics S and T, if $S \leq T$ then for any implementation semantics P, $P \models S$ implies $P \models T$.*

Proof (Lemma 3). This lemma is a special case of the transitivity of the refinement relation. Consider an implementation semantics P of S. Then $P \leq S$ and $S \leq T$, implies $P \leq T$, which proves that $P \models T$. □

Lemma 4 (Completeness). *For all locally consistent specification semantics S and T, if for any implementation semantics P, $P \models S$ implies $P \models T$, then $S \leq T$.*

In the following we write $p \models s$ for states p and s of TIOTS P (respectively S) meaning that there exists a relation R' witnessing $P \models S$ that contains the pair of states (p,s).

Proof (Lemma 4). Assume that every model of S is a model of T. Consider the relation $R \subseteq St^S \times St^T$:

$$R = \{(s,t) \mid \text{for each implementation TIOA } P \text{ it holds that } (p_0^P \models s \implies p_0^P \models t)\} \ , \tag{14}$$

where p_0^P denotes the initial state of P. We shall argue that R witnesses $S \leq T$. It follows directly from the definition of R and the assumption on model inclusion that $(s_0,t_0) \in R$. Now consider a pair $(s,t) \in R$. There are two cases to be considered:

- For any input $i?$ there exists $t' \in St^T$ such that $t \xrightarrow{i?}{}^T t'$. We need to show existence of a state $s' \in St^S$ such that $s \xrightarrow{i?}{}^S s'$ and $(s',t') \in R$.

  Observe that due to input-enabledness, for the same $i?$, there exists a state $s' \in St^S$ such that $s \xrightarrow{i?}{}_{[\![S]\!]_{\text{sem}}} s'$. We need to show that $(s',t') \in R$. By Theorem 1 we have that there exists an implementation semantics P with initial state p_0^P such that $p_0^P \models s'$ (technically speaking, s may be a non-initial state of S, but then we can consider a version of S with initial state changed to s to apply Theorem 1, concluding existence of the implementation P as above).

  We will now argue that arbitrary implementation semantics (not only P) satisfying the state s' also satisfies t'. So consider an implementation semantics $Q \models S$ and its initial state q_0^Q such that $q_0^Q \models s'$. We show that $q_0^Q \models t'$.

  Create an implementation Q' by merging Q and P above and adding a fresh state $q_0^{Q'}$ with all the same transitions like the initial location of P (so targeting locations of the P-part), except for the transition labeled by i?, which should go to q_0^Q; so: $q_0^{Q'} \xrightarrow{i?}{}^{Q'} q_0^Q$ and otherwise $q_0^{Q'} \xrightarrow{a}{}^{Q'} p$ whenever $p_0^P \xrightarrow{a}{}^P p$ for $a \neq i?$. The transitions for all the other states of Q' are like in P and Q, depending to which of the two implementation semantics the state originally belonged. Now $q_0^{Q'} \models s$ as $p \models s$ and it follows all evolutions of p for $a \neq i?$ and $q \xrightarrow{i?}{}^{Q'} q_0$ and $q_0 \models s'$. By assumption, every implementation semantics of s is also an

implementation semantics of t, so $q_0^{Q'} \models t$ and consequently $q_0 \models t'$ as $q_0^{Q'}$ is deterministic on $i?$.

Summarizing, for any implementation $q_0 \models s'$ we were able to argue that $q_0 \models t'$, thus necessarily $(s', t') \in R$.

– Consider any action a (which is an output or a delay) for which exists s' such that $s \xrightarrow{a}{}^S s'$. Similarly as above, one can construct (and thus postulate existence) of an implementation P containing $p \in St^P$ such that $p \models s$ which has a transition $p \xrightarrow{a}{}^P p'$. Since then also $p \models t$ we have that there exists $t' \in St^T$ such that $t \xrightarrow{a}{}^T t'$. It remains to argue that $(s', t') \in R$. This is done in the same way as with the first case, by considering any model of s', then by extending it deterministically to a model of s, concluding that it is now a model of t and the only a-derivative, which is p', must be a model of t'. Consequently $(s', t') \in R$. □

A complete refinement in the above sense is also sometimes called *thorough* (see e.g. [6]). The restriction of the theorem to locally consistent specifications is not a serious one. As we shall see later (Theorem 2), any consistent specification can be transformed into a locally consistent one preserving the set of implementations.

6 Consistency and Conjunction

6.1 Consistency

We will now study how consistency and refinement interact with time lock errors (violation of independent progress) in specifications. In particular we will give an operational characterization of Def. 8.

An *immediate error* occurs in a state of a specification semantics if the state disallows progress of time and output transitions—such a specification will break if the environment does not send an input. For a specification semantics S we define the set of immediate error states $\mathsf{err}^S \subseteq St^S$ as:

$$\mathsf{err}^S = \left\{ s \;\middle|\; (\exists d.\, s \not\xrightarrow{d}) \text{ and } \forall d \forall o! \forall s'. s \xrightarrow{d} s' \text{ implies } s' \not\xrightarrow{o!} \right\}$$

It follows that no immediate error states can occur in implementations, or in locally consistent specifications.

In general, immediate error states in a specification do not necessarily mean that a specification cannot be implemented. Fig. 8 shows a partially inconsistent specification, a version of the coffee machine that becomes inconsistent if it ever outputs tea. The inconsistency can be possibly avoided by some implementations, which would not implement delay or output transitions leading to it. More precisely an implementation will exist if there is a strategy for the output player in a safety game to avoid err^S.

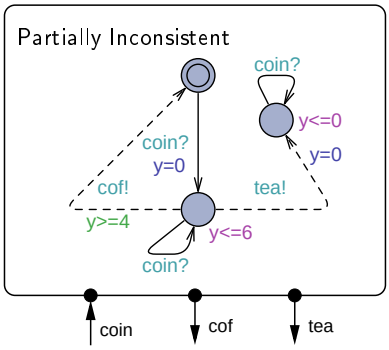

Figure 8: A partially inconsistent specification.

We will solve the safety game, by seeking states which can delay until a safe move, without passing through any unsafe states (or states from which a spoiling move exists). We first define the safe timed predecessor operator [33,51,17], which gives all the states that can safely delay until an output into X while avoiding the set of unsafe states Y:

$$\mathsf{Pred}_{\mathrm{t}}^S(X, Y) = \{ s \in St^S \mid \exists d_0 \in \mathcal{R}_{\geq 0}.\, \exists s' \in X.\, s \xrightarrow{d_0}{}^S s' \text{ and } \mathsf{post}^S_{[0,d_0]}(s) \subseteq \overline{Y} \} \quad (15)$$

Since in our game it is possible to play by delaying indefinitely (not necessarily until an output is possible), we need another operator, $\mathsf{Idle}_{\mathrm{t}}$, that captures states that can delay indefinitely without passing through unsafe states. This operator is analogous to the above one, except that it delays indefinitely. Again, Y denotes the unsafe states:

$$\mathsf{Idle}_{\mathrm{t}}^S(Y) = \{ s \in St^S \mid \forall d \in \mathcal{R}_{\geq 0}.\, \exists s' \in \overline{Y}.\, s \xrightarrow{d} s' \} \quad (16)$$

Now the set of safe states is computed as the greatest fixpoint of the following operator π, which is an adjustment of the standard *controllable predecessors* operator [33,51] that accounts for infinite delay moves:

$$\pi(X) = \overline{\mathsf{err}^S} \cap \left[\mathsf{Idle}_{\mathrm{t}}^S \left(\mathsf{ipred}^S(\overline{X}) \right) \cup \mathsf{Pred}_{\mathrm{t}}^S \left(\mathsf{opred}^S(X), \mathsf{ipred}^S(\overline{X}) \right) \right] \quad (17)$$

The π operator formalizes a two player game, when both players choose a delay, possibly zero, and a move to be made. The move with a shorter delay is executed. If the two delays are equal then the move is nondeterministic, and thus the operator computing the strategy requires that both moves have to be non-losing.

The set of all consistent states cons^S (i.e. the states for which the environment has a winning strategy) is defined as the greatest fixpoint of π: $\mathsf{cons}^S = \pi(\mathsf{cons}^S)$, which is guaranteed to exist by monotonicity of π and completeness of the powerset lattice due to the theorem of Knaster and Tarski [59]. For transition systems

enjoying finite symbolic representations, automata specifications included, the fixpoint computation converges after a finite number of iterations [51,17].

Lemma 5. *A specification semantics* $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ *is consistent iff* $s_0^S \in \mathsf{cons}^S$.

Correctness of the fixpoint characterization of winning strategies for safety games has first been observed in [51]. We have updated the theorem to our setting (which allows idling as a possible move). Below we provide a proof for this extended version.

Proof (Lemma 5). First, assume that $s_0 \in \mathsf{cons}^S$. Show that S is consistent in the sense of Def. 8. In a similar fashion to the proof of Lemma 1 we first postulate existence of a function δ, which chooses a delay and an output for every consistent state s:

$$\delta_s = \begin{cases} d' & \text{if } \exists s', s'' \in \mathsf{cons}^S.\ s \xrightarrow{d'}{}^S s' \text{ and } \exists o!.\ s' \xrightarrow{o!}{}^S s'' \\ +\infty & \textit{otherwise} \end{cases} \tag{18}$$

For each state $s \in \mathsf{cons}^S$ the value of δ_s can be defined, since either $s \in \mathsf{Idle}_\mathrm{t}(\mathsf{ipred}^S(\overline{\mathsf{cons}^S}))$ or $s \in \mathsf{Pred}_\mathrm{t}(\mathsf{opred}^S(\mathsf{cons}^S), \mathsf{ipred}^S(\overline{\mathsf{cons}^S}))$. In the former case it must be able to delay indefinitely through states in cons^S (and thus δ_s postulating the infinite delay is reasonable), in the latter case it can delay until an output predecessor of a state in cons^S, without leaving cons^S during the delay. Note that δ is a function, so it always gives a unique value of a delay for any state s, thus in the first case we mean that an arbitrary fixed value is chosen out of possibly uncountably many values for d'. It is important however that δ is time additive in the following sense: if $s \xrightarrow{d} s'$ and $d \le \delta_s$ then $\delta_{s'} + d = \delta_s$. It is always possible to choose such a function δ due to time additivity of $\rightarrow^S$, and the fact that cons^S is a fixpoint of π.

We show this by constructing an implementation semantics $P = (St^P, p_0, \Sigma^P, \rightarrow^P)$ such that $St^P = \{p_s \mid s \in St^S\}$, $\Sigma^P = \Sigma^S$ with the same partitioning in the inputs and outputs, $p_0 = p_{s_0}$ and the transition relation is the largest relation generated by the following rules:

1. $p_s \xrightarrow{o!}{}^P p_{s'}$ iff $s \xrightarrow{o!}{}^S s'$ and $s' \in \mathsf{cons}^S$ and $\delta_s = 0$
2. $p_s \xrightarrow{i?}{}^P p_{s'}$ iff $s \xrightarrow{i?}{}^S s'$
3. $p_s \xrightarrow{d}{}^P p_{s'}$ iff $s \xrightarrow{d}{}^S s'$ and $d \le \delta_s$

Observe that the construction of P is essentially identical to the one in the proof of Lemma 1 above. It can be argued in almost the same way as in the above proof, that P satisfies the axioms of TIOTSs and is an implementation semantics. Here one has to use the definition of π in order to see that the side condition in the first rule, that is $s' \in \mathsf{cons}^S$, does not introduce a violation of independent progress.

It remains to argue that $P \models S$. This is done by arguing that the following relation R:

$$R = \left\{ (p, s) \in St^P \times St^S \mid p_s = p \right\} \tag{19}$$

witnesses the refinement of S by P.

For the opposite direction, assume that S is consistent and show that $s_0 \in \mathsf{cons}^S$. Since S is consistent, then there exists an implementation semantics P and $P \models S$, witnessed by a satisfaction relation R. Without loss of generality consider an implementation, which only has reachable states, and all its states are related to some states of S in R (so R is a total relation). Consider the following subset of states of S:

$$X = \{ s \in St^S \mid (p, s) \in R \text{ for some state } p \text{ of } P \} \tag{20}$$

Obviously $s_0^S \in X$. It suffices to show that X is a post-fixed point of π. Then $s_0 \in X \subseteq \pi(X) \subseteq \mathsf{cons}^S$, since cons^S is the greatest such (post-) fixed point.

Remember that $(p, s) \in R$ for some state p of P. Also p satisfies independent progress. We consider two cases:

- p can delay indefinitely: $\forall d.\ p \xrightarrow{d} p'$ for some state p'. But then also $s \xrightarrow{d} s'$ for some state $s' \in St^S$ and $(p', s') \in R$. So we have that all $s' \in X$. To show that $s \in \pi(X)$ we need to see that $s' \in \overline{err^S}$ and $s \in \mathsf{Idle}_\mathrm{t}^S(\mathsf{ipred}^S(\overline{X}))$. For the former this is quite obvious, as s must satisfy independent progress, if p does. For the latter assume that $s \xrightarrow{d} s' \xrightarrow{i?} s''$ for some $s'' \in \overline{X}$. It must be that $p \xrightarrow{d} p'$ for some state $p' \in St^P$, since p can delay indefinitely, and by satisfaction $(p', s') \in R$. Then also $p' \xrightarrow{i?} p''$ for some state p'' and $(p'', s'') \in R$ by satisfaction. But then $s'' \in X$, which contradicts our assumption that $s'' \in \overline{X}$. Thus all timed successors of s avoid unsafe states as per definition of $\mathsf{Idle}_\mathrm{t}^S(\mathsf{ipred}^S(\overline{X}))$.
- p can delay until a safe output: $\exists d_0 \in \mathcal{R}_{\ge 0}.\ p \xrightarrow{d_0} p' \xrightarrow{o!} p''$ for some states p' and p''. Then by satisfaction $s \xrightarrow{d_0} s' \xrightarrow{o!} s''$ for some states s' and s'', such that (p', s'), $(p'', s'') \in R$, so s', $s'' \in X$. To argue that $s \in \mathsf{Pred}_\mathrm{t}^S(\mathsf{opred}(X), \mathsf{ipred}(\overline{X}))$ it remains to show that $\mathsf{post}_{[0,d_0]}^S(s) \cap \mathsf{ipred}^S(\overline{X}) = \emptyset$. So assume the opposite: $s \xrightarrow{\hat{d}} \hat{s}' \xrightarrow{i?} \hat{s}''$ for some delay $\hat{d} \le d_0$ and states $\hat{s}'$, $\hat{s}''$ with $\hat{s}'' \in \overline{X}$. Since p is time additive we have that $p \xrightarrow{\hat{d}} \hat{p}'$ for some state $\hat{p}' \in St^P$ and by satisfaction $\hat{p}' \xrightarrow{i?} \hat{p}''$ for some state $\hat{p}''$; witnessing that $\hat{s}'$, $\hat{s}'' \in X$, which contradicts our assumption. Thus it must be that $s \in \mathsf{Pred}_\mathrm{t}^S(\mathsf{opred}(X), \mathsf{ipred}(\overline{X}))$. □

Corollary 3. *Consistency can be soundly established for any specification* A *by applying the above procedure that establishes Lemma 5 for* $[\![A]\!]_{\mathrm{sem}}$.

The set of (in)consistent states can be computed for timed games, and thus for specification automata, using controller synthesis algorithms [17]. We discuss it briefly in Section 9.

The inconsistent states can be pruned from a consistent S leading to a locally consistent specification. Pruning is applied in practice to decrease the size of specifications.

For a consistent specification semantics $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ we define the pruned specification semantics $S^\Delta = (\mathsf{cons}^S, s_0, \Sigma^S, \rightarrow^{S^\Delta})$, where $\rightarrow^{S^\Delta} = \rightarrow^S \cap (\mathsf{cons}^S \times (\Sigma^S \cup \mathcal{R}_{\geq 0}) \times \mathsf{cons}^S)$.

Theorem 2. *Let S be a consistent specification semantics. S^Δ is locally consistent and $[\![S]\!]_{\mathrm{mod}} = [\![S^\Delta]\!]_{\mathrm{mod}}$.*

Proof (Theorem 2). All the inconsistent states (that do not fulfill the independent progress condition) are removed from the pruned specification semantics, so obviously S^Δ is locally consistent.

Then, as we proved in Lemma 5, if we consider an implementation P of S and the set $X = \{ s \in St^S \mid (p, s) \in R \text{ for some state } p \text{ of } P \}$ of the states from S that are related to some state in P, then this set of states is consistent: $X \subseteq \mathsf{cons}^S$. This allows to use the same refinement relation R to show that $P \leq S$ iff $P \leq S^\Delta$. □

For specification automata pruning is realized by applying a controller synthesis algorithm, obtaining a maximum winning strategy, which can then be presented as a specification automaton itself.

6.2 Conjunction

Consistency guarantees realizability of a single specification. It is of further interest whether several specifications can be *simultaneously* met by the same component, without reaching error states of any of them. We formalize this notion by defining a logical conjunction for specifications.

Definition 10 (Product $\times$). Let $S = (St^S, s_0^S, \Sigma, \rightarrow^S)$ and $T = (St^T, s_0^T, \Sigma, \rightarrow^T)$ be two specification semantics. A *product* of S and T, written $S \times T$, is defined to be the specification semantics $(St^S \times St^T, (s_0^S, s_0^T), \Sigma, \rightarrow)$, where the transition relation $\rightarrow$ is the largest relation generated by the following rule:

$$\frac{s \xrightarrow{a}{}^S s' \qquad t \xrightarrow{a}{}^T t' \qquad a \in \Sigma \cup \mathcal{R}_{\geq 0}}{(s,t) \xrightarrow{a} (s',t')} \tag{21}$$

In general, a result of the product may be locally inconsistent, or even inconsistent. To guarantee consistency we apply a consistency check to the result, checking if $(s_0, t_0) \in \mathsf{cons}^{S \times T}$ and, possibly, pruning the inconsistent parts:

Definition 11 (Conjunction $\wedge$). For specifications S and T over the same alphabet, such that $S \times T$ is consistent, define $S \wedge T = (S \times T)^\Delta$.

Conjunction is commutative, associative and it is the greatest lower bound for locally consistent specifications in the following sense:

Theorem 3. *For any locally consistent specification semantics S, T, and U over the same alphabet:*

1. $S \wedge T \leq S$ *and* $S \wedge T \leq T$
2. $(U \leq S)$ *and* $(U \leq T)$ *implies* $U \leq (S \wedge T)$
3. $[\![S \wedge T]\!]_{\mathrm{mod}} = [\![S]\!]_{\mathrm{mod}} \cap [\![T]\!]_{\mathrm{mod}}$
4. $[\![(S \wedge T) \wedge U]\!]_{\mathrm{mod}} = [\![S \wedge (T \wedge U)]\!]_{\mathrm{mod}}$

All the above facts naturally translate to syntactic specifications (TIOAs).

We omit the (fairly standard) proof for the first claim. Intuitively the claim holds because $S \times T$ transitions are strictly transitions of S (and of T) and because the pruning producing $(S \times T)^\Delta$ only removes output and delay transitions (which are allowed to be dropped by the refinement). It never removes input transitions from reachable states.

The third claim follows from the first two and the fact that the refinement coincides with model inclusion. The fourth claim follows from repetitive application of the third claim (and the fact that set intersection is the least upper bound in every powerset lattice). We only give a detailed proof for the second claim below.

Proof (Theorem 3.2). Assume that the relation R_1 witnesses $U \leq S$, and relation R_2 witnesses $U \leq T$. First, show that the following set X is a post fixed point of π:

$$X = \{(s,t) \mid \exists u \in St^U . (u,s) \in R_1 \text{ and } (u,t) \in R_2\} \tag{22}$$

Then we know that then $(s,t) \in X \subseteq \pi(X) \subseteq \mathsf{cons}^{S \times T}$, so all states in X are states of the conjunction.

Consider an arbitrary pair (s,t) in X, such that $(u,s) \in R_1$ and $(u,t) \in R_2$ for some state $u \in St^U$. Show that $(s,t) \in \overline{\mathsf{err}^{S \times T}}$. This is easily seen ad absurdum. By Lemma 1 we know that there exists an implementation P and its state p such that $p \models u$. Since P is an implementation semantics it satisfies independent progress. So p can delay independently, or until an output. By satisfaction u can do the same, and by refinement both s and t can do the same. By construction of the product (s,t) can thus do the same, and it cannot be that $(s,t) \in \overline{\mathsf{err}^{S \times T}}$.

Similarly, show $(s,t) \in \mathsf{Idle}_{\mathrm{t}}^{S \times T}\left(\mathsf{ipred}^{S \times T}(\overline{X})\right) \cup \mathsf{Pred}_{\mathrm{t}}^{S \times T}\left(\mathsf{opred}^{S \times T}(X), \mathsf{ipred}^{S \times T}(\overline{X})\right)$. This is again argued by the properties of u (and the fact that U is consistent). Consider the state u witnessing that $(s,t) \in X$. Since U is consistent, it must be that u either admits delaying indefinitely, or it delays until an output.

- Assume that for each delay d there exists a state u' such that $u \xrightarrow{d} u'$ then, by refinement and construction $(s,t) \xrightarrow{d} (s',t')$ for some $(s',t') \in X$. Since u is locally consistent, all intermediate successors states are implementable thus intermediate time successors of (s,t) cannot be in $\mathsf{ipred}^{S \times T}(\overline{X})$. Formally, consider an intermediate successor u'', so $u \xrightarrow{d'} u''$ and thus $(s,t) \xrightarrow{d'} (s'',t'')$ for some (s'',t'') with $(u'',s'') \in R_1$ and $(u'',t'') \in R_2$. Now if $(s'',t'') \xrightarrow{i?} (s''',t''')$ for

some $(s''', t''') \in \overline{X}$ we get a contradiction as by refinement it must be that $u'' \xrightarrow{i?} u'''$ and u''' witnesses that $(s''', t''') \in X$.
- If u cannot delay indefinitely, then it can delay until an output (by local consistency). We use an almost identical argument that then both s and t must be able to do this, and so must their product. Avoiding $\mathsf{ipred}^{S\times T}(\overline{X})$ is argued ad absurdum exactly like in the previous case. So we conclude that X describes a consistent part of the product.

Now it remains to show that U indeed refines the part of $S \times T$ induced by X. This is a standard proof by arguing that the following relation R is a refinement relation:

$$R = \{(u,(s,t)) \in St^U \times X \mid (u,s) \in R_1 \text{ and } (u,t) \in R_2\} \quad (23)$$

Since $X \subseteq \mathsf{cons}^{S\times T}$, we have that R also witnesses refinement of $S \wedge T$ by U. □

We turn our attention to syntactic representations again. Consider two specifications TIOAs $A_1 = (Loc_1, q_0^1, Clk_1, E_1, Act^1, Inv_1)$ and $A_2 = (Loc_2, q_0^2, Clk_2, E_2, Act^2, Inv_2)$ with $Act_i^1 = Act_i^2$ and $Act_o^1 = Act_o^2$. Their conjunction, denoted $A_1 \wedge A_2$, is the TIOA $A = (Loc, q_0, Clk, E, Act^1, Inv)$ given by: $Loc = Loc_1 \times Loc_2$, $q_0 = (q_0^1, q_0^2)$, $Clk = Clk_1 \uplus Clk_2$, $Inv((q_1,q_2)) = Inv(q_1) \wedge Inv(q_2)$. The set of edges E is defined by the following rule:

- If $(q_1, a, \varphi_1, c_1, q_1') \in E_1$ and $(q_2, a, \varphi_2, c_2, q_2') \in E_2$, then $((q_1,q_2), a, \varphi_1 \wedge \varphi_2, c_1 \cup c_2, (q_1', q_2')) \in E$

It might appear as if two systems can only advance on an input if both are ready to receive an input, but because of input enabledness this is always the case.

The following theorem lifts all the results from the TIOTSs level to the symbolic representation level:

Theorem 4. *Let A_1 and A_2 be two specification automata, we have $[\![A_1]\!]_{\text{sem}} \wedge [\![A_2]\!]_{\text{sem}} = [\![A_1 \wedge A_2]\!]_{\text{sem}}$.*

7 Compatibility and Composition

We shall now define *structural composition*, also called *parallel composition*, between specifications. We follow the optimistic approach of [34], i.e., *two specifications can be composed if there exists at least one environment in which they can work together*. Parallel composition is made of three main steps. First, we compute the classical product between timed specifications [42], where components synchronize on common inputs/outputs. The second step is to identify incompatible states in the product, i.e., states in which the two components cannot work together. The last step is to seek for an environment that can avoid such error states, i.e., an environment in which the two components can work together in a safe way. Before going further, we would like to contrast the structural and logical composition.

The main use case for parallel composition is in fact dual to the one for conjunction. Indeed, as observed in the previous section, conjunction is used to reason about internal properties of an implementation set, so if a local inconsistency arises in conjunction we limit the implementation set to avoid it in implementations. A pruned specification can be given to a designer, who chooses a particular implementation satisfying conjoined requirements. A conjunction is consistent if the output player can avoid inconsistencies, and its main theorem states that its set of implementation coincides with the intersection of implementation sets of the conjuncts.

In contrast, parallel composition is used to reason about external use of two (or more) components. We assume an independent implementation scenario, where the two composed components are implemented by independent designers. The designer of any of the environment components can only assume that the composed implementations will adhere to original specifications being composed. Consequently if an error occurs in parallel composition of the two specifications, the *environment* is the only entity that possibly has the power to avoid it. Thus, following [31], we say that a composition is *useful*, and composed components are *compatible*, if the input player has a strategy in the safety game to avoid error states in the composition. The main theorem will state that if an environment is compatible with a useful specification, it is also compatible with any of its refinements, including implementations.

We now propose our formal definition for parallel composition. We consider two specification semantics $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ and $T = (St^T, s_0^T, \Sigma^T, \rightarrow^T)$, and we say that they are *composable* iff their output alphabets are disjoint $\Sigma_o^S \cap \Sigma_o^T = \emptyset$. We say that two specifications are composable if their semantics are composable.

As we did for conjunction, before defining the parallel composition we first introduce a suitable notion of *product*.

Definition 12 (Parallel product $\otimes$). The *parallel product* of S and T, which roughly corresponds to the one defined on timed input/output automata [42], is the specification $S \otimes T = (St^S \otimes St^T, (s_0^S, s_0^T), \Sigma^{S\otimes T}, \rightarrow^{S\otimes T})$, where the alphabet $\Sigma^{S\otimes T} = \Sigma^S \cup \Sigma^T$ is partitioned in 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 of the product is the largest relation generated by the following rules:

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

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

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

Observe that if we compose two locally consistent specifications using the above product rules, then the resulting product is also locally consistent. Since we normally work with consistent specifications in a development process, immediate errors as defined for conjunction are not applicable to parallel composition. Moreover, unlike [34], our specifications are input-enabled, and there is no way to define an error state in which a component can issue an output that cannot be captured by the other component. However, the absence of "model-related" error states allows us to define more elaborated errors, specified by the designer. Those cannot easily be considered in [34].

When reasoning about parallel composition we use model specific error states, i.e., error states indicated by the designer. These error states could arise in several ways. First, a specification may contain an error state in order to model unavailable inputs in presence of input-enabledness (transitions under inputs that the system is not ready to receive, should target such an incompatible state. Typically universal states are used for the purpose of signaling unpredictability of the behaviour after receiving an unanticipated input). Second, a temporal property written in some logic such as TCTL [3] can be interpreted over our specification, which when analyzed by a model checker, will result in a partition of the states into good ones (say satisfying the property) and bad ones (violating the property). Third, an incompatibility in a composition can be propagated from incompatibilities in the composed components. It should always be the case that a state in a product (s,t) is an incompatible state if s is an incompatible state in S, or t is an incompatible state in T.

Formally, we will model all these sources of incompatibility as a set of error states. We will call this set of states, *strictly undesirable* states and refer to it as $\mathsf{undesirable}^S$. In the rest of the section, to simplify the presentation, we will include the set of strictly undesirable states as part of specification definitions.

We say that a specification is *useful* if there exists an environment E that can always avoid reaching a strictly undesirable state, whatever the specification will do. Thus the environment is characterizing a winning strategy for the input player in a safety game to avoid undesirable states. The environment E is said to be *compatible* with S.

We compute the set of useful states of S using a fixpoint characterization. This characterization is a dual of the safety game for consistency presented in the previous sections. We consider a variant of controllable timed predecessor operator, where the roles of the inputs and outputs are reversed:

$$\omega(X) = \overline{\mathsf{undesirable}^S} \cap \Big[\mathsf{Idle}_{\mathrm{t}}^S(\mathsf{opred}^S(\overline{X})) \cup \mathsf{Pred}_{\mathrm{t}}(\mathsf{ipred}(X), \mathsf{opred}(\overline{X}))\Big] \quad (24)$$

Now the set of useful states useful^S can be characterized as the greatest fixpoint of ω, so $\mathsf{useful}^S = \omega(\mathsf{useful}^S)$. Again existence and uniqueness of this fixpoint is warrented by monotonicity of ω. Since the ω is a simple dual of π we omit the proofs in this section, as they are essentially isomorphic to the ones for consistency and conjunction; with exception of the congruence theorem, whose proof is standard.

Theorem 5. *A consistent specification semantics S is useful iff $s_0 \in \mathsf{useful}^S$. A consistent specification A is useful iff $s_0^{[\![A]\!]_{\mathrm{sem}}} \in \mathsf{useful}^{[\![A]\!]_{\mathrm{sem}}}$.*

The proof of Theorem 5 is a dual to the one of Lemma 5.

As for inconsistent states, undesirable states can be pruned from the specification. For a useful specification semantics $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ we define the pruned specification semantics $S^\beta = (\mathsf{useful}^S \cup \{\mathsf{u}\}, s_0, \Sigma^S, \rightarrow^{S^\beta})$, where u is a universal state (allows arbitrary behaviour) and $\rightarrow^{S^\beta} = \rightarrow^S \cap (\mathsf{useful}^S \cup \{\mathsf{u}\} \times (\Sigma^S \cup \mathcal{R}_{\geq 0}) \times \mathsf{useful}^S \cup \{\mathsf{u}\})$. The following theorem shows that pruning the specification does not change the set of compatible environments.

Theorem 6. *Let S be a useful specification semantics. Then E is an environment compatible with S iff E is compatible with S^β.*

The proof of Theorem 6 is a dual to the one of Theorem 2 that shows that P is an implementation semantics of specification S iff P is an implementation semantics of S^Δ.

Having introduced the general notion of usefulness of components and specifications, we are now ready to define compatibility of specifications and parallel composition. We propose the following definition, which is in the spirit of [31].

Definition 13 (Compatibility). Two composable specification semantics S and T are *compatible* iff the initial state of $S \otimes T$ is useful. Two composable specifications A and B are compatible if the initial state of $[\![A]\!]_{\mathrm{sem}} \otimes [\![B]\!]_{\mathrm{sem}}$ is useful.

Definition 14 (Composition $\|$). For two compatible specification semantics S and T define their parallel composition $S \parallel T = (S \otimes T)^{\beta}$, and $\mathsf{undesirable}^{S\|T} = \{(s,t) \mid s \in \mathsf{undesirable}^{S} \text{ or } t \in \mathsf{undesirable}^{T}\}$.

As we have discussed above, the set of strictly undesirable states, $\mathsf{undesirable}^{S\otimes T}$, can be increased by the designer as needed, for example by adding state for which desirable temporal properties about the interplay of S and T do not hold.

Observe that parallel composition is commutative, and that two specifications composed give rise to well-formed specifications. It is also associative in the following sense:

$$[\![(S \parallel T) \parallel U]\!]_{\text{mod}} = [\![S \parallel (T \parallel U)]\!]_{\text{mod}} \qquad (25)$$

Theorem 7. *Refinement is a pre-congruence with respect to parallel composition; for any specification semantics S_1, S_2, and T such that $S_1 \leq S_2$ and S_1 composable with T, we have that S_2 composable with T and $S_1 \parallel T \leq S_2 \parallel T$. Moreover if S_2 compatible with T then S_1 compatible with T.*

Theorem 7 allows the independent implementability scenario:

Corollary 4. *For any consistent specification semantics S and T, such that S is composable with T, $S \parallel T$ is consistent. Moreover, if P_1 is implementation semantics that satisfies S and P_2 is an implementation semantics that satisfies TS, then $P_1 \parallel P_2 \models S \parallel T$.*

Proof. If S is composable with T then P_1 is composable with P_2 since the alphabets are the same. Then a first appplication of Theorem 7 proves that $P_1 \parallel P_2 \leq P_1 \parallel T$, and a second one that $P_1 \parallel T \leq S \parallel T$. Finally since refinement is transitive we proves that $P_1 \parallel P_2 \models S \parallel T$.

We now switch to the symbolic representation. Parallel composition of two specification TIOAs is defined in the following way. Consider two TIOA $A_1 = (Loc_1, q_0^1, Clk_1, E_1, Act_1, Inv_1)$ and $A_2 = (Loc_2, q_0^2, Clk_2, E_2, Act_2, Inv_2)$ with $Act_o^1 \cap Act_o^2 = \emptyset$. Their parallel composition which is denoted $A_1 \parallel A_2$ is the TIOA $A = (Loc, q_0, Clk, E, Act, Inv)$ given by: $Loc = Loc_1 \times Loc_2$, $q_0 = (q_0^1, q_0^2)$, $Clk = Clk_1 \uplus Clk_2$, $Inv((q_1,q_2)) = Inv(q_1) \wedge Inv(q_2)$, and the set of actions $Act = Act_i \uplus Act_o$ is given by $Act_i = Act_i^1 \setminus Act_o^2 \cup Act_i^2 \setminus Act_o^1$ and $Act_o = Act_o^1 \cup Act_o^2$. The set of edges E is defined by the following rules:

- If $(q_1, a, \varphi_1, c_1, q_1') \in E_1$ with $a \in Act_1 \setminus Act_2$ then for each $q_2 \in Loc_2$ $((q_1,q_2), a, \varphi_1, c_1, (q_1',q_2)) \in E$
- If $(q_2, a, \varphi_2, c_2, q_2') \in E_2$ with $a \in Act_2 \setminus Act_1$ then for each $q_1 \in Loc_1$ $((q_1,q_2), a, \varphi_1, c_1, (q_1,q_2')) \in E$
- If $(q_1, a, \varphi_1, c_1, q_1') \in E_1$ and $(q_2, a, \varphi_2, c_2, q_2') \in E_2$ with $a \in Act_1 \cap Act_2$ then $((q_1,q_2), a, \varphi_1 \wedge \varphi_2, c_1 \cup c_2, (q_1',q_2')) \in E$

Just like for conjunction, after the composition, the result can be pruned to limit the representation to useful states. Note that the result of this pruning may lead to a locally inconsistent specification. The consistency pruning (Δ) can be applied subsequently to fix this, if desirable.

Finally, the following theorem lifts all the results from timed input/output transition systems to the symbolic representation level.

Theorem 8. *Let A_1 and A_2 be two specification automata, we have $[\![A_1]\!]_{\text{sem}} \parallel [\![A_2]\!]_{\text{sem}} = [\![A_1 \parallel A_2]\!]_{\text{sem}}$.*

8 Quotient

The quotient operator allows for factoring out behavior from a larger component. If one has a large component specification T and a small one S then $T \backslash\backslash S$ is the specification of all the models that when composed with S refine T. In other words, $T \backslash\backslash S$ specifies the work that still needs to be done, given availability of an implementation of S, in order to provide an implementation of T. We first describe the theory behind the operator, then we show how it can be exploited to reason on assumptions and guarantees.

We have the following requirements on the sets of inputs and outputs of the dividend T and the divisor S when applying quotienting: $\Sigma_{\text{i}}^S \subseteq \Sigma_{\text{i}}^T \cup \Sigma_{\text{o}}^T$ and $\Sigma_{\text{o}}^S \subseteq \Sigma_{\text{o}}^T$ (and S must be well-formed, so Σ_{i}^S and Σ_{o}^S are disjoint).

We proceed similarly to structural and logical compositions, and start with a pre-quotient that may introduce error states. Those errors are then pruned to obtain the quotient.

Definition 15 (Pre-quotient $\diagup$). Given two specification semantics $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ and $T = (St^T, t_0^T, \Sigma^T, \rightarrow^T)$ their *pre-quotient* is a specification semantics $T \diagup S = (St, (s_0^S, t_0^T), \Sigma, \rightarrow)$, where $St = (St^S \times St^T) \cup \{\mathsf{u}, \mathsf{e}\}$ where u and e are fresh states such that, u is universal (allows arbitrary behaviour), and e is inconsistent (no output-controllable behaviour can satisfy it). State e disallows progress of time and has no output transitions. The universal state guarantees nothing about the behaviour of its implementations (thus any refinement with a suitable alphabet is possible), and dually the inconsistent state allows no implementations.

Moreover we require that $\Sigma = \Sigma^T$ with $\Sigma_{\text{i}} = \Sigma_{\text{i}}^T \cup \Sigma_{\text{o}}^S$ and $\Sigma_{\text{o}} = \Sigma_{\text{o}}^T \setminus \Sigma_{\text{o}}^S$. Finally the transition relation $\rightarrow^{T \diagup S}$ is the largest relation generated by the following rules:

$$\frac{t \xrightarrow{a}{}^{T} t' \qquad s \xrightarrow{a}{}^{S} s' \qquad a \in \Sigma^S \cup \mathcal{R}_{\geq 0}}{(t,s) \xrightarrow{a}{}^{T \diagdown S} (t',s')}\,[\mathsf{all}]$$

$$\frac{s \not\xrightarrow{a}{}^{S} \qquad a \in \Sigma^S_{\mathrm{o}} \cup \mathcal{R}_{\geq 0}}{(t,s) \xrightarrow{a}{}^{T \diagdown S} \mathsf{u}}\,[\mathsf{unreachable}]$$

$$\frac{t \not\xrightarrow{a}{}^{T} \qquad s \xrightarrow{a}{}^{S} s' \qquad a \in \Sigma^S \cap \Sigma^T_{\mathrm{o}}}{(t,s) \xrightarrow{a}{}^{T \diagdown S} \mathsf{e}}\,[\mathsf{unsafe}]$$

$$\frac{t \xrightarrow{a}{}^{T} t' \qquad a \in \Sigma^T \setminus \Sigma^S}{(t,s) \xrightarrow{a}{}^{T \diagdown S} (t',s)}\,[\mathsf{dividend}]$$

$$\frac{a \in \Sigma \cup \mathcal{R}_{\geq 0}}{\mathsf{u} \xrightarrow{a}{}^{T \diagdown S} \mathsf{u}}\,[\mathsf{universal}] \qquad \frac{a \in \Sigma_{\mathrm{i}}}{\mathsf{e} \xrightarrow{a}{}^{T \diagdown S} \mathsf{e}}\,[\mathsf{inconsistent}]$$

It is not hard to see that the pre-quotient $T \diagdown S$ is input-enabled. Inputs of $T \diagdown S$ are $\Sigma_{\mathrm{i}} = \Sigma^T_{\mathrm{i}} \cup \Sigma^S_{\mathrm{o}}$. The universal state u (respectively the inconsistent state e) is input-enabled for Σ_{i} due to the [universal] (resp. [inconsistent]) rule. For the remaining states input-enabledness follows from the remaining rules. Let $a \in \Sigma_{\mathrm{i}}$. For $a \in \Sigma^S_{\mathrm{o}}$ we get that the transition exists by the [unreachable], [unsafe], or [all] rule. Otherwise, if $a \in \Sigma^T_{\mathrm{i}}$ a transition is induced by the [dividend], or [all] rule.

Theorem 9 states that the proposed pre-quotient operator has exactly the property that it is dual of structural composition with regards to refinement.

Theorem 9. *For any two specification semantics S and T such that the pre-quotient $T \diagdown S$ is defined, and for any implementation semantics X over the same alphabet as $T \diagdown S$, we have that $S \parallel X$ is defined and $S \parallel X \leq T$ iff $X \leq T \diagdown S$.*

We now give the proof for Theorem 9. First observe that since X has the same input and output alphabets as $T \diagdown S$, sets Σ^X_{o} and Σ^S_{o} are disjoint, and thus $S \parallel X$ is defined. We split the argument for the two directions of the equivalence into two separate lemmas below.

Lemma 6. *For any two specification semantics S and T such that $T \diagdown S$ is defined, and an implementation X over the same alphabet as $T \diagdown S$:*

$$S \parallel X \leq T \text{ implies } X \leq T \diagdown S$$

Proof (Lemma 6). We have the refinement relation R_1 showing that $S \parallel X \leq T$ and need to present a relation witnessing $X \leq T \diagdown S$. Consider:

$$R_2 = \{(x,(t,s)) \mid ((s,x),t) \in R_1\} \cup \{(x,\mathsf{u}) \mid x \in St^X\} \quad (26)$$

We have to prove that R_2 is a refinement relation. Let $(x,(t,s)) \in R_2$.

- Assume that $(t,s) \xrightarrow{i?} (t',s')$. Need to show that $x \xrightarrow{i?} x'$ and $(x',(t',s')) \in R_2$. Split in sub-cases depending on which rule was used to conclude $(t,s) \xrightarrow{i?} (t',s')$.
  - [all] If both $t \xrightarrow{i!} t'$ and $s \xrightarrow{i!} s'$ then:
    as x is input-enabled we have $x \xrightarrow{i?} x'$ and by [sync-io] that $(s,x) \xrightarrow{i!} (s',x')$. Then since $(s,x),t) \in R_1$ it must be that $((s',x'),t') \in R_1$ and $(x',(t',s')) \in R_2$.
    Similarly if both $t \xrightarrow{i?} t'$ and $s \xrightarrow{i?} s'$ then:
    because x is input-enabled we have $x \xrightarrow{i?} x'$ and by rule [sync-in] we have $(s \parallel x) \xrightarrow{i?} (s' \parallel x')$ and thus $((s',x'),t') \in R_1$, which allows concluding that $(x',(t',s')) \in R_2$.
    Observe that other input/output combinations with an application of [all] are not possible here: $t \xrightarrow{i!} t'$ and $s \xrightarrow{i?} s'$ would result in an output of the quotient, contradicting the assumption; $t \xrightarrow{i?} t'$ and $s \xrightarrow{i!} s'$ is impossible as $\Sigma^S_{\mathrm{o}} \subset \Sigma^T_{\mathrm{o}}$ and the inputs are disjoint from outputs.
  - [unreachable] Assume premise of [unreachable]. Then $(t,s) \xrightarrow{i?} \mathsf{u}$. By input-enabledness of x get $x \xrightarrow{i?} x'$ and by construction: $(x',\mathsf{u}) \in R_2$.
  - [unsafe] If $i \in \Sigma^T_{\mathrm{o}} \cap \Sigma^S_{\mathrm{o}}$ then this rule cannot be used to conclude that $(t,s) \xrightarrow{i?} (t',s')$ because then $t \not\xrightarrow{i!}$ and $s \xrightarrow{i!} s'$, which implies that $((s,x),t) \notin R_1$ (or that R_1 is not a refinement relation).
    If $i \in \Sigma^T_{\mathrm{o}} \cap \Sigma^S_{\mathrm{i}}$ then $i \in \Sigma^{T \diagdown S}_{\mathrm{o}}$ so it cannot be that $(t,s) \xrightarrow{i?}$.
  - [dividend] We have that $t \xrightarrow{i?} t'$ and, by input-enabledness, $x \xrightarrow{i?} x'$ and $i \notin \Sigma^S$. By [indep-r] obtain $(s,x) \xrightarrow{i?} (s',x')$, which with $((s,x),t) \in R_1$ allows concluding $((s,x'),t') \in R_1$ and in turn $(x',(t',s)) \in R_2$.
  - [universal] Then $(t,s) = \mathsf{u}$. It is trivial to see that the transitions induced by this rule satisfy the definition of refinement.
  - [inconsistent] Then $(t,s) = \mathsf{e}$. This rule could have not been used to induce $(t,s) \xrightarrow{i?} (t',s')$, simply because $(x,\mathsf{e}) \notin R_2$.
- Assume $x \xrightarrow{o!} x'$ and show that $(t,s) \xrightarrow{o!} (t',s')$ and $(x',(t',s')) \in R_2$. Note that $o \in \Sigma^X_{\mathrm{o}} = \Sigma^T_{\mathrm{o}} \setminus \Sigma^S_{\mathrm{o}}$
  If $o \in (\Sigma^T_{\mathrm{o}} \setminus \Sigma^S_{\mathrm{o}}) \cap \Sigma^S_{\mathrm{i}}$ and then by the parallel composition rule [sync-io] we have $(s,x) \xrightarrow{o!} (s',x')$ and since $((s,x),t) \in R_1$ then also $t \xrightarrow{o!} t'$ for some state t' and $((s',x'),t') \in R_1$. But then by construction also $(x',(s',t')) \in R_2$. It remains to see that $(t,s) \xrightarrow{o!} (t',s')$, but this follows from rule [all].
  If $o \in (\Sigma^T_{\mathrm{o}} \setminus \Sigma^S_{\mathrm{o}}) \setminus \Sigma^S_{\mathrm{i}}$ the argument is analogous, except that [indep-r] and [dividend] are used instead of respectively [sync-io] and [all].
- Assume that $x \xrightarrow{d} x'$ and show $(t,s) \xrightarrow{d} (t',s')$ and $(x',(t',s')) \in R_2'$.
  If $s \not\xrightarrow{d}$ then we can conclude by [unreachable] that $(t,s) \xrightarrow{d} \mathsf{u}$ and $(x',\mathsf{u}) \in R_2$. Otherwise, if $s \xrightarrow{d} s'$ then $(s,x) \xrightarrow{d} (s',x')$ and by $((s,x),t) \in R_1$ we know that $t \xrightarrow{d} t'$ for some state t' and $((s',x'),t') \in R_1$ which in turn gives $(x',(t',s')) \in R_2$. It remains to show that $(t,s) \xrightarrow{d} (t',s')$, which follows from [all].

Lemma 7. *For any two specification semantics S and T such that $T \nwarrow S$ is defined, and an implementation X over the same alphabet as $T \nwarrow S$:*

$$S \parallel X \leq T \Longleftarrow X \leq T \nwarrow S$$

Proof (Lemma 7).

We have the refinement relation R_2 witnessing that $X \leq T \nwarrow S$ and want to give a relation showing that $S \parallel X \leq T$. Consider:

$$R_1 = \{((s,x),t) \mid (x,(t,s)) \in R_2\}$$

We have to prove that R_1 is a refinement relation. Assume that $(x,(t,s)) \in R_1$.

- Assume that $t \xrightarrow{i?} t'$ and show states s', x' such that $(s,x) \xrightarrow{i?} (s',x')$ and $((s',x'),t') \in R_1$.
  Since x is input-enabled then $x \xrightarrow{i?} x'$ for some x'. If $i \in \Sigma_{\mathrm{i}}^S$ then also $s \xrightarrow{i?} s'$ and by rule [sync-io] we have that $(s,x) \xrightarrow{i?} (s',x')$. Further by [all] also $(t,s) \xrightarrow{i?} (t',s')$ and since $(x,(t,s)) \in R_2$ also $(x',(t',s')) \in R_2$. This by construction gives $((s',x'),t') \in R_1$.
  Otherwise, if $i \in \Sigma_{\mathrm{i}}^T \setminus \Sigma_{\mathrm{i}}^S$, use an analogous argument relying on [indep-r] and [dividend] rules instead of respectively [sync-io] and [all].
- Assume that $(s,x) \xrightarrow{o!} (s',x')$ and show that $t \xrightarrow{o!} t'$ and $((s',x'),t') \in R_1$ for some state t'.
  Case 2.1: If $o \in \Sigma_{\mathrm{o}}^X \cap \Sigma_{\mathrm{o}}^S$ then have $s \xrightarrow{o!} s'$ and $x \xrightarrow{o?} x'$ by rule [sync-io]. Assume that $t \not\xrightarrow{o!}$. Then by [unsafe] $(t,s) \xrightarrow{o?} \mathsf{e}$ and (determinism and independent-progress!) it cannot be that $(x,(t,s)) \in R_2$, since $(x',\mathsf{e}) \notin R_2$ for any x'. So there must exist t' such that $t \xrightarrow{o!} t'$. Moreover by [all] we get $(t,s) \xrightarrow{o?} (t',s')$ and since $(x,(t,s)) \in R_2$ also get $(x',(t',s')) \in R_2$. By construction of R_1 get $((s',x'),t') \in R_1$.
  Case 2.2: Assume $o \in \Sigma_{\mathrm{o}}^X \cap \Sigma_{\mathrm{i}}^S$. Then have $s \xrightarrow{o?} s'$ and $x \xrightarrow{o!} x'$ by [sync-io]. We use the same argument as above to conclude that $t \xrightarrow{o!} t'$ for some state t'. Otherwise [unsafe] allows concluding that $(t,s) \xrightarrow{o!} \mathsf{e}$ and $(x,(t,s)) \in R_2$ is violated as $(x',\mathsf{e}) \notin R_2$. By [all] we get $(t,s) \xrightarrow{o!} (t',s')$ and since $(x,(t,s)) \in R_2$ also get $(x',(t',s')) \in R_2$. By construction of R_1 get $((s',x'),t') \in R_1$.
  Case 2.3: If $o \in \Sigma_{\mathrm{o}}^X \setminus \Sigma^S = \Sigma_{\mathrm{o}}^T \setminus \Sigma^S$ then by [indep-r] we have $(s,x) \xrightarrow{o!} (s,x')$ with $x \xrightarrow{o!} x'$ Further, by [dividend] have $(t,s) \xrightarrow{o!} (t',s)$ and, since $(x,(t,s)) \in R_2$, also $(x',(t',s)) \in R_2$ which in turn gives $((s,x'),t') \in R_1$ by construction of the latter.
- Assume $(s,x) \xrightarrow{d} (s',x')$ and show that $t \xrightarrow{d} t'$ and $((s',x'),t') \in R_1$. By [delay] we have that $s \xrightarrow{d} s'$ and $x \xrightarrow{d} x'$. Since $x \xrightarrow{d} x'$, $s \xrightarrow{d} s'$ and $(x,(t,s)) \in R_2$ it must be that $(t,s) \xrightarrow{d} (t',s')$ (because only rule [all] could have been used) and $(x',(t',s')) \in R_2$. Thus also $t \xrightarrow{d} t'$ from the premise of [all] and $((s',x'),t') \in R_1$

Finally, the actual quotient, denoted $T \backslash\!\backslash S$, is defined if $T \nwarrow S$ is consistent. It is obtained by pruning the states of the pre-quotient $T \nwarrow S$ from where the implementation has no strategy to avoid immediate errors states $\mathsf{err}^{T \backslash\!\backslash S}$ using the same game characterization like in Section 6. It follows from Theorem 2 that Theorem 9 also holds for the actual quotient operator $\backslash\!\backslash$ (as opposed to the pre-quotient).

Definition 16 (Quotient $\backslash\!\backslash$). For any specifications S and T such that $T \nwarrow S$ is defined and consistent, define $T \backslash\!\backslash S = (T \nwarrow S)^{\Delta}$.

Quotienting for specifications (TIOAs) is defined in the following way. Consider two specifications $A_T = (Loc_T, q_0^T, Clk_T, E_T, Act_T, Inv_T)$ and $A_S = (Loc_S, q_0^S, Clk_S, E_S, Act_S, Inv_S)$ with $Act_i^S \subseteq Act_i^T$ and $Act_o^S \subseteq Act_o^T$. The quotient, which is denoted $A_T \backslash\!\backslash A_S$ is the TIOA given by: $Loc = Loc_T \times Loc_S \cup \{l_u, l_\emptyset\}$, $q_0 = (q_0^T, q_0^S)$, $Clk = Clk_T \uplus Clk_S \uplus \{x_{new}\}$, $Inv((q_T,q_S)) = Inv(l_u) = true$ and $Inv(l_\emptyset) = \{x_{new} \leq 0\}$. The two new states l_u and $l_\emptyset$ are respectively universal and inconsistent. The set of actions $Act = Act_i \uplus Act_o$ is given by $Act_i = Act_i^T \cup Act_o^S \cup \{i_{new}\}$ and $Act_o = Act_o^T \setminus Act_o^S$.

The set of edges E is defined by the following rules:

- [unreachable1] For each $q_T \in Loc_T, q_S \in Loc_S$ and $a \in Act$, $((q_T,q_S),a,\neg Inv_S(q_S),\{x_{new}\},l_u) \in E$.
- [unsafe1] For each $q_T \in Loc_T, q_S \in Loc_S$, $((q_T,q_S),i_{new},\neg Inv_T(q_T) \wedge Inv_S(q_S),\{x_{new}\},l_\emptyset) \in E$.
- [all] If $(q_T,a,\varphi_T,c_T,q_T') \in E_T$ and $(q_S,a,\varphi_S,c_S,q_S') \in E_S$, then $((q_T,q_S),a,\varphi_T \wedge \varphi_S, c_T \cup c_S, (q_T',q_S')) \in E$
- [unsafe2] For each $(q_S,a,\varphi_S,c_S,q_S') \in E_S$ with $a \in Act_o^S$, $((q_T,q_S),a,\varphi_S \wedge \neg G_T, \{x_{new}\}, l_\emptyset) \in E$ where $G_T = \bigvee\{\varphi_T \mid (q_T,a,\varphi_T,c_T,q_T')\}$
- [dividend] For each $(q_T,a,\varphi_T,c_T,q_T') \in E_T$ and $a \notin Act_S$, $((q_T,q_S),a,\varphi_T,c_T,(q_T',q_S)) \in E$
- [unreachable2] For each $(q_T,a,\varphi_T,c_T,q_T') \in E_T$ with $a \in Act_o^S$, $((q_T,q_S),a,\neg G_S,\{\},l_u) \in E$ where $G_S = \bigvee\{\varphi_S \mid (q_S,a,\varphi_S,c_S,q_S')\}$
- [universal] For each $a \in Act_i$, $(l_\emptyset,a,true,\{\},l_\emptyset) \in E$
- [inconsistent] For each $a \in Act$, $(l_u,a,true,\{\},l_u) \in E$

Finally, the following theorem lifts all the results from timed input/output transition systems to the symbolic representation level.

Theorem 10. *Let A_1 and A_2 be two specification automata, we have*

$$(\llbracket A_1 \rrbracket_{\mathrm{sem}} \nwarrow \llbracket A_2 \rrbracket_{\mathrm{sem}})^{\Delta} = (\llbracket A_1 \nwarrow A_2 \rrbracket_{\mathrm{sem}})^{\Delta} \qquad (27)$$

8.1 Assumptions and Guarantees

In the following we will illustrate the utility of quotienting. This section is a summary of results presented in [25]. The contribution of the present paper is in applying the definition presented in this section to the parking example of Section 11. We start with an example

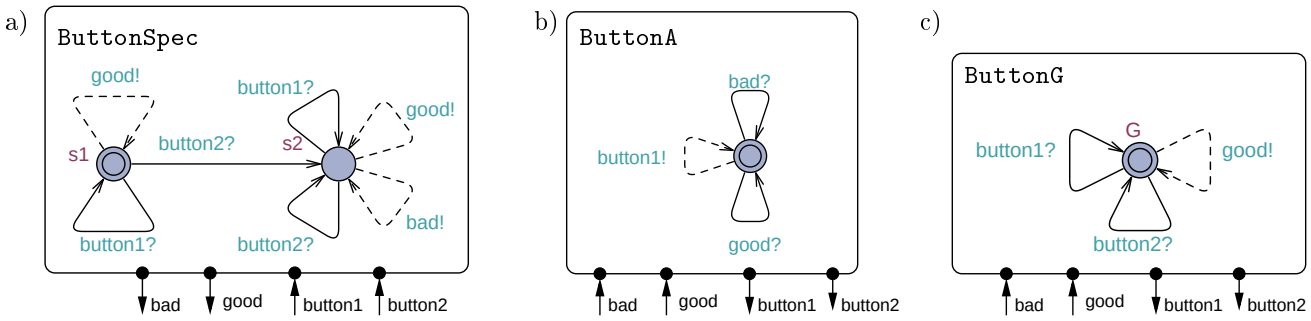

Figure 9: Specification of a) the `ButtonSpec`, b) the assumption `ButtonA`, c) the guarantee `ButtonG`.

that consists of three Timed I/O Automata specifications as shown in Fig. 9. We start with a simple specification, shown in Fig. 9(a) of a system with two buttons. The specification states that as long as only `button1` is pressed (assumption) then only `good` output will be produced (guarantee). If at some point `button2` is pressed then the system could start to produce `bad` output. Figure 9 thus represents the combination of assumptions and guarantees, each of them being described with a TIOA. In general, one does not obtain such specification directly, but rather from the combination of some automata representing the assumptions and the guarantees. We now show how quotient can be used to combine assumptions and guarantees to obtain the automaton in Fig.9.

The following definition taken from [25] presents an operator known as weaken or weakening, that is used for easier specification of assume guarantee specifications. Weakening computes the largest guarantee one can get under some assumption.

Definition 17 (Weaken $>>$). For any specifications A and G we define $G >> A$ as follows:

$$G >> A \equiv (A||G)\backslash\backslash A$$

Let us go back to our example and show how it can exploit the weakening operator. We would like to express the assumptions and guarantees that we have to the system separately, and then retrieve the automaton in Fig.9. In Fig. 9(b) we specify the assumption that `button2` is never pressed while in Fig. 9(c) we specify the guarantee that the system never produces `bad` output. Even though, in this example, our `ButtonSpec` is quite simple the assumption `ButtonA` and guarantee `ButtonG` are even simpler and extremely easy to understand. We then compute ButtonG >> ButtonA and show that it coincides (in terms of refinement) with ButtonSpec, i.e., we use ECDAR to prove the following two refinements:

```
refinement: (ButtonG >> ButtonA) <= ButtonSpec
refinement: ButtonSpec <= (ButtonG >> ButtonA)
```

Thus effectively being able to substitute `ButtonG` >> `ButtonA` for `ButtonSpec` in any context.

The possibility of splitting assumptions from guarantees becomes even more appealing when having multiple assumptions and guarantees that are conjoined.

9 Tool Support

Our specification theory has been implemented in a new tool called ECDAR. We shall now describe the functionality of the tool, then provide some details on the various game-based algorithms implemented in ECDAR, and finally demonstrate what is possible in the tool using a small case study. ECDAR is freely available at `ecdar.cs.aau.dk`.

9.1 Architecture and Functionality

The architecture of ECDAR builds on UPPAAL. The tool features a graphical user interface (GUI), and a model-checker in the form of a server or a standalone verifier. The user can edit, simulate, and specify properties in the GUI.

Editor. The timed I/O automata (TIOA) are represented as graphs with solid (input) and dashed (output) edges. Since TIOAs must be input enabled, only broadcast communications are allowed. The user has access to the other features of the language such as user-defined types and functions. All figures of specifications and implementations in this paper have been made using the editor of ECDAR.

Simulator. The simulator, based on UPPAAL-TIGA, shows networks of automata and will allow the user to select transitions according to how components are composed (parallel composition or conjunction). The simulator supports open systems and follows the semantics of TIOAs as described in this paper. It cannot at the moment simulate systems where the quotient operator is used[1].

[1] The quotient generates components that cannot be displayed in the GUI.

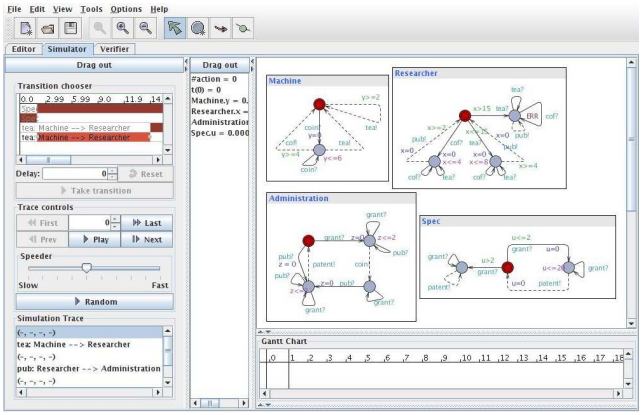

Figure 10: Playing a refinement counter-strategy in the simulator.

Specification Interface. Another view in the interface is used to specify properties using the expressions of our theory. This view is similar to UPPAAL's model checking view. Unlike in UPPAAL, the simulator only works when a query has been checked previously because the structure of the system (as given by the different operations) is defined in the query.

The properties supported are of the following types:

- consistency check with the syntax `consistency:` *system*,
- refinement check with the syntax `refinement:` *system* `<=` *system*
- implementation check with the syntax `implementation:` *system*,

where *system* is a composition of components using the parallel composition, conjunction, or quotient operator. The consistency and refinement checks follow directly the algorithms presented in this paper. The engine can check if a system is an implementation according to the constraints we have defined, such as output urgency and independent progress.

The tool provides a strategy to prove or disprove the property, which can be used to refine the model. The strategy can be played interactively. Fig. 10 shows a screenshot of such an interactive game. When the checked property is satisfied for consistency and implementation, the user can choose inputs and the engine responds with outputs. For refinement it is an alternating 2-player game: the user plays the attacker and the engine the defender if the property is satisfied. If the property is not satisfied, the roles are inverted. ECDAR can also output the resulting strategy in a textual format.

9.2 Implementation of ECDAR

ECDAR exploits the verification engine for timed games implemented in UPPAAL-TIGA, the game extension of UPPAAL [9,10]. ECDAR differs from UPPAAL-TIGA by implementing compositional reasoning primitives.

The Game Solver of UPPAAL-TIGA The engine of UPPAAL-TIGA supports the computation of winning strategies for timed games with respect to a large class of timed temporal logic winning objectives such as reachability/safety or even Büchi. All the algorithms implemented in UPPAAL-TIGA build on the so-called reachability algorithm of UPPAAL-TIGA introduced in [17]. Roughly speaking, this algorithm uses an on-the-fly approach to perform forward exploration of reachable states and back-propagation of (so-far) computed winning states in an interleaved manner using fixed-point operators as shown in this paper. Crucial to any game solving algorithm is the symbolic representation and efficient manipulation of state-sets. In UPPAAL-TIGA, our symbolic representations exploit *zones*, *i.e.* sets of clock valuations characterized by constraints on individual clocks and clock-differences. In particular the operators used in the fixpoint algorithm of UPPAAL-TIGA are computed using *federations* (unions of zones). In addition, the engine implements the turn-based game solver of [15]. We refer to this engine as the *simulation* engine.

The Game Solver of ECDAR The engine of ECDAR reuses the same basic design as UPPAAL-TIGA to implement its consistency checker with the addition of a special component to characterize consistent states. In addition, all components implementing the semantics of the transition system are changed on-the-fly to choose between the different operations of *parallel composition*, *conjunction*, and *quotienting*. ECDAR also reuses the generated state graphs as internal inputs for incremental consistency checks whereas UPPAAL-TIGA only takes a network of timed game automata as input. Before using the result of a consistency check (for refinement or to apply an operation), the state-graph is *pruned* with respect to the strategy obtained from the consistency game. The procedure is as follows: for every symbolic state, restrict it to the winning states of the strategy; and for every output transition, restrict to the ones allowed by the strategy (by strengthening its guard). The pruning feature is absent from UPPAAL-TIGA.

The consistency checker is used to check whether a specification admits at least one implementation. This question reduces to the one of deciding if there exists a strategy for the output player to avoid reaching *bad states* in the specification, i.e., states that do not satisfy the *independent progress* property. To solve this consistency game, we apply the reachability algorithm of UPPAAL-TIGA where input transitions are controllable, output transitions uncontrollable, and where states that do not have any outputs nor allow time to elapse are target states. The game is then solved as in UPPAAL-TIGA, but with different components that change the semantics and with the addition of pruning.

The refinement checker is used to decide whether an implementation satisfies a given specification or if a specification refines another one. As we already said, refinement checking reduces to a 2-player alternating game. To solve this game, we change the rules of the simulation game of UPPAAL-TIGA to match the semantics of refinement, i.e., the rules w.r.t. controllable and uncontrollable transitions are inverted. In this game where we check the refinement $S \leq T$, the first player (the attacker) plays outputs on S and inputs on T, whereas the second player (the defender) plays inputs on S and inputs on T. The product of S and T according to these rules is then constructed on-the-fly, which is the forward exploration step. We detect error states on-the-fly and we back-propagate them. There are two kinds of error states: 1) Either the attacker may delay and violates invariants on T, which is, the defender cannot match a delay, or 2) the defender has to play a given action and cannot do so, i.e., a deadlock. This is similar to UPPAAL-TIGA in principle, except that the underlying structures are different: a pruned state-graph for ECDAR and a network of timed game automata for UPPAAL-TIGA.

We discuss checking for independent progress, output determinism, and output urgency in more detail. A symbolic state is a tuple $\langle q, Z\rangle$, where q is a location, and Z a zone [48]. A state is not urgent, if its invariant allows a positive delay:[2]

$$urgent(\langle q, Z\rangle) \equiv \forall\, v \in Z.\ \exists d \geq 0.\ v{+}d \models inv(q)\ \Rightarrow d = 0$$

A state is *unbounded* if Z has no upper bound, i.e., it contains valuations where it is possible to delay infinitely. Since we are handling convex sets defined with difference constraints, if a state is unbounded then it is possible to delay infinitely from all its valuations.

$$unbounded(\langle q, Z\rangle) \equiv \{v \in Z \mid \forall t \geq 0.\ v + t \in Z\} \neq \emptyset$$

Algorithm 1 combines these notions to check for independent progress. We check that for this notion of deadlock in lines 3–5. For a set of clock valuations Z we write $Z^{\downarrow}$ (line 5) meaning the set of its time predecessors: $Z^{\downarrow} = \{v \mid \exists\, d \geq 0.\ v + d \in Z\}$.

Algorithm 2 shows how we check for output determinism. It is applied iteratively to every reachable symbolic state of a specification. For an output o and a symbolic state (q, Z) we identify edges that can be enabled in this state, and check whether they cause nondeterminism.

Output urgency for implementations is established by constructing a zone graph and checking the following condition for each symbolic state $\langle q, Z\rangle$:

$$\neg urgent(\langle q, Z\rangle) \Rightarrow \text{ for each edge } (q, o!, \varphi, c, q').\ \varphi \cap NextInv(q') \cap Z = \emptyset.$$

[2] ECDAR borrows from UPPAAL the syntactic constructs to obtain this effect conveniently: urgent locations and urgent channels

Algorithm 1: Symbolic check for independent progress.

function *consistent*$(\langle q, Z\rangle)$
1 **if** *unbounded*$(\langle q, Z\rangle)$ **then return true**
2 $deadlock \leftarrow Z$
3 **if** *urgent*$(\langle q, Z\rangle)$ **then**
4 **foreach** *edge* $(q, o!, \varphi, c, q')$ **do**
$deadlock \leftarrow deadlock \setminus (\varphi \cap NextInv(q') \cap Z)$
5 **else foreach** *edge* $(q, o!, \varphi, c, q')$ **do**
$deadlock \leftarrow deadlock \setminus (\varphi \cap NextInv(q') \cap Z)^{\downarrow}$
6 **return** $deadlock = \emptyset$

Algorithm 2: Symbolic check for output determinism.

function *output-determinism*$(\langle q, Z\rangle)$
1 $succ \leftarrow \emptyset$
2 $succ = \{e = (q, o!, \varphi, c, q') \mid \varphi \cap NextInv(q') \cap Z \neq \emptyset$ for any guard φ and output $o!\}$
3 **foreach** *pair of edges* $(e_1, e_2) \in succ.\ e_1 \neq e_2 \wedge output(e_1) = output(e_2)$ **do**
4 **let** $(q_1, o!, \varphi_1, c_1, q_1') = e_1$ and $(q_2, o!, \varphi_2, c_2, q_2') = e_2$
5 **if** $\varphi_1 \cap \varphi_2 \cap Z \neq \emptyset$ **then return false**
end
6 **return true**

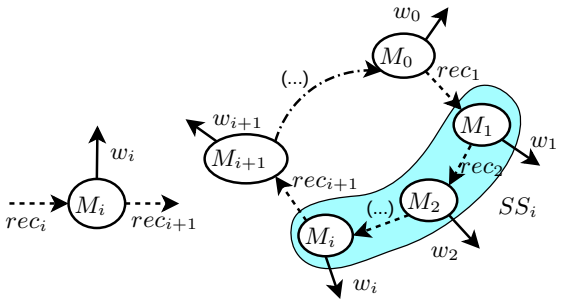

Figure 11: Overview of Milner's scheduler example and the sub-specification SS_i.

10 Application 1: Milner's Scheduler Case Study

We use a modified real-time version of Milner's scheduler algorithm, to show how inductive arguments for refinement can be constructed using compositional operators of our theory. The model consists of N nodes arranged in a ring. A token is sent around, which takes some time, and the nodes on the ring perform some work when the token arrives. Fig. 11 (left) shows a single node that can receive a token on rec_i. The node subsequently begins external work by outputting on w_i. In parallel to this it

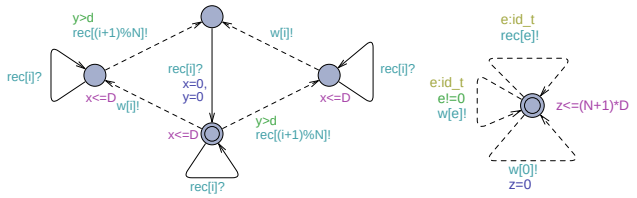

Figure 12: Left: Template for a single node M_i. Right: Template for the overall specification.

can forward the token by outputting on rec_{i+1}, but only after a delay between d and D time units. Fig. 11 (right) illustrates a ring of such nodes M_i in which some nodes have been grouped together. This grouping exemplifies a part of the specification, which we will later be able to replace with an abstraction SS_i in order to execute a compositional proof.

We model the scheduler using templates in a modular way, which allows us to scale the model by instantiating as many nodes as needed. A single node of our scheduler is shown in the left side of Fig. 12. In the initial location of the specification, it is ready to receive a message on the channel `rec[i]?`. After this there are two ways to return to the initial state depending on the order in which it starts its work (`w[i]!`) and passes on the token (`rec[(i+1)%N]!`). The first node of the system M_0 is instantiated with a different initial location (the bottom-most one), reflecting the fact that it holds the token initially. The right side of Fig. 12 shows the overall specification S_0 of the system. It requires that `w[0]!` occurs at least every $(N+1) * D$ time units. Remaining actions can be executed freely.

One way to verify that the scheduler is correct is to verify a property of the type:

```
refinement: ( M0 || M1 || M2 || M3 || M4 ) <= S0
```

We call this type of verification *monolithic*, since it constructs a specification precisely representing the entire system. It is natural to verify the monolithic property in order to show that the composed system refines the overall specification. Unfortunately, this strategy fails due to state-space explosion. As the number of components is increased, the state space grows, and more interleaving is introduced in the system.

In order to combat the problem we apply compositional verification. The idea is to create N sub-specifications that are used in a series of refinement steps. First one shows that $M_1 \leq SS_1$. After this it is proved for increasing indexes, 1 to N that $SS_i||M_{i+1} \leq SS_{i+1}$. Finally the property $SS_n||M_0 \leq S_0$ is checked. Fig. 13 gives the properties for five nodes. The sub-specification aims at capturing the important aspect of the subsystem needed for the next step in the verification process of the overall property. It is very important to notice that the sub-specification is, like all the other components in

```
refinement: M1 <= SS1
refinement: ( SS1 || M2 ) <= SS2
refinement: ( SS2 || M3 ) <= SS3
refinement: ( SS3 || M4 ) <= SS4
refinement: ( SS4 || M0 ) <= S0
```

Figure 13: Incremental verification.

	$d = 29$	20	10	9	8	6	4
$n = 5$	0.080	0.097	0.191	*0.169*	*0.172*	*0.151*	*0.205*
monolithic	0.034	0.034	0.073	1.191	1.189	64.933	> 600
$n = 6$	0.102	0.133	0.231	*0.228*	*0.238*	*0.238*	*0.294*
monolithic	0.040	0.043	0.095	6.786	6.791	> 600	> 600
$n = 8$	0.225	0.349	0.516	**0.515**	*0.540*	*0.600*	*0.582*
monolithic	0.076	0.076	0.230	88.542	88.642	> 600	> 600
$n = 12$	0.830	1.414	1.802	**1.895**	**1.831**	*2.079*	*2.181*
monolithic	0.220	0.223	0.843	> 600	> 600	> 600	> 600
$n = 20$	4.990	9.739	12.377	**11.923**	**12.041**	**12.438**	*12.764*
monolithic	1.038	1.030	4.523	> 600	> 600	> 600	> 600
$n = 30$	22.053	45.709	55.728	**55.345**	**55.112**	**54.702**	*56.164*
monolithic	3.791	3.778	17.652	> 600	> 600	> 600	`om`

Table 1: Results of the verification experiments. Timings in seconds. `om` = 4GB.

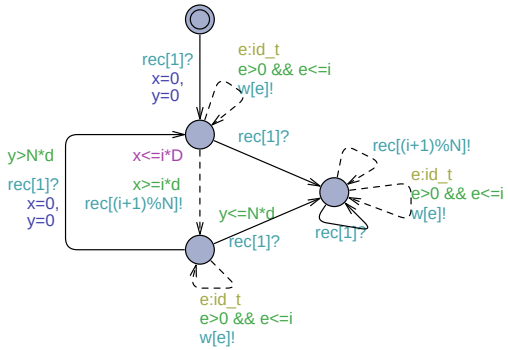

Figure 14: The sub-specification SS_i that abstracts the the sub-system $M_1||\ldots||M_i$.

the system, created as a template, and that thus it is modelled only once and then instantiated with different indices.

Here the sub-specification SS_i, as shown in Fig. 14, is a model for a sequence of nodes $M_1||\ldots||M_i$ (see *Fig.* 11). Informally SS_i is expressed as following, noting that the relevant ports for this subsystem are `rec[1]?`, `w[e]! (0<e<=i)` and `rec[i+1]!`: Under the assumption that a) the time elapsing between two `rec[1]?` is more than $N * d$ time-units and b) there are no two consecutive `rec[1]?` without a `rec[i+1]!`, then it is guaranteed that `rec[i+1]!` will occur within $[i * d, i * D]$ time units from `rec[1]?`.

We have conducted experiments for different values of N, the number of nodes in the ring, and d the minimum time delay before passing on the token. We have fixed the upper time limit for passing the token to 30.

The results of the experiments are shown in Table 1. The table shows the time used to check a given property measured in seconds. For each value of N we have two rows. The top one represents the verification of all the steps in the compositional verification while the bottom row represents the verification of one monolithic property. If the verification took more than 600 seconds we stopped it. We had one instance where ECDAR ran out of memory which is indicated by `om`. The time results that are written in *italics* are the cases in which the compositional verification gave a negative result. In these cases one needs to propose more precise sub-specifications in order to make the compositional verification work. The monolithic method gives positive results in these cases.

In the case where d is close to D there is very little interleaving in the system and the verification of the monolithic property is the fastest. The smaller the d value the more interleaving appears in the system and in these complex cases the compositional verification shows its strength. The cases where the compositional verification beats the monolithic are marked by **boldface**.

11 Application 2: A Parking System

In this example we use real-time specifications in an assume/guarantee approach, to build a system that describes the behavior of a car park. Such a system has been studied in [55], with a top to bottom approach that builds a specification of the system from a list of requirements written in natural language, and then projects these specifications on an architecture of components. We use a different approach that starts with a set of requirements for these components and then builds the formal specifications of these components, which can be composed together in order to build the specification of the system. We also made the example much more realistic by adding timing requirements. This also requires to check global timing properties, which we perform using the ECDAR toolset.

The system is composed of four components: EntryGate, ExitGate, Controller, and Payment. It is parameterized by the maximum number N_{max} of cars that can enter the parking. We will also consider the environment of the system that consists in the car users. However, we adopt an abstract view of the system in which cars are not individualized, but we remember the number of cars that have entered.

The components are defined by the following requirements that describe either guarantees on the outputs of the components or assumptions on the inputs provided by the environment. For each gate:

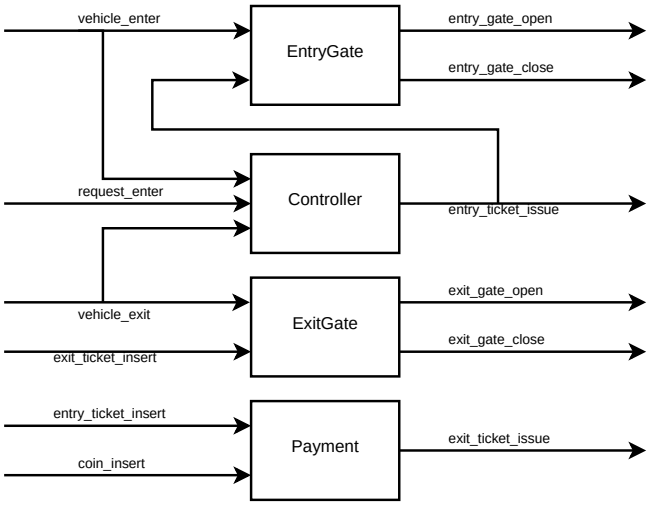

Figure 15: Parking components and communication channels

Req. 1 *A vehicle shall not pass when the gate is closed.*

Req. 2 *Once a vehicle has passed the gate, another vehicle cannot pass before the gate closes.*

Req. 3 *After the gate has opened, it does not open before it closes. After the gate has closed, it does not close before it opens.*

Req. 4 *The gate must close within 5 seconds after a vehicle passes, and only then.*

Specific to EntryGate:

Req. 5 *An entry ticket is issued only when the entry gate is closed.*

Req. 6 *The gate must open within 5 seconds after an entry ticket has been issued, and only then.*

Specific to ExitGate:

Req. 7 *An exit ticket is inserted only when the entry gate is closed.*

Req. 8 *The gate must open within 5 seconds after an exit ticket has been inserted, and only then.*

For Controller:

Req. 9 *A vehicle does not exit when the parking is empty.*

Req. 10 *A vehicle does not enter before receiving an entry ticket.*

Req. 11 *If the parking is not full, an entry ticket is issued within 10 seconds after being requested.*

For Payment:

Req. 12 *A user inserts a coin every time an entry ticket is inserted, and only then.*

Req. 13 *A user may insert an entry ticket only initially or after an exit ticket has been issued.*

Req. 14 *The payment machine issues an exit ticket within 40 seconds once the entry ticket and the coin have been inserted.*

The communications between the components are described in Fig. 15.

Number of vehicles	10	10^2	10^3	10^4	10^5
Consistency	<0.1s	<0.1s	0.4s	4.4s	45.4s
Compatibility	<0.1s	<0.1s	0.2s	1.6s	18s

Table 2: ECDAR performance in analyzing SubSys

11.1 The entry gate subsystem

We begin with the specifications of the two components EntryGate and Controller. It forms a subsystem that has three inputs (`vehicle_enter`, `vehicle_exit`, and `request_enter`) and three outputs (`entry_gate_open`, `entry_gate_close`, and `entry_ticket_issue`). Each requirement is translated into a timed specification. Reqs. 1-2 are assumptions on the EntryGate inputs. They are translated into specifications EnA1 (Fig. 16(a)) and EnA2 (Fig. 16(b)), respectively. Req. 5 is translated into a specification EnA3 similar to EnA1. Conversely, Reqs. 3-4-6 correspond to guarantees on the outputs of EntryGate, and they are translated into specifications EnG1 (Fig. 16(c)), EnG2 (Fig. 16(d)), and EnG3 (Fig. 16(e)), respectively.

The Controller is responsible for the delivery of the entry ticket. We impose an additional requirement on the Controller that should be sufficient to satisfy the assumption in Req. 5:

Req. 15 *Request to enter are ignored for 6 seconds after a vehicle has entered.*

Then, Reqs. 9-10-11-15 defined the specification CtAG, shown in Fig. 16, that encompasses both assumptions and guarantees in the same model, using a universal state to model incompatible inputs.

We check with ECDAR the consistency of this subsystem and the compatibility between its two components. The EntryGate component is defined using the weaken operator between the assumptions and the guarantees:

$$\mathsf{EnA} := (\mathsf{EnA1} \wedge \mathsf{EnA2} \wedge \mathsf{EnA3})$$

$$\mathsf{EnG} := (\mathsf{EnG1} \wedge \mathsf{EnG2} \wedge \mathsf{EnG3})$$

$$\mathsf{EntryGate} := \mathsf{EnG} >> \mathsf{EnA}$$

The subsystem is constructed using the parallel composition.

$$\mathsf{SubSys} := \mathsf{EntryGate} \parallel \mathsf{CtAG}$$

We provide a minimal environment that is build from the assumptions EnA1, EnA2, and the one described in Req. 9, translated into EnvCt1, such that $\mathsf{Env} := \mathsf{EnA1} \wedge \mathsf{EnA2} \wedge \mathsf{EnvCt1}$. We check that $\mathsf{SubSys} \parallel \mathsf{Env}$ is consistent and that no universal state is reached. This proves that the components are compatible, and that the assumptions Req. 5 and Req. 11 are both satisfied by the other component. Benchmarking results are given in Table 2 for different number of vehicles in the car park. They show that these tests scale well.

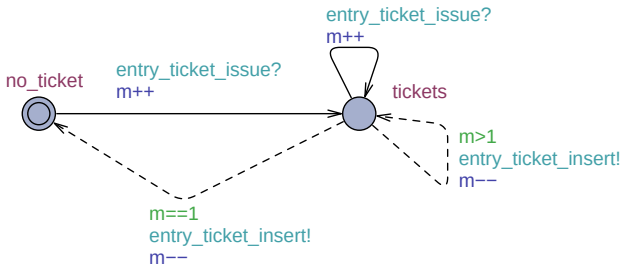

Figure 18: EnvEnTickets: Specification of the environment w.r.t. entry tickets.

11.2 Parking system correctness

We pursue our study by including the components ExitGate and Payment. For ExitGate, Reqs. 1-2-7 yield the specifications of the assumptions ExA1, ExA2, and ExA3, and Reqs. 3-4-8 yield the guarantees ExG1, ExG2, and ExG3, in the same manner as were constructed the ones of EntryGate. For Payment, Reqs. 12-13-14 yield the specifications PayA1 (Fig. 17(a)), PayA2 (Fig. 17(b)), and PayG1 (Fig. 17(c)), respectively. Then the system under study is the following:

$$\mathsf{ExA} := (\mathsf{ExA1} \wedge \mathsf{ExA2} \wedge \mathsf{ExA3})$$

$$\mathsf{ExG} := (\mathsf{ExG1} \wedge \mathsf{ExG2} \wedge \mathsf{ExG3})$$

$$\mathsf{ExitGate} := \mathsf{ExG} >> \mathsf{ExA}$$

$$\mathsf{Payment} := \mathsf{PayG} >> (\mathsf{PayA1} \wedge \mathsf{PayA2})$$

$$\mathsf{Sys} := \mathsf{EntryGate} \parallel \mathsf{ExitGate} \parallel \mathsf{Payment} \parallel \mathsf{CtAG}$$

This system is however underspecified, since no formal relation exists between the tickets that are issued and the ones that are inserted. Therefore we add the following requirements:

Req. 16 *An entry ticket is inserted only if it has been issued before.*

Req. 17 *An exit ticket is inserted only if it has been issued before.*

These yield two specifications, EnvEnTickets in Fig. 18, and similarly EnvExTickets, that are added in conjunction to the environment, along with the assumptions of ExitGate and Payment.

We want to check the correctness of the parking system, expressed by the property that no car can exit without paying. Therefore we design a specification SpecExp (Fig. 19(a)), that increases its revenue expectation e each time a vehicle enter, and decreases it when the payment is received. If all the vehicles exit when the number n of vehicles in the parking is strictly greater than e, that means that the payment has been received previously. We check by refinement that the system satisfy this property:

$$\mathsf{Sys} \parallel \mathsf{Env} \leq \mathsf{SpecExp}$$

Benchmarking results for this property are listed in the first row of Table 3.

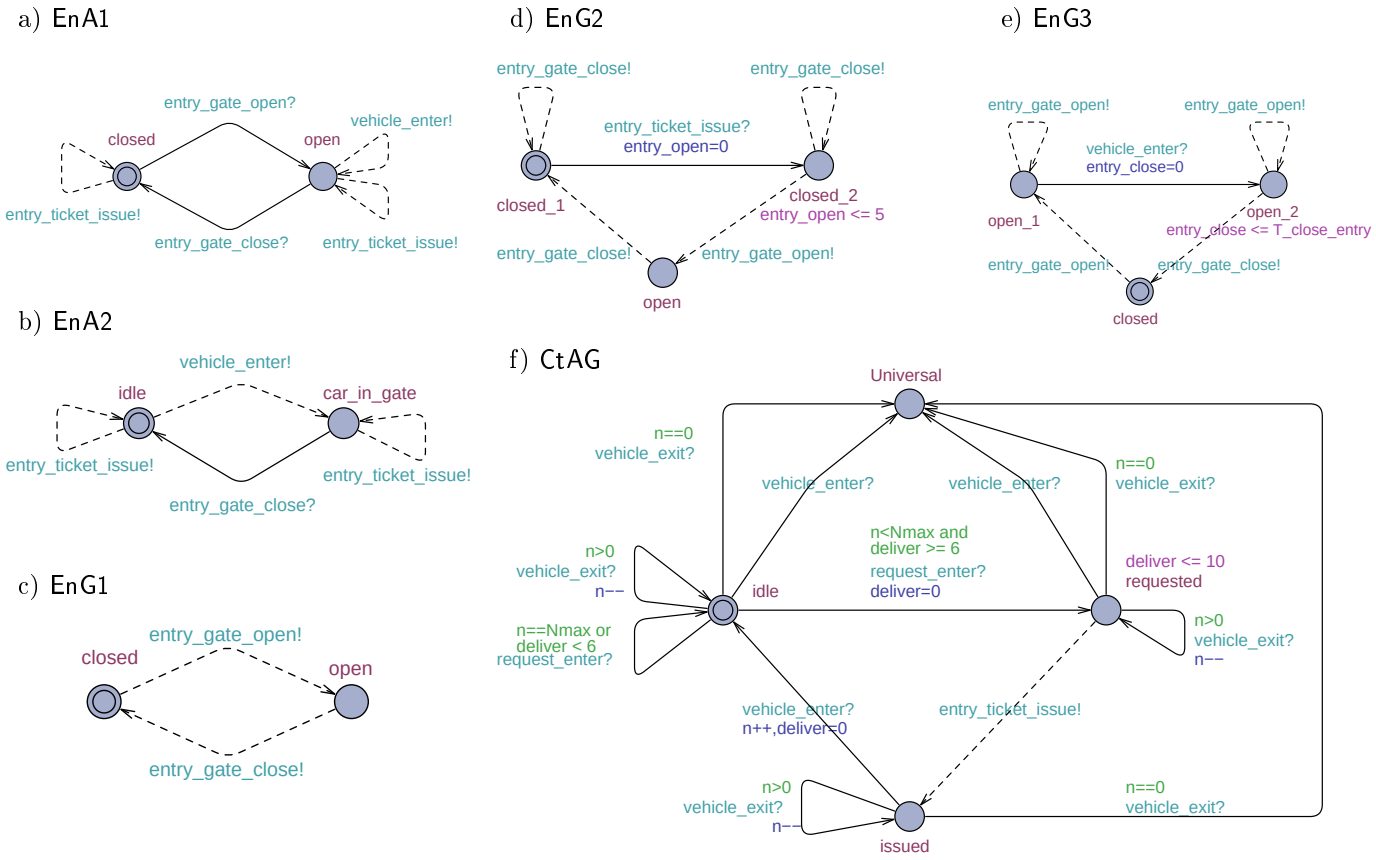

Figure 16: Timed specifications of the entry gate subsystem (all models are input-enabled and therefore assume that self-loops exist for input actions that are not represented).

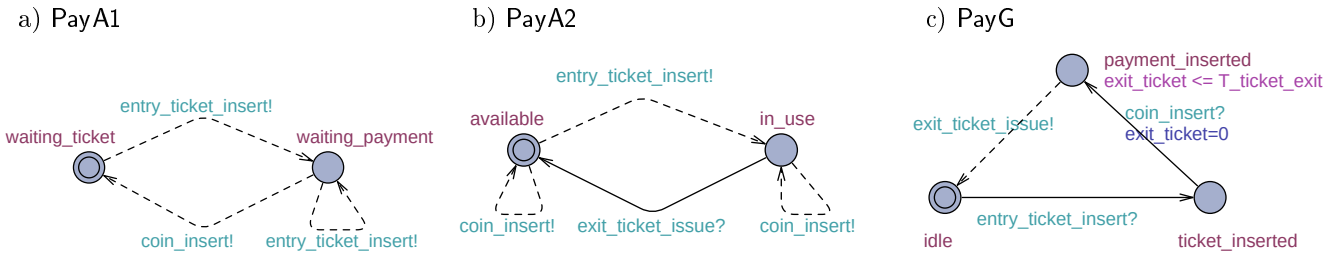

Figure 17: Timed specifications of payment machine (self-loops with input actions are not represented).

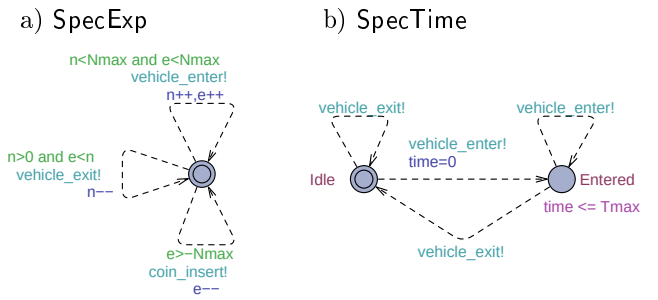

Figure 19: System properties.

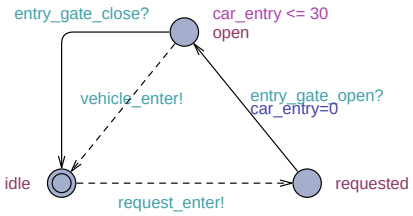

Figure 20: EnvEnCar: Specification of the environment w.r.t. entry cars.

11.3 Timing constraints

In the last part of our study we perform a timing analysis of the system. Inherent timing constraints of the components have already been taken into account in the guarantees (EnG2,EnG3,ExG2,ExG3,CtAG, and PayG). We would like to check a global timing constraint: *the time between a vehicle entering the parking and a vehicle exiting is bounded by some maximum delay.* For this study we need to precisely specify the timing behaviors of the environment, that is to say the vehicle drivers, which lead us to add or modify some requirements:

Req. 18 *A user inserts a coin within 30 seconds every time an entry ticket is inserted, and only then.*

Req. 19 *Once an entry ticket is issued, the user inserts it in the payment machine within 1 hour.*

Req. 20 *Once an exit ticket is issued, the user inserts it at the exit gate within 5 minutes.*

Req. 21 *When a gate opens, a vehicle passes within 30 seconds.*

Consequently, to satisfy Reqs. 18-19-20 we modify the specifications of the environment PayA1, EnvEnTickets, and EnvExTickets. To satisfy Req. 21 we add two additional specifications to the environment, EnvEnCar (displayed in Fig. 20), and similarly EnvExCar, that describe the behavior of the users. We check the compatibility of this new environment that is sufficient to satisfy the assumptions of EntryGate and ExitGate, since no universal state is reached in Sys $\|$ Env.

Finally the timing property is translated into a specification SpecTime displayed in Fig. 19(b). The property is checked with the refinement:

$$\mathsf{Sys} \parallel \mathsf{Env} \leq \mathsf{SpecTime}$$

Number of vehicles	2	4	8	16	32
SpecExp	<0.1s	0.2s	1.5s	11.5s	90s
Compatibility	0.2s	0.6s	3.5s	17.3s	72.5s
SpecTime	0.2s	1.5s	19.5s	94s	327s

Table 3: Ecdar performances in analyzing Sys

We prove that the property is satisfied for $T_{max} = 4100$. Table 3 presents the benchmarking results for the analysis of Sys.

12 Conclusion and future work

This paper presents a complete game-based interface theory for timed systems. Our theory implements all the good operations for a specification theory, namely: consistency, refinement, structural/logical composition, and quotient. Our results have been implemented in the Ecdar toolset that is an extension of the well-established Uppaal model checker. Our tool has been applied to serious size case studies (while most of existing frameworks remain at the theory level).

Our research can be pursued in various directions, one of them being to continue intensive testing of Ecdar and give a complete characterization of problems for which our theory is indeed practically useful. Targeting large size systems will certainly require to improve the efficiency of the algorithms implemented in Ecdar. As an example, we postulate that state-space reduction through bisimulation quotient should considerably improve the pruning algorithm. Still, in the context of Ecdar, developing a user-feedback mechanism is challenging, but needed to broaden our user base.

Another promising direction is the one of robust specification theories. One says that an implementation is robust with respect to a given specification if it remains an implementation of the specification under small perturbations of time. Studying robustness is crucial as it is generally not possible to implement a specification without considering perturbations introduced by the external environment [62] (e.g. hardware constraints). We recently investigated this problem for our timed interfaces for a fixed value of the perturbation [46] and we proposed a technique to evaluate the maximal perturbation under which an implementation remains robust [60]. In the future we want to fully integrate this theory in Ecdar.

We will also investigate the problem of stuttering and hidden actions, which we plan to do via an exploitation of imperfect information games [18].

Finally, it would be worth extending our theory to systems with both timed and stochastic aspects, hence proposing the first specification theory for probabilistic timed automata [43, 44]. In a series of recent work [35, 16], we have proposed specification theories for stochas-

tic systems. We postulate that such specification theories can be combined with our timed interfaces one, just like timed automata have been combined with Markov decision processes.

Acknowledgements. Work partially supported by VKR Centre of Excellence – MT-LAB, the European project COMBEST, and ARC (TP)I.

References

1. Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. In *EMSOFT*, pages 229–238. ACM, 2010.
2. B. Thomas Adler, Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc: A tool for interface compatibility and composition. In *CAV*, volume 4144 of *LNCS*, pages 59–62. Springer, 2006.
3. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
4. Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *CONCUR*, volume 1466 of *LNCS*. Springer, 1998.
5. Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, volume 2791 of *LNCS*, pages 60–72. Springer, 2003.
6. Adam Antonik, Michael Huth, Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal and mixed specifications: key decision problems and their complexities. *Mathematical Structures in Computer Science*, 20(1):75–103, 2010.
7. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.
8. Sebastian S. Bauer, Line Juhl, Kim G. Larsen, Axel Legay, and Jirí Srba. Extending modal transition systems with structured labels. *Mathematical Structures in Computer Science*, 22(4):581–617, 2012.
9. Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In *CAV*, volume 4590 of *LNCS*. Springer, 2007.
10. Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing uppaal over 15 years. *Softw., Pract. Exper.*, 41(2):133–142, 2011.
11. Jasper Berendsen and Frits W. Vaandrager. Compositional abstraction in real-time model checking. In *FORMATS*, volume 5215 of *LNCS*. Springer, 2008.
12. Nathalie Bertrand, Axel Legay, Sophie Pinchinat, and Jean-Baptiste Raclet. A compositional approach on modal specifications for timed systems. In *ICFEM*, LNCS. Springer, 2009.
13. T. Bourke and A. Sowmya. Automatically transforming and relating uppaal models of embedded systems. In *EMSOFT*, pages 59–68. ACM, 2008.
14. Timothy Bourke, Alexandre David, Kim G. Larsen, Axel Legay, Didier Lime, Ulrik Nyman, and Andrzej Wasowski. New results on timed specifications. In *WADT*, volume 7137 of *LNCS*, pages 175–192. Springer, 2010.
15. Peter Bulychev, Thomas Chatain, Alexandre David, and Kim G. Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *FORMATS*, volume 5813 of *LNCS*, pages 73–87. Springer, 2009.
16. Benoit Caillaud, Benoit Delahaye, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, and Andrzej Wąsowski. Compositional design methodology with constraint Markov chains. In *QEST*, pages 123–132. IEEE, 2010.
17. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, 2005.
18. Franck Cassez, Alexandre David, Kim Guldstrand Larsen, Didier Lime, and Jean-François Raskin. Timed control with observation based and stuttering invariant strategies. In *ATVA*, volume 4762 of *LNCS*, pages 192–206. Springer, 2007.
19. Karlis Cerans, Jens Chr. Godskesen, and Kim Guldstrand Larsen. Timed modal specification - theory and tools. In *CAV*, pages 253–267. Springer-Verlag, 1993.
20. Arindam Chakabarti, Luca de Alfaro, Thomas A. Henzinger, and Marielle I. A. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *EMSOFT 03: 3rd Intl. Workshop on Embedded Software*, LNCS. Springer, 2003.
21. Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Synchronous and bidirectional component interfaces. In *CAV*, volume 2404 of *LNCS*, pages 414–427, 2002.
22. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, 1999.
23. Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Methodologies for specification of real-time systems using timed i/o automata. In *FMCO*, volume 6286 of *LNCS*, pages 290–310. Springer, 2009.
24. Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In *HSCC*, pages 91–100. ACM ACM, 2010.
25. Alexandre David, Kim Guldstrand Larsen, Axel Legay, Mikael H. Møller, Ulrik Nyman, Anders P. Ravn, Arne Skou, and Andrzej Wasowski. Compositional verification of real-time systems using ecdar. *STTT*, 14(6):703–720, 2012.
26. Alexandre David, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Ecdar: An environment for compositional design and analysis of real time systems. In *ATVA*, volume 6252 of *LNCS*, pages 365–370. Springer, 2010.
27. Luca de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verification (Theory in Practice)*, volume 2772 of *LNCS*. Springer, 2003.
28. Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable interfaces. In *FroCos*, volume 3717 of *LNCS*, pages 81–105. Springer, 2005.
29. Luca de Alfaro and Marco Faella. An accelerated algorithm for 3-color parity games with an application to

timed games. In *CAV*, volume 4590 of *LNCS*. Springer, 2007.
30. Luca de Alfaro, Marco Faella, Thomas A. Henzinger, Rupak Majumdar, and Mariëlle Stoelinga. The element of surprise in timed games. In *CONCUR*, volume 2761 of *LNCS*, pages 142–156. Springer, 2003.
31. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *FSE*, pages 109–120, Vienna, Austria, September 2001. ACM Press.
32. Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In *In Engineering Theories of Software Intensive Systems, Marktoberdorf Summer School*. Kluwer Academic Publishers, 2004.
33. Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. Symbolic algorithms for infinite-state games. In *CONCUR*, volume 2154 of *LNCS*, pages 536–550. Springer, 2001.
34. Luca de Alfaro, Thomas A. Henzinger, and Marielle I. A. Stoelinga. Timed interfaces. In *EMSOFT*, volume 2491 of *LNCS*, pages 108–122. Springer, 2002.
35. Benoît Delahaye, Joost-Pieter Katoen, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, Falak Sher, and Andrzej Wasowski. Abstract Probabilistic Automata. In *VMCAI*, pages 324–339. Springer, 2011.
36. José Luiz Fiadeiro and Luis Filipe Andrade. Interconnecting objects via contracts. In *Proc. of the 38th International Conference on Technology of Object-Oriented Languages and Systems, Components for Mobile Computing (TOOLS'38)*, pages 182–183. IEEE Computer Society, 2001.
37. José Luiz Fiadeiro and T. S. E. Maibaum. Interconnecting formalisms: Supporting modularity, reuse and incrementality. In *Proc. of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT FSE'95)*, pages 72–80. ACM, 1995.
38. Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 1998.
39. Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *REX Workshop*, volume 600 of *LNCS*, pages 226–251. Springer, 1991.
40. Thomas A. Henzinger and Slobodan Matic. An interface algebra for real-time components. In *IEEE Real Time Technology and Applications Symposium*, pages 253–266. IEEE Computer Society, 2006.
41. Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In *FM*, volume 4085 of *LNCS*, pages 1–15. Springer, 2006.
42. Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. *The Theory of Timed I/O Automata, Second Edition*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.
43. M. Z. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. In *FORMATS*, volume 3253 of *LNCS*, pages 293–308. Springer, 2004.
44. M. Z. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. *Inf. Comput.*, 205(7):1027–1077, 2007.
45. Kim G. Larsen. Modal specifications. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.
46. Kim G. Larsen, Axel Legay, Louis-Marie Traonouez, and Andrzej Wasowski. Robust specification of real time components. In *FORMATS 2011*, volume 6919 of *LNCS*. Springer, 2011.
47. Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.
48. Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, volume 965 of *LNCS*, pages 62–88, August 1995.
49. Nancy Lynch. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38, Princeton University, Princeton, N.J., 1988.
50. Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, The MIT Press, November 1988.
51. Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.
52. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1988.
53. Rocco De Nicola and Roberto Segala. A process algebraic view of input/output automata. *Theoretical Computer Science*, 138, 1995.
54. Amalinda Post, Jochen Hoenicke, and Andreas Podelski. rt-inconsistency: A new property for real-time requirements. In *FASE*, volume 6603 of *LNCS*, pages 34–49. Springer, 2011.
55. J.-B. Raclet, B. Caillaud, D. Nickovic, R. Passerone, A. Sangiovanni-Vincentelli, T. Henzinger, and K. G. Larsen. Contracts for the design of embedded systems part i: Methodology and use cases. Technical report. Submitted, `http://www.irisa.fr/distribcom/benveniste/pub/ProcIEEE_contractsPart1.pdf`.
56. Eugene W. Stark, Rance Cleavland, and Scott A. Smolka. A process-algebraic language for probabilistic I/O automata. In *CONCUR*, LNCS, pages 189–2003. Springer, 2003.
57. Jun Sun, Yang Liu, and Jin Song Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008.
58. Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Andre Etienne. Modeling and verifying hierarchical real-time systems using stateful timed csp. *ACM Trans. Softw. Eng. Methodol.*, 2012. Accepted.
59. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
60. Louis-Marie Traonouez. A parametric counterexample refinement approach for robust timed specifications. In *FIT*, volume 87 of *EPTCS*, pages 17–33, 2012.
61. Frits W. Vaandrager. On the relationship between process algebra and input/output automata. In *LICS*, pages 387–398, 1991.

62. Martin Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. Robust safety of timed automata. *Formal Methods in System Design*, 33:45–84, December 2008.
63. Wang Yi. Real-time behaviour of asynchronous agents. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR*, volume 458 of *LNCS*, pages 502–520. Springer, 1990.