Timed I/O Automata: A Complete Specification Theory for Real-time Systems

Alexandre David
Computer Science
Aalborg University, Denmark
adavid@cs.aau.dk

Kim G. Larsen
Computer Science
Aalborg University, Denmark
kgl@cs.aau.dk

Axel Legay
INRIA/IRISA
Rennes Cedex, France
axel.legay@irisa.fr

Ulrik Nyman
Computer Science
Aalborg University, Denmark
ulrik@cs.aau.dk

Andrzej Wąsowski
IT University
Copenhagen, Denmark
wasowski@itu.dk

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 complete 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 on top of an engine for timed games, UPPAAL-TIGA, and illustrated with a small case study.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]

General Terms

Theory, 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* [20], the mathematical foundations of reasoning about interfaces, is an active research area. It supports inferring properties of the global implementation, or designing and advisedly reusing components.

HSCC'10, April 12–15, 2010, Stockholm, Sweden.

In a logical interpretation, interfaces are specifications and components that implement an interface are understood as models/implementations. Specification theories should support various features including (1) *refinement*, which allows 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 to combine specifications, and (4) last but not least, a *quotient operator* that is dual to structural composition. The latter is crucial to perform incremental design. Also, the operations have to be related by compositional reasoning theorems, guaranteeing both incremental design and independent implementability [13].

Building good specification theories is the subject of intensive studies [10, 12]. One successfully promoted direction is the one of interface automata [12, 13, 22, 28]. In this framework, an interface is represented by an input/output automaton [26], *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 [26], 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 [15], 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 [15] focuses mostly on structural composition, in this paper we go one step further and build what we claim to be the first game-based specification theory for timed systems.

Component Interface specification and consistency. We represent specifications by timed input/output transition systems [21], i.e., timed transitions systems whose sets of discrete transitions are split into Input and Output transitions. Contrary to [15] and [21] 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.

Refinement and logical conjunction. 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. In the input/output setting, checking refinement reduces to deciding an alternating timed simulation between the two specifications [12]. In our timed extension, checking such simulation can be done with a slight modification of the theory proposed in [6]. 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. 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.

Structural composition. Following [15], specifications interact by synchronizing on inputs and outputs. However, like in [21, 26], 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. Unlike in [21, 26], input-enabledness, shall not be seen as a way to avoid error states. Indeed, such error states can be designated by the designer as states which do not warrant desirable temporal properties. 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.

Quotient. We propose a quotient operator dual to composition. 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.

Implementation. Our methodology has been implemented as an extension of UPPAAL-TIGA [3]. It builds on timed input/output automata, a symbolic representation for timed input/output transition systems. We show that conjunction, composition, and quotienting are 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 [27, 8]. Finally, refinement between specifications is checked using a variant of the recent efficient game-based algorithm of [6].

Example. Universities operate under increasing pressure and competition. One of the popular factors used in determining the level of national funding is that of societal impact, which is approximated by the number of patent applications filed. Clearly one would expect that the number (and size) of grants given to a university has a (positive) influence on the amount of patents filed for.

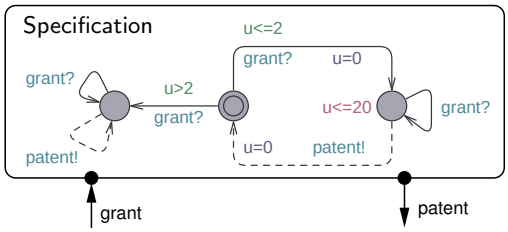

Figure 1: Overall specification for a University.

Figure 2 gives the insight as to the organisation of a very small University comprising three components Administration, Machine and Researcher. The Adminsitration is responsible for interaction with society in terms of aquiring grants (grant) and filing patents (patent). However, the other components are necessary for patents to be obtained. The Researcher will produce the crucial publications (pub) within given time intervals, provided timely stimuli in terms of coffee (cof) or tea (tea). Here coffee is clearly prefered over tea. The beverage is provided by a Machine, which given a coin (coin) will provide either coffee or tea within some time interval, or even the possibility of free tea after some time.

In Figure 2 the three components are specifications, each allowing for a multitude of incomparable, actual implementations differing with respect to exact timing behavior (*e.g.* at what time are publications actually produced by the Researcher given a coffee) and exact output produced (*e.g.* does the Machine offer tea or coffee given a coin).

As a first property, we may want to check that the composition of the three components comprising our University is compatible: we notice that the specification of the Researcher contains an Err state, essentially not providing any guarantees as to what behaviour to expect if tea is offered at a late stage. Now, compatibility checking amounts simply to deciding whether the user of the University (*i.e.* the society) has such a strategy for using it that the Researcher will avoid ever entering this error state.

As a second property, we may want to show that the composition of arbitrary implementations conforming to respective component specification is guaranteed to satisfy some overall specification. Here Figure 1 provides an overall specification (essentially saying that whenever grants are given to the University sufficiently often then patents are also guaranteed within a certain upper time-bound). To avoid clutter, we have omitted looping guard free output edges for the three actions (coin, cof and tea) which are present in all the states. Checking this property amounts to establishing a refinement between the composition of the three component specifications and the overall specification. We leave the reader in suspense until the concluding section before we reveal whether the refinement actually holds or not!

2. SPECIFICATIONS & REFINEMENT

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) [18], and enriched with syntactic constructions for *Timed I/O Automata* (TIOAs), which act as a symbolic and finite representation for TIOTSs.

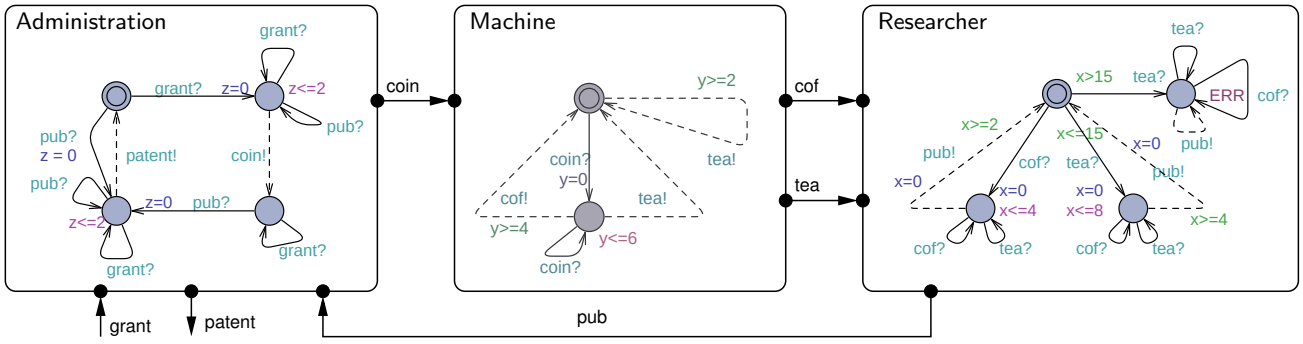

Figure 2: Specifications for and interconnections between the three main components of a modern University: Administration, Machine and Researcher.

However, it is important to emphasize that 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 do not know how to manipulate them automatically).

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, $\Sigma^S = \Sigma_{\mathrm{i}}^S \oplus \Sigma_{\mathrm{o}}^S$ is a finite set of actions partitioned into inputs (Σ_{i}^S) and outputs (Σ_{o}^S) and $\rightarrow^S \subseteq St^S \times (\Sigma^S \cup \mathcal{R}_{\geq 0}) \times St^S$ is a transition relation. We write $s \xrightarrow{a}{}^S s'$ instead of $(s, a, s') \in \rightarrow^S$ and use $i?$, $o!$ and d to range over inputs, outputs and $\mathcal{R}_{\geq 0}$ respectively. In addition any TIOTS satisfies the following:*

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

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

[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 $s' \in St^S$

In the interest of simplicity, we work with *deterministic* TIOTSs: 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'.

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

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

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

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

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

We shall now introduce a finite 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 $u \in [Clk \mapsto \mathcal{R}_{\geq 0}]$. We write $u + d$ to denote a valuation such that for any clock r we have $(u+d)(r) = x+d$ iff $u(r) = x$. Given $d \in \mathcal{R}_{\geq 0}$, 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 will 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. 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 $Inv : Loc \mapsto \mathcal{B}(C)$ 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. Examples of TIOAs have been proposed 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:

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

The TIOTSs induced by TIOAs satisfy the axioms 1–3 of Definition 1. In order to guarantee determinism, the TIOA has to be deterministic: for each action–location pair only one transition can be enabled at the same time. This is a standard check. We assume that all TIOAs below are deterministic.

Having introduced a syntactic representation for TIOTSs, we now turn back to the semantic level in order to define the basic concepts of implementation and specification:

Definition 3. *A TIOTS S is a* specification *if each of its states $s \in St^S$ is input-enabled: $\forall\, i? \in \Sigma_{\mathrm{i}}^S.\, \exists\, s' \in St^S.\, s \xrightarrow{i?}{}^S s'$.*

The assumption of input-enabledness, also seen in many interface theories [25, 17, 30, 33, 29], 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. Input-enbledness encourages explicit modeling of this unpredictability, and compositional reasoning about it; for example, deciding if an unpredictable behaviour of one component induces unpredictability of the entire system.

It is easy to check if a TIOA induces an input-enabled TIOTS. Thus we define Timed Input/Output Specification Automata (*Specification Automata* in short) to be TIOAs, whose semantic TIOTS is a specification. In practice tools can interpret absent input transitions in at least two reasonable ways. First, they can be interpreted as ignored inputs, corresponding to location loops in the automaton. Second, they may be seen as unavailable ('blocking') inputs, which can be achieved by assuming implicit transitions to a designated error state. Later, in Section 4 we will call such a state *strictly undesirable* and give a rationale for this name.

The role of specifications in a specification theory is to abstract, or underspecify, sets of possible implementations. *Implementations* are concrete executable ralizations of systems. 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. *An implementation $P = (St^P, p_0, \Sigma^P, \rightarrow^P)$ is a specification such that for each state $p \in St^P$ we 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 consequently, due to determinism $p = p''$)

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

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 progress. 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.

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 environement and obtain an equivalent system.

We study these kind of properties in later sections. It is well known from the literature [12, 13, 6] that in order to give these kind of guarantees a refinement should have the flavour of *alternating (timed) simulation* [2].

Definition 5. *A specification $S = (St^S, s_0, \Sigma, \rightarrow^S)$ refines a specification $T = (St^T, t_0, \Sigma, \rightarrow^T)$, 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 $[\![A_1]\!]_{\mathrm{sem}} \leq [\![A_2]\!]_{\mathrm{sem}}$.

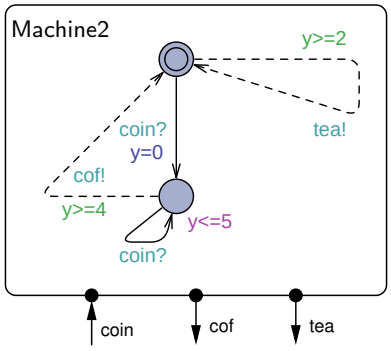

Figure 3: A coffee machine specification that refines the coffee machine in Figure 2.

It is easy to see that the refinement is reflexive and transitive, so it is a preorder on the set of all specifications. 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 6. Figure 3 shows a coffee machine that is a refinement of the one in Figure 2. 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 6. *An implementation P satisfies a specification S, denoted P sat S iff $P \leq S$. We write $[\![S]\!]_{\mathrm{mod}}$ for the set of all implementations of S, so $[\![S]\!]_{\mathrm{mod}} = \{P \mid P \text{ sat } S\}$.*

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 7. *A specification S is consistent, if there exists an implementation P such that P sat S.*

All specification automata in Figure 2 are consistent. An example of an inconsistent specification can be found in Figure 4. Notice that the invariant in the second state ($x \leq 4$) is stronger than the guard ($x \geq 5$) on the cof edge.

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

Definition 8. *A specification S is* locally consistent*, iff every state $s \in St^S$ allows independent progress.*

Theorem 1. *Every locally consistent specification is consistent in the sense of Definition 7.*

The opposite implication in the theorem does not hold as we shall see later. Local consistency, or independent

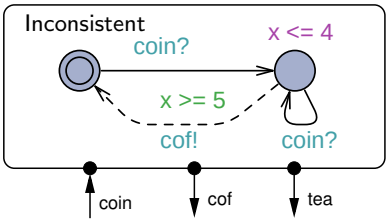

Figure 4: An inconsistent specification.

progress, can be checked for specification automata establishing local consistency for the syntactic representation. Technically it suffices to check for each location that if the supremum of all solutions of every location invariant exists then it satisfies the invariant itself and allows at least one enabled output transition.

Prior specification theories for discrete time [22] and probabilistic [7] 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 most 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 sat P for any implementation and thus each implementation is consistent as per Definition 7. Furthermore each implementation cannot be refined anymore by any underspecified specifications:

THEOREM 2. *Any locally consistent specificiation S refining an implementation P is an implementation as per Def. 4.*

Just like the specifications, we represent implementations syntactically using Timed I/O Automata. A TIOA P is an *implementation automaton* if its underlying TIOTS $[\![P]\!]_{\text{sem}}$ is an implementation. To decide whether a specification automaton is an implementation automaton, one checks for output urgency and independent progress.

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. 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 any preorder in general, turns out to hold for our refinement:

THEOREM 3. *For any two locally consistent specifications S, T we have that $S \leq T$ iff $[\![S]\!]_{\text{mod}} \subseteq [\![T]\!]_{\text{mod}}$.*

The restriction of the theorem to locally consistent specifications is not a serious one. As we shall see, any consistent specification can be transformed into a locally consistent one preserving the set of implementations.

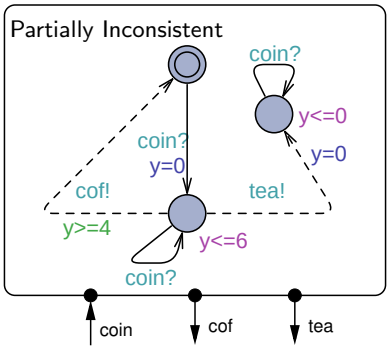

Figure 5: A partially inconsistent specification.

3. CONSISTENCY AND CONJUNCTION

An *immediate error* occurs in a state of a specification if the specification disallows progress of time and output transitions in a given state – such a specification will break if the environment does not send an input. For a specification 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. 5 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, who 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. In order to be able to build on existing formalizations [8] we will consider a dual reachability game, asking for a strategy of the input player to reach err^S. We first define a timed predecessor operator [14, 27, 8], which gives all the states that can delay into X while avoiding Y:

$$\mathsf{cPred}_{\mathrm{t}}^S(X,Y) = \{ s \in St^S \mid \exists d_0 \in \mathcal{R}_{\geq 0}.\, \exists s' \in X. \\ s \xrightarrow{d_0}{}^S s' \text{ and } \mathsf{post}^S_{[0,d_0]}(s) \subseteq \overline{Y} \} \quad (5)$$

Now the *controllable predecessors* operator is:

$$\pi(X) = \mathsf{err}^S \cup \mathsf{cPred}_{\mathrm{t}}^S(X \cup \mathsf{ipred}^S(X), \mathsf{opred}^S(\overline{X})) \quad (6)$$

and the set of all inconsistent states $\overline{\mathsf{cons}^S}$ (i.e. the states for which the environment has a winning strategy) is defined as the least fixpoint of π: $\overline{\mathsf{cons}^S} = \pi(\overline{\mathsf{cons}^S})$, which is guaranteed to exist by monotonicity of π and completeness of the powerset lattice due to the theorem of Knaster and Tarski [31]. For transitions systems enjoying finite symbolic representations, automata specifications included, the fixpoint computation converges after a finite number of iterations [8].

Now we define the set of consistent states, cons^S, simply as the complement of $\overline{\mathsf{cons}^S}$. We obtain it by complementing the result of the above fixpoint computation for $\overline{\mathsf{cons}^S}$.

For the purpose of proofs it is more convenient to formalize the dual operator, say θ, whose *greatest* fixpoint directly and equivalently characterizes cons^S. While least fixpoints are convenient for implementation of on-the-fly algorithms, characterizations with greatest fixpoint are useful in proofs as they allow use of coinduction. Unlike induction on the number of iterations, coinduction is a sound proof principle without assuming finite symbolic representation for the transition system (and thus finite convergence of the fixpoint computation). We avoid discussing θ in details here, since it is a mere technicality.

THEOREM 4. *A specification $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ is consistent iff $s_0 \in \mathsf{cons}^S$.*

The set of (in)consistent states can be computed for timed games, and thus for specification automata, using controller synthesis algorithms [8]. We discuss it briefly in Section 6.

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.

THEOREM 5. *For a consistent specification $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ and $S^\Delta = (\mathsf{cons}^S, s_0, \Sigma^S, \rightarrow^{S^\Delta})$, where $\rightarrow^{S^\Delta} = \rightarrow^S \cap (\mathsf{cons}^S \times (\Sigma^S \cup \mathcal{R}_{\geq 0}) \times \mathsf{cons}^S)$, S^Δ is locally consistent and $[\![S]\!]_{\mathrm{mod}} = [\![S^\Delta]\!]_{\mathrm{mod}}$.*

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

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.

We define a product first. Let $S = (St^S, s_0^S, \Sigma, \rightarrow^S)$ and $T = (St^T, s_0^T, \Sigma, \rightarrow^T)$ be two specifications. A product of S and T, written $S \times T$, is defined to be a specification $(St^S \times St^T, (s_0^S, s_0^T), \Sigma, \rightarrow)$, where the transition relation $\rightarrow$ is the largest relation generated by the following rule:

$$\frac{s \xrightarrow{a}{}^S s' \qquad t \xrightarrow{a}{}^T t' \qquad a \in \Sigma \cup \mathcal{R}_{\geq 0}}{(s,t) \xrightarrow{a} (s',t')} \tag{7}$$

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

Clearly conjunction is commutative. Associativity of conjunction follows from associativity of the product, and the fact that pruning does not remove any implementations (Theorem 5). Conjunction is also the greatest lower bound for lcoally consistent specifications with respect to refinement:

THEOREM 6. *For any locally consistent specifications 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. *$[\![(S \wedge T) \wedge U]\!]_{\mathrm{mod}} = [\![S \wedge (T \wedge U)]\!]_{\mathrm{mod}}$*

We turn our attention to syntactic representations again. 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 $Act_i^1 = Act_i^2$ 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 $(q_2, a, \varphi_2, c_2, q_2') \in E_2$ this gives rise to $((q_1, q_2), a, \varphi_1 \wedge \varphi_2, c_1 \cup c_2, (q_1', q_2')) \in E$

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 7. *Let A_1 and A_2 be two specifiation automata, we have $[\![A_1]\!]_{\mathrm{sem}} \wedge [\![A_2]\!]_{\mathrm{sem}} = [\![A_1 \wedge A_2]\!]_{\mathrm{sem}}$.*

4. COMPATIBILITY & COMPOSITION

We shall now define *structural composition*, also called *parallel composition*, between specifications. We follow the optimistic approach of [15], 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 [21], 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 is possibly in power to avoid it. Thus, following [12], 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 specifications $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ and $T = (St^T, s_0^T, \Sigma^T, \rightarrow^T)$ and we say that they are *composable* iff their output alphabets are disjoint $\Sigma_o^S \cap \Sigma_o^T = \emptyset$.

As we did for conjunction, before defining the parallel composition we first introduce a suitable notion of *product*. The *parallel product* of S and T, which roughly corresponds to the one defined on timed input/output automata [21], 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: $\Sigma_i^{S|T} = (\Sigma_i^S \setminus \Sigma_o^T) \cup (\Sigma_i^T \setminus \Sigma_o^S)$, $\Sigma_o^{S\otimes T} = \Sigma_o^S \cup \Sigma_o^T$.

The transition relation of the product is the largest relation generated by the following rules:

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

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

$$\frac{s \xrightarrow{a}{}^S s' \qquad t \xrightarrow{a}{}^T t' \qquad a \in \Sigma_i^{S|T}}{(s,t) \xrightarrow{a}{}^{S|T} (s',t')}\ [\mathsf{sync\text{-}in}]$$

$$\frac{s \xrightarrow{d}{}^S s' \qquad t \xrightarrow{d}{}^T t' \qquad d \in \mathcal{R}_{\geq 0}}{(s,t) \xrightarrow{d}{}^{S|T} (s',t')}\ [\mathsf{delay}]$$

$$\frac{s \xrightarrow{a}{}^S s' \qquad t \xrightarrow{a}{}^T t' \qquad a \in (\Sigma_i^S \cap \Sigma_o^T) \cup (\Sigma_o^S \cap \Sigma_i^S)}{(s,t) \xrightarrow{a}{}^{S|T} (s',t')}\ [\mathsf{sync\text{-}io}]$$

Observe that if we compose to 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 [15], our specifications are input-enabled, and there is no way to define an error state in where a component can issue an output that cannot be captured by the other component.

The absence of "model-related" error states allows us to define more elaborated errors, specified by the designer. Those cannot easily be considered in [15]. We now give more details. 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, to signal unpredictability of the behaviour after receiving an unanticipated input). Second, a temporal property written in some logic such as TCTL [1] can be interpreted over our specification, which when analyzed by a model checker, will result in partitioning 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 alway 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 a part of specification definition.

We will say that a specification is *useful* if there exists an environment E that can avoid reaching a strictly undesirable state whathever the specification will do. The environment E is said to be *compatible* with S. We are now ready to define structural composition.

We now propose to compute the set of useful states of S using a fixpoint characterisation. We consider a variant of controllable time predecessor operator, where the roles of the inputs and outputs are reversed:

$$\omega(X) = \mathsf{undesirable}^S \cup \mathsf{cPred_t}(X \cup \mathsf{ipred}(X), \mathsf{opred}(\overline{X})) \quad (8)$$

Now the set of useless states $\overline{\mathsf{useful}^S}$ can be characterized as the least fixpoint of ω, so $\overline{\mathsf{useful}^S} \supseteq \omega(\mathsf{useful}^S)$. Again existance and uniqueness of this fixpoint is warrented by monotonicity of ω. The set of useful states is defined as the complement: $\mathsf{useful}^S = \overline{\overline{\mathsf{useful}^S}}$.

THEOREM 8. *For a consistent specification S, S is useful iff $s_0 \in \mathsf{useful}^S$.*

The following theorem shows that pruning the specification does not change the set of compatible environment.

THEOREM 9. *If S such that $s_0 \in \mathsf{useful}^S$ and $S^\beta = (\mathsf{useful}^S \cup \{u\}, s_0, \Sigma^S, \rightarrow^{S^\beta}, \{u\})$. Then E is compatible with S iff E compatible with S^β.*

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 [12].

DEFINITION 10. *Two composable specifications S and T are compatible iff the initial state of $S \otimes T$ is useful.*

DEFINITION 11. *For two compatible specifications S and T define their parallel composition $S|T = (S \otimes T)^\beta$, 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, $\mathsf{undesirable}^{S|T}$, can be increased by 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:

$$[\![(S|T)|U]\!]_{\mathrm{mod}} = [\![S|(T|U)]\!]_{\mathrm{mod}} \quad (9)$$

THEOREM 10. *Refinement is a pre-congruence with respect to parallel composition; for any specifications 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 $S_1|T \leq S_2|T$. Moreover if S_2 compatible with T then S_1 compatible with T.*

We now switch to the symbolic representation. Parallel composition of two TIOA 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 $A_1|A_2$ 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 $q_2 \in Loc_2$ this gives $((q_1, q_2), a, \varphi_1, c_1, (q_1', q_2)) \in E$
- If $(q_2, a, \varphi_2, c_2, q_2') \in E_2$ with $a \in Act_2 \setminus Act_1$ then for each $q_1 \in Loc_1$ this gives $((q_1, q_2), a, \varphi_1, c_1, (q_1, q_2')) \in E$
- 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 $a \in Act_1 \cap Act_2$ this gives rise to $((q_1, q_2), a, \varphi_1 \wedge \varphi_2, c_1 \cup c_2, (q_1', q_2')) \in E$

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

THEOREM 11. *Let A_1 and A_2 be two specification automata, we have $[\![A_1]\!]_{\text{sem}} \mid [\![A_2]\!]_{\text{sem}} = [\![A_1||A_2]\!]_{\text{sem}}$.*

5. QUOTIENT

An essential operator in a complete specification theory is the one of *quotienting*. It 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 exactly those components 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 have the following requirements on the sets of inputs and outputs of the dividend T and the divisor S when applying quotienting: $\Sigma_{\text{i}}^S \subseteq \Sigma_{\text{i}}^T$ and $\Sigma_{\text{o}}^S \subseteq \Sigma_{\text{o}}^T$.

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

Given two specifications $S = (St^S, s_0^S, \Sigma^S, \rightarrow^S)$ and $T = (St^T, s_0^T, \Sigma^T, \rightarrow^T)$ the pre-quotient is a specification $T \nwarrow S = (St, (s_0, t_0), \Sigma, \rightarrow)$, 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 $\rightarrow^{T \nwarrow S}$ is the largest relation generated by the following rules:

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

$$\frac{s \not\xrightarrow{a}{}^S \quad a \in \Sigma_{\text{o}}^S \cup \mathcal{R}_{\geq 0}}{(t,s) \xrightarrow{a}{}^{T \nwarrow S} \mathsf{u}}\,[\mathsf{unreachable}]$$

$$\frac{t \not\xrightarrow{a}{}^T \quad s \xrightarrow{a}{}^S s' \quad a \in \Sigma_{\text{o}}^S}{(t,s) \xrightarrow{a}{}^{T \nwarrow S} \mathsf{e}}\,[\mathsf{unsafe}]$$

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

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

Theorem 12 states that the proposed pre-quotient operator has exactly the property that it is dual of structural composition with regards to refinement.

THEOREM 12. *For any two specifications S and T such that the pre-quotient $T \nwarrow S$ is defined, and for any implementation X over the same alphabet as $T \nwarrow S$ we have that $S|X$ is defined and $S|X \leq T$ iff $X \leq T \nwarrow S$.*

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 prequotient $T \nwarrow S$ 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 3. Effectively we define the quotient to be $T\backslash\backslash S = (T \nwarrow S)^{\Delta}$. It follows from Theorem 5 that Theorem 12 also holds for the actual quotient operator $\backslash\backslash$ (as opposed to the prequotient).

Quotienting for TIOA is defined in the following way. Consider two TIOA $A_T = (Loc_T, q_0^T, Clk_T, E_T, Act_T, Inv_T)$ 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:

- For each $q_T \in Loc_T, q_S \in Loc_S$ and $a \in Act$ this gives $((q_T, q_S), a, \neg Inv_S(q_S), \{x_{new}\}, l_u) \in E$.
- For each $q_T \in Loc_T, q_S \in Loc_S$ this gives $((q_T, q_S), i_{new}, \neg Inv_T(q_T) \wedge Inv_S(q_S), \{x_{new}\}, l_\emptyset) \in E$.
- If $(q_T, a, \varphi_T, c_T, q_T') \in E_T$ and $(q_S, a, \varphi_S, c_S, q_S') \in E_S$ this gives $((q_T, q_S), a, \varphi_T \wedge \varphi_S, c_T \cup c_S, (q_T', q_S')) \in E$
- For each $(q_S, a, \varphi_S, c_S, q_S') \in E_S$ with $a \in Act_o^S$ this gives rise to $((q_T, q_S), a, \varphi_S \wedge \neg G_T, \{x_{new}\}, l_\emptyset)$ where $G_T = \bigvee\{\varphi_T \mid (q_T, a, \varphi_T, c_T, q_T')\}$
- For each $(q_T, a, \varphi_T, c_T, q_T') \in E_T$ and $a \notin Act_S$ this gives $((q_T, q_S), a, \varphi_T, c_T, (q_T', q_S)) \in E$
- For each $(q_T, a, \varphi_T, c_T, q_T') \in E_T$ with $a \in Act_o^S$ this gives rise to $((q_T, q_S), a, \neg G_S, \{\}, l_u)$ where $G_S = \bigvee\{\varphi_S \mid (q_S, a, \varphi_S, c_S, q_S')\}$
- For each $a \in Act_i$ this gives rise to $(l_\emptyset, a, x_{new} = 0, \{\}, l_\emptyset)$
- For each $a \in Act$ this gives rise to $(l_u, a, true, \{\}, l_u)$

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

THEOREM 13. *Let A_1 and A_2 be two specification automata, we have $([\![A_1]\!]_{\text{sem}} \nwarrow [\![A_2]\!]_{\text{sem}})^{\Delta} = ([\![A_1 \nwarrow A_2]\!]_{\text{sem}})^{\Delta}$.*

6. TOOL IMPLEMENTATION

Our specification theory has been implemented using the verification engine for timed games UPPAAL-TIGA.

Application of The Engine. The engine of UPPAAL-TIGA supports the computation of winning strategies for timed

games with respect to a large class of TCTL winning objectives. The basic algorithm applied is that of [8] being an on-the-fly algorithm performing forward exploration or reachable states and back-propagation of (so-far) computed winning states in an interleaved manner.

Crucial to the algorithm of [8] is the symbolic representation and efficient manipulation of state-sets using *zones*, *i.e.* sets of clock valuations characterized by constraints on individual clocks and clock-differences. In particular the operators cPred_t, ipred and opred used in the fixpoint characterization of consistent and compatible states may be effectively computed using *federations* (unions of zones).

The operations of *conjunction* and *quotienting* may produce specification with *inconsistent* states, which needs to be determined and subsequently pruned away. For this, we apply the algorithm of UPPAAL-TIGA to a timed reachability game, 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 additional effort in implementing this algorithm required an on-the-fly detection of such states (instead of using ordinary state predicates) and back-propagating these according to the algorithm of UPPAAL-TIGA.

Dually, for the *composition* of component specifications with designated error-states, we want to check for *composability*. For this we apply UPPAAL-TIGA to a timed safety game, where input transitions are controllable, output transitions uncontrollable, and where error states in any of the two (or more) components should be avoided.

Finally, the problem of checking the refinement $S \leq T$ is solved as a turn-based game between two players. The first player, or attacker, plays outputs on S and inputs on T, whereas the second player, or 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. In this implementation we can specialize our algorithm to simplify the operator cPred_t analogously to [6] with inverted rules wth respect to controllable and uncontrollable transitions. This has been implemented and the tool has been extended to handle open systems.

Using the Tool. Returning to the example of the University of Figure 2, we may apply the extended version of UPPAAL-TIGA for checking the desired properties of compatibility and consistency. Checking for *compatibility* simply amounts to checking for controllability of the following safety property:

```
control: A[] not Researcher.ERR
```

Checking for *refinement* between the University and the stated overall specification Specification is checked using the following new type of refinement property:

```
refinement: {Administration,Machine,Researcher}
              \{coin,tea,cof,pub} <= {Spec}
```

We are now able to reveal that, unexpectedly, this refinement does *not* hold! In fact the Administration's ability to take in publications and produce patents without a preceeding grant, combined with the Machine's ability to produce

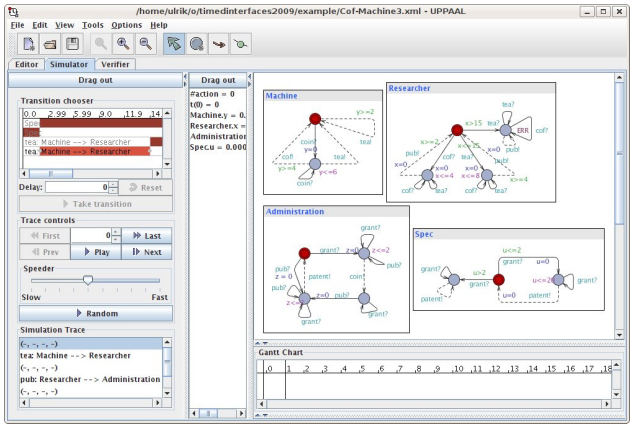

Figure 6: A refinement counter-strategy in Uppaal

tea without any coin, violates the overall Specification's requirement that patents can only be produced given a preceeding grant. In Figure 6 a screen-shot of the extension of UPPAAL-TIGA playing the counter-strategy against a (disbelieving but to be convinced) user is shown.

7. CONCLUDING REMARKS

We have proposed a complete game-based specification theory for timed systems, in which we distinguish between a component and the environment in which it is used. To the best of our knowledge, our contribution is the first game-based approach to support both refinement, consistency checking, logical and structural composition, and quotient. Our results have been implemented in the UPPAAL toolset [3].

There have been several other attempts to propose an interface theory for timed systems (see [15, 11, 5, 4, 9, 32, 16, 24] for some examples). Our model shall definitely be viewed as an extension of the timed input/output automaton model proposed by Lynch et al. [21]. The majors differences are in the game-based treatment of interactions and the addition of quotient and conjunction operators.

In [15], de Alfaro et al. suggested *timed interfaces*, a model that is similar to the one of TIOTSs. Our definition of composition builds on the one proposed in there. However, the work in [15] is incomplete. Indeed there is no notion of implementation and refinement. Moreover, conjunction and quotient are not studied. Finally, the theory has only been implemented in a prototype tool [11] which does not handle continuous time, while our contribution takes advantages of the powerful game engine of UPPAAL-TIGA.

In [22] 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 [5, 4] a timed extension of *modal* automata was proposed, which embeds all the operations presented in the present paper. However, modalities are orthogonal to inputs and outputs, and it is well-known [23] that, contrary to the game-semantic approach, they cannot be used to distinguish between the behaviors of the component and those of the environment.

One could also investigate whether our approach can be

used to perform scheduling of timed systems (see [11, 19, 16] for examples). For example, the quotient operation could perhaps be used to synthesize a scheduler for such problem.

8. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[2] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *CONCUR'98*, volume 1466 of *LNCS*. Springer, 1998.

[3] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *CAV*, volume 4590 of *LNCS*. Springer, 2007.

[4] N. Bertrand, A. Legay, S. Pinchinat, and J.-B. Raclet. A compositional approach on modal specifications for timed systems. In *ICFEM*, LNCS. Springer, 2009.

[5] N. Bertrand, S. Pinchinat, and J.-B. Raclet. Refinement and consistency of timed modal specifications. In *LATA*, volume 5457 of *LNCS*, Tarragona, Spain, 2009. Springer.

[6] P. Bulychev, T. Chatain, A. David, and K. G. Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *FORMATS*, volume 5813 of *LNCS*, pages 73–87. Springer, 2009.

[7] B. Caillaud, B. Delahaye, K. G. Larsen, A. Legay, M. Peddersen, and A. Wasowski. Compositional design methodology with constraint markov chains. Technical report, Hal-INRIA, 2009.

[8] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, 2005.

[9] K. Čerāns, J. C. Godskesen, and K. G. Larsen. Timed modal specification - theory and tools. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 253–267. Springer, 1993.

[10] A. Chakabarti, L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *EMSOFT 03: 3rd Intl. Workshop on Embedded Software*, LNCS. Springer, 2003.

[11] L. de Alfaro and M. Faella. An accelerated algorithm for 3-color parity games with an application to timed games. In *CAV*, volume 4590 of *LNCS*. Springer, 2007.

[12] L. de Alfaro and T. A. Henzinger. Interface automata. In *FSE*, pages 109–120, Vienna, Austria, Sept. 2001. ACM Press.

[13] L. de Alfaro and T. A. Henzinger. Interface-based design. In *In Engineering Theories of Software Intensive Systems, Marktoberdorf Summer School*. Kluwer Academic Publishers, 2004.

[14] L. de Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. In K. G. Larsen and M. Nielsen, editors, *CONCUR*, volume 2154 of *LNCS*, pages 536–550. Springer, 2001.

[15] L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Timed interfaces. In A. L. Sangiovanni-Vincentelli and J. Sifakis, editors, *EMSOFT*, volume 2491 of *LNCS*, pages 108–122. Springer, 2002.

[16] Z. Deng and J. W. s. Liu. Scheduling real-time applications in an open environment. In *in Proceedings of the 18th IEEE Real-Time Systems Symposium, IEEE Computer*, pages 308–319. Society Press, 1997.

[17] S. J. Garland and N. A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 1998.

[18] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *REX Workshop*, volume 600 of *LNCS*, pages 226–251. Springer, 1991.

[19] T. A. Henzinger and S. Matic. An interface algebra for real-time components. In *IEEE Real Time Technology and Applications Symposium*, pages 253–266. IEEE Computer Society, 2006.

[20] T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM*, volume 4085 of *LNCS*, pages 1–15. Springer, 2006.

[21] D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager. Timed i/o automata: A mathematical framework for modeling and analyzing real-time systems. In *RTSS*, pages 166–177. IEEE Computer Society, 2003.

[22] K. G. Larsen. Modal specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.

[23] K. G. Larsen, U. Nyman, and A. Wasowski. Modal I/O automata for interface and product line theories. In R. D. Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[24] I. Lee, J. Y.-T. Leung, and S. H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman, 2007.

[25] N. 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.

[26] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, The MIT Press, Nov. 1988.

[27] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.

[28] R. Milner. *Communication and Concurrency*. Prentice Hall, 1988.

[29] R. D. Nicola and R. Segala. A process algebraic view of input/output automata. *Theoretical Computer Science*, 138, 1995.

[30] E. W. Stark, R. Cleavland, and S. A. Smolka. A process-algebraic language for probabilistic I/O automata. In *CONCUR*, LNCS, pages 189–2003. Springer, 2003.

[31] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[32] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT*, pages 34–43. ACM, 2006.

[33] F. W. Vaandrager. On the relationship between process algebra and input/output automata. In *LICS*, pages 387–398, 1991.