# Synchronous Interface Theories and Time Triggered Scheduling

Benot Delahaye, Uli Fahrenberg, Axel Legay, Dejan Ničković



# Synchronous Interface Theories and Time Triggered Scheduling

Benoît Delahaye[1], Uli Fahrenberg[2], Thomas A. Henzinger[3], Axel Legay[21], and Dejan Ničković[3]

[1] Aalborg University, Denmark
[2] Irisa/INRIA Rennes, France
[3] IST Austria, Klosterneuburg, Austria

**Abstract.** We propose synchronous interfaces, a new interface theory for discrete-time systems. We use an application to time-triggered scheduling to drive the design choices for our formalism; in particular, additionally to deriving useful mathematical properties, we focus on providing a syntax which is adapted to natural high-level system modeling. As a result, we develop an interface model that relies on a guarded-command based language and is equipped with shared variables and explicit discrete-time clocks. We define all standard interface operations: compatibility checking, composition, refinement, and shared refinement. Apart from the synchronous interface model, the contribution of this paper is the establishment of a formal relation between interface theories and real-time scheduling, where we demonstrate a fully automatic framework for the incremental computation of time-triggered schedules.

## 1 Introduction

Interface models and theories were developed with the aim to provide a theoretical foundation for compositional design. Interface models describe both input assumptions on a component and its output guarantees and they support *incremental design* and *independent implementability*, two central concepts in component-based design. Interface theories [1, 8, 13, 16, 18, 21, 23, 30] and related approaches [10, 29] have been subject of active research in the past years, and today provide a strong and stable foundation for component-based design. However, although the theoretical foundations of interface theories can be now considered to be quite solid, the practical applicability of the framework has remained rather limited. One of the reasons is the fact that little attention has been given to adapt the modeling languages to the actual engineering needs in the target application domains.

In this paper, we propose *synchronous interfaces* (SI), a new interface theory motivated by an application to time-triggered scheduling and thus providing features that make our model closer to real-life needs of the engineers. In time-triggered communication scheduling, one allocates message transmissions to shared communication channels in a way that respects application-imposed and real-time constraints. The time-triggered scheduling problem can be naturally specified within an interface theory framework, by modeling scheduling

constraints as interface guarantees, and considering the environment to be the scheduler. We remark that even though our model has been developed with an eye to time-triggered scheduling, the application domain of the theory is much broader.

Incorrect scheduling of communication messages leads to violation of real-time and contention-freedom constraints, thus resulting in *timing incompatibilities*. This is in contrast to the standard interface theories that are untimed and are focused on reasoning about *value incompatibilities*. Continuous-time extensions of interface theories [15, 20] were developed to tackle this problem. While those are of clear interest and can solve interesting problems [14], they suffer from the complexity of handling continuous time in an explicit manner that is often unnecessary in practical application areas. We believe that discrete time provides the right level of abstraction for many application areas, and demonstrate it with the time-triggered scheduling application.

We base the syntax of SI on the model of *reactive modules*, a high-level and general-purpose modeling language that provides a syntax close to procedural guarded-command languages. In addition, we extend our model with *shared variables* that allow simple specification of contention freedom constraints, and explicit discrete-time *clocks* that facilitate modeling timing constraints.

Semantically, a SI is a set of concurrent processes whose behavior is evolving in discrete time. We equip our theory with operations that support incremental design and independent implementability: (1) *well-formedness check* that computes the set of environment choices for which the interface meets its guarantees; (2) *composition* that allows to combine two interfaces and compute the assumptions under which they interact in a compatible way and (3) *refinement* and *shared refinement* that are used to compare behaviors of different interfaces.

The second contribution of this paper is the *incremental* computation of time-triggered schedules, that is resolved as an incremental design problem with SI. We model scheduling constraints as SI guarantees and consider the environment to be the (unknown) scheduler. We apply well-formedness checking to restrict the environment to those schedules that satisfy the scheduling constraints. The composition operator allows to solve scheduling problems incrementally, by decomposing them into subproblems whose restricted environments are combined into a full schedule (using the well-formedness check again).

*Related Work.* Compositional scheduling for hierarchical RT systems has been extensively studied in [22, 32] and other papers by the same authors, but in a setting which in a sense is complementary to ours. The focus of this work is on computing bounds on resource use under some (simple) schedulers, and on inferring resource bounds for complex systems in a compositional manner, whereas we focus on schedulability, i.e., computation of schedules under given task dependencies and resource bounds. Incremental time-triggered scheduling was also studied in [33], using an approach that computes schedules with an SMT solver, but may miss a feasible schedule.

Another area of related work, similar in spirit but different in methods, is the recent application of timed-automata based formalisms to schedulability prob-

lems. In [2], simple job-shop scheduling problems are solved using timed automata, and in [11, 31], priced timed automata and games are used for schedulability under resource constraints. Another work in this area is [24], which is using timed automata extended with tasks for solving scheduling problems under uncertainty. Other approaches to solve worst-case scheduling problems are reported in [12, 28, 34]. Interface theories with shared variables were also proposed in [13] and [16, 17]. However, unlike in SI, the information about ownership of the shared variables by individual components within a composed system is not preserved, thus not making them suitable to express time-triggered scheduling problems.

Other examples of component-design based methodologies include the BIP toolset [6, 7] and its timed extension [3]. However, while BIP proposes features that are definitively beyond the scope of our work (generation of code, compilers, invariant-based verification), the approach does not permit to reason easily on shared variables, and does not provide (shared) refinement or pruning operators. Observe that several BIP-based approaches [9, 25] capable of restraining the behaviors of a distributed system by avoiding deadlocks have the potential to solve scheduling problems. However, a detailed study of (incremental) scheduling problems has not been considered in the mentioned papers, hence it is not clear whether TTEthernet scheduling would easily translate into the BIP framework.

## 2 Synchronous Interfaces

A synchronous interface comes equipped with a finite set $X$ of typed *variables* which is partitioned into sets $X = \mathit{extX} \cup \mathit{ctrX} \cup \mathit{sharedX}$ of *external*, *controlled*, and *shared* variables. External variables, also called *input* variables in interface theories [19], are controlled by the environment. At each round, the environment sets the values of external variables; the interface can read, but not modify them. Controlled, or *output* variables, are controlled by the interface: in each round, the interface assigns new values to all controlled variables. We further partition $\mathit{ctrX} = \mathit{intfX} \cup \mathit{privX}$ into *interface* and *private* variables. Interface variables can be seen by other SI, while private variables are local; hence private variables do not influence the communication behavior of a SI, and we can safely ignore them. We let $\mathit{obsX} = X \setminus \mathit{privX}$ denote the set of *observable* variables. We use *unprimed* symbols, such as $x$, to denote a *latched* value, and *primed* symbols, such as $x'$, to denote an *updated* value of the variable $x$. We naturally extend this notation to sets of variables. The function $\mathtt{type}(x)$ returns the type of variable $x$. In particular, clock variables have the type $\mathbb{C}$.

We follow the approach in [13] and introduce shared variables in the model, to facilitate communication using shared resources. We let the environment ensure the mutual exclusion property. Contrary to [13], we keep additional information on which individual component in the system owns the shared variable at each step of computation. In every computation step, the environment gives write access to a shared variable to at most one interface active in the system. We will define interface semantics following a game-oriented approach, hence this assumption is not a restriction.

**Definition 1.** *A* guarded command $\gamma$ *from variables* $X$ *to* $Y$ *consists of a* guard $p_\gamma$ *and an* action $\mathrm{Act}_\gamma$*. The guard* $p_\gamma$ *is a predicate over* $X$*, and* $\mathrm{Act}_\gamma$ *is either a* discrete action*: an expression* $\alpha_\gamma$ *from* $X$ *to* $Y$*, or a* wait action*, using the keyword* **wait**.

We use $\gamma[p_\gamma \setminus p'_\gamma]$ for the operation that consists in replacing the predicate $p_\gamma$ by the predicate $p'_\gamma$. Given a guarded command $\gamma$, the function $\pi(\gamma)$ returns the associated predicate $p_\gamma$ Control and shared variables are collected into *atoms*, which additionally contain guarded commands which specify rules for initializing and updating variables.

In interface theories, non-determinism reflects the fact that, given all the available information at a given step of the execution of an interface, several behaviors are possible for its next step. We let the environment resolve non-deterministic choices; this is implemented by assuming that for each $x \in \mathit{sharedX}$, there exists a non-empty set $\mathit{isCtr}_x = \{\mathit{isCtr}_x^A\}_{A \in M}$ of external variables, one for each atom $A \in M$ potentially controlling $x$. A variable $\mathit{isCtr}_x^A$ indicates whether atom $A$ can safely write $x$ at a given step of computation.

**Definition 2.** *An* $X$-atom $A$ *consists of a* declaration *and a* body *of guarded commands. We distinguish between atoms defined on controlled variables* $\mathrm{ctr}(A)$ *and those defined on shared variables* $\mathrm{shared}(A)$*.*

- *The atom declaration for* $\mathrm{ctr}(A)$ *consists of sets* $\mathrm{ctrX}_A \subseteq \mathrm{ctrX}$*,* $\mathrm{readX}_A \subseteq X$*, and* $\mathrm{waitX}_A \subseteq X \setminus \mathrm{ctrX}_A$ *of* controlled*,* read*, and* awaited *variables. The atom body for* $\mathrm{ctr}(A)$ *consists of a set* $\mathrm{Init}_A$ *of* initial *discrete guarded commands from* $\mathrm{waitX}'_A$ *to* $\mathrm{ctrX}'_A$ *and a set* $\mathrm{Update}_A$ *of* update *guarded commands from* $\mathrm{readX}_A \cup \mathrm{waitX}'_A$ *to* $\mathrm{ctrX}'_A$*.*
- *The atom declaration for* $\mathrm{shared}(A)$ *consists of sets* $\mathrm{sharedX}_A \subseteq \mathrm{sharedX}$*,* $\mathrm{readX}_A \subseteq X$*, and* $\mathrm{waitX}_A \subseteq X \setminus \mathrm{sharedX}_A$ *of* shared*,* read *variables, and* awaited *variables, with* $\mathrm{isCtr}_x^A \in \mathrm{waitX}_A$ *for all* $x \in \mathrm{sharedX}_A$*. The atom body for* $\mathrm{shared}(A)$ *consists of a set* $\mathrm{Init}(A)$ *of* initial *discrete guarded commands from* $\mathrm{waitX}'_A$ *to* $\mathrm{sharedX}'_A$ *and a set* $\mathrm{Update}(A)$ *of* update *guarded commands from* $\mathrm{readX}_A \cup \mathrm{waitX}'_A$ *to* $\mathrm{sharedX}'_A$*.*

We denote by $P_{\mathit{Init}(A)} = \{p_\gamma \mid \gamma \in \mathit{Init}(A)\}$ and $P_{\mathit{Update}(A)} = \{p_\gamma \mid \gamma \in \mathit{Update}(A)\}$ the sets of predicates declared in initial and in update guarded commands of $A$. We say that a variable $y$ awaits $x$, denoted $y \succ_A x$, if $y \in \mathit{ctrX}_A \cup \mathit{sharedX}_A$ and $x \in \mathit{waitX}_A$.

**Definition 3.** *A* synchronous interface *(SI)* $M$ *consists of a* declaration $X_M$ *and a* body $\mathcal{A}_M$*, where* $X_M$ *is a finite set of variables, and* $\mathcal{A}_M = \mathrm{ctr}(\mathcal{A}_M) \cup \mathrm{shared}(\mathcal{A}_M)$ *is a finite set of* $X_M$*-atoms for which* $\bigcup_{A \in \mathrm{ctr}(\mathcal{A}_M)} \mathrm{ctrX}_A = \mathrm{ctrX}_M$ *and* $\bigcup_{A \in \mathrm{shared}(\mathcal{A}_M)} \mathrm{sharedX}_A = \mathrm{sharedX}_M$*,* $\mathrm{ctrX}_{A_1} \cap \mathrm{ctrX}_{A_2} = \emptyset$ *for all atoms* $A_1, A_2 \in \mathrm{ctr}(\mathcal{A}_M)$*, and such that the transitive closure* $\succ_M = (\bigcup_{A \in \mathcal{A}_M} \succ_A)^+$ *is asymmetric.*

These conditions ensure that the atoms in $M$ control exactly the variables in $\mathit{ctrX}_M \cup \mathit{sharedX}_M$, that each variable in $\mathit{ctrX}_M$ is controlled by exactly one

```
1 module M_ex
2 external r : B, b : N, c : N
3          isCtr_x^b : B, isCtr_x^c : B;
4 shared x : N;

5 atom b reads b, x awaits r, isCtr_x^b
6   init
7     [] ¬isCtr_x^b' →;
8   update
9     [] isCtr_x^b' ∧ ¬r' → x' := x + b;
10    [] isCtr_x^b' ∧ r' → x' := 0;
11    [] ¬isCtr_x^b' →;

12 atom c reads c, x awaits r, isCtr_x^c
13   init
14     [] ¬isCtr_x^c' →;
15   update
16     [] isCtr_x^c' ∧ ¬r' → x' := x + c;
17     [] isCtr_x^c' ∧ r' → x' := 0;
18     [] ¬isCtr_x^c' →;
```

Fig. 1: An example of a SI

atom in $\mathcal{A}_M$, and that the await dependencies between variables in $\mathcal{A}_M$ are acyclic. A linear order $A_1, \ldots, A_n$ of the atoms in $\mathcal{A}_M$ is *consistent* if for all $1 \leq i < j \leq n$, the awaited variables in $A_i$ are disjoint from the control variables in $A_j$. The asymmetry of $\succ_M$ guarantees the existence of a consistent order of atoms in $\mathcal{A}_M$. We denote by $P_M = \bigcup_{A \in \mathcal{A}_M} P_{Init(A)} \cup \bigcup_{A \in \mathcal{A}_M} P_{Update(A)}$ the set of all predicates declared in the guarded commands of $M$. Remark that in our examples, we name atoms by the set of variables they control. This is only possible when all atoms have disjoint sets of controlled variables.

*Example 1.* Consider the SI $\mathrm{M_{ex}}$ given in Figure 1. $\mathrm{M_{ex}}$ consists of two external Integer variables $b, c$, three external Boolean variables $r, isCtr_x^b, isCtr_x^c$, and a shared Integer variable $x$. Intuitively, $\mathrm{M_{ex}}$ models a simple additive controller that works as follows. The shared variable $x$ is either incremented or reset at each time step in which the module controls $x$. $\mathrm{M_{ex}}$ controls $x$ whenever ($isCtr_x^b \vee isCtr_x^c = \mathtt{t}$). In this case, if $r = \mathtt{t}$, then $x$ is reset. Else, if Atom $b$ (resp. $c$) controls $x$ ($isCtr_x^b = \mathtt{t}$), then $x$ is incremented by the value of $b$ (resp. $c$). If none of these atoms assign a value to the variable, then the environment will do.

## 3 Semantics

The intuition about the semantics of a SI is as follows: in each round, the environment assigns arbitrary values of correct type to external variables. Then, the atoms are executed in a (abitrary) static consistent order. As we do not assume that modules are input-enabled, there may be valuations of the external variables for which one or several atoms cannot be executed. Such configurations result in *deadlock states*. A given valuation of the variables is *reachable* if there exists a succession of rounds of the atom ending in this valuation. The appendix contains an illustration of this intuition.

Formally, the semantics of a SI is given by a *labeled transition system* (LTS). Given a SI $M$ with set of variables $X_M$, we denote by $V[X_M]$ the set of valuations on variables in $X_M$. A *state* $s$ of an interface $M$ is a valuation in $V[X_M]$. We write $\Sigma_M = \Sigma_{X_M}$ for the set of states of $M$. Given a state $s \in \Sigma_M$ and $Y \subseteq X_M$, we denote by $s[Y]$ the projection of the state $s$ to the valuations of variables in $Y$. Note that we will define the semantics in a way which keeps enough information about the syntax to be able to go back from semantics to syntax; this is important

for several of the operations which we define in the next section, as these are defined only at the semantics level.

Given a state $s$ of an interface $M$, we denote by $\texttt{safe}^{sv}_M(s)$ the predicate that indicates whether the state is safe with respect to shared variables in $M$; formally, $\texttt{safe}^{sv}_M(s) = \texttt{t}$ iff $\forall x \in sharedX_M, \bigwedge_{A,A' \in M, A \neq A'} s[isCtr^A_x] \wedge s[isCtr^{A'}_x] = \texttt{f}$. Intuitively, a state $s$ of $M$ is safe if and only if, for all shared variables $x$, there is at most one atom that controls $x$.

**Definition 4.** *Let $X$, $Y$, $Z \subseteq Y$ be sets of variables and $\gamma$ a guarded command from $X$ to $Y$. We define the semantics $[\![\gamma]\!] \subseteq \Sigma_X \times \Sigma_Y$ of $\gamma$ as follows:*

- *If $\gamma$ is of the form $p_\gamma \to \alpha_\gamma$, where $\alpha_\gamma : V[X] \to V[Z]$, then $(s,t) \in [\![\gamma]\!]$ iff (1) $s \models p_\gamma$; (2) $\forall z \in Z, t[z] = \alpha(s)[z]$; (3) $\forall y \in Y \backslash Z$ such that $\texttt{type}(y) \neq \mathbb{C}$, $t[y] = s[y]$ and (4) $\forall y \in Y \backslash Z$ such that $\texttt{type}(y) = \mathbb{C}$, $t[y] = s[y] + 1$*
- *If $\gamma$ is of the form $p_\gamma \to$ **wait**, then $(s,t) \in [\![\gamma]\!]$ iff (1) $s \models p_\gamma$, (2) $\forall y \in Y$ such that $\texttt{type}(y) = \mathbb{C}$, $t[y] = s[y] + 1$, (3) $\forall y \in Y$ such that $\texttt{type}(y) \neq \mathbb{C}$, $t[y] = s[y]$, and (4) $t \models p_\gamma$.*

*Let $A$ be an atom from $X$ to $Y$ and let $\Gamma_A$ be either* $\mathrm{Init}(A)$ *or* $\mathrm{Update}(A)$*, i.e., a finite set of guarded commands. Then, $\Gamma_A$ defines a relation $[\![\Gamma_A]\!] \subseteq \Sigma_X \times \Sigma_Y$ such that $(s,t) \in [\![\Gamma_A]\!]$ iff $(s,t) \in [\![\gamma]\!]$ for some $\gamma \in \Gamma_A$.*

The semantics of SI is a LTS whose states represent valuations of variables, and whose transitions correspond to complete rounds of updates for all atoms $A_i$ in a static consistent order $A_1, \dots A_n$.

**Definition 5.** *The semantics of a SI $M$ is the LTS $[\![M]\!] = (S_M, S^0_M, \to_M, L_M)$ with $S_M = V[X_M] \cup \{s_{init}\}$, $S^0_M = \{s_{init}\}$, $L_M \subseteq P_M^{\mathcal{A}_M}$ the set of all functions $l : \mathcal{A}_M \to P_M$ for which $l(A) \in P_A$ for all $A \in \mathcal{A}_M$, and $\to_M$ defined as follows:*

- *$(s_{init}, l, t) \in \to_M$ iff $\texttt{safe}^{sv}_M(t)$ and there exist $\gamma_1, \dots, \gamma_n$ such that for all $1 \leq i \leq n$, $\gamma_i \in \mathrm{Init}(A_i), l(A_i) = \pi(\gamma_i)$ and there exists $s^0 \in V[X_M]$ such that $t = [\![\mathrm{Init}(A_n)]\!] \circ \dots \circ [\![\mathrm{Init}(A_1)]\!](s^0)$*
- *$(s,l,t) \in \to_M$ iff $\texttt{safe}^{sv}_M(s)$, $\texttt{safe}^{sv}_M(t)$, and there exist $\gamma_1, \dots, \gamma_n$ such that for all $1 \leq i \leq n$, $\gamma_i \in \mathrm{Update}(A_i), l(A_i) = \pi(\gamma_i)$ and $t = [\![\mathrm{Update}(A_n)]\!] \circ \dots \circ [\![\mathrm{Update}(A_1)]\!](s)$*

Note that we label each transition by the guarded commands that are effectively executed during the round, hence we preserve full syntactic information about the interface in its semantics. An illustration of this is given in the appendix. In the following, we may omit this labelling in our notations when we do not need the information.

A *trajectory* of a SI $M$ is a *finite* sequence of states $s_0, s_1, \dots, s_n$ in $[\![M]\!]$ such that: (1) $s_0 = s_{init}$; (2) $(s_i, s_{i+1}) \in \to_M$ for all $0 \leq i < n$; and (3) no deadlock states are reachable from $s_n$. The sequence $s_1[obsX_M], \dots, s_n[obsX_M]$ of observable valuations is called a *trace* of $M$; the *trace language* $L(M)$ of $M$ is the set of traces of $M$. In our optimistic approach, computing environments that cannot result in deadlock states amounts to projecting $L(M)$ onto the external variables. Note that we can effectively compute these environments if the interface has a finite representation of its trace language, i.e. if $[\![M]\!]$ has a finite state space. We say that $M$ is *well-formed* if $L(M) \neq \emptyset$.

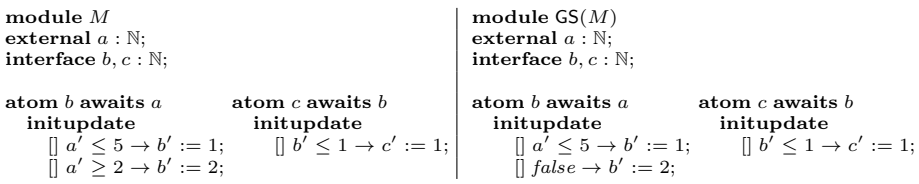



Fig. 2: An example of guard strengthening. Left: $M$, right: $\mathsf{GS}(M)$

## 4 Operations

We now describe some operations on SI which will allow us to use SI as an interface theory. Note that we will also use some of these operations for incremental scheduling in Section 5; but notably shared refinement is not used in incremental scheduling, yet a necessary ingredient in any interface theory.

**Guard strengthening.** Given a well-formed interface $M$, we are interested in computing an equivalent module (in terms of infinite executions) in which no deadlock states are reachable. This *guard strengthening* $\mathsf{GS}(M)$ is computed by strengthening the guards of $M$ in such a way that deadlocks are forbidden. An illustration of guard strengthening is given in Figure 2.

Semantically, the construction relies on a notion of recursively *pruning* deadlock states together with states which inevitably lead to them: Let $M$ be a SI and let $[\![M]\!] = (S_M, S_M^0, \rightarrow_M, L_M)$ be its associated LTS. Define the function $\mathrm{Succ}_X : S_M \times V[X] \rightarrow 2^{S_M}$ by

$$\mathrm{Succ}_X(q, v) = \{q' \mid (q, q') \in \rightarrow_M \text{ and } q'[X] = v\}.$$

This function gives all successors of $q$ in $[\![M]\!]$ which for variables in $X$ match the valuation $v$. Next we define a mapping Pred which outputs the controllable predecessors of a subset $B \subseteq S_M$:

$$\mathrm{Pred}(B) = \{s \mid \forall v \in V[extX_M] : \mathrm{Succ}_{extX_M}(s, v) \neq \emptyset \Rightarrow \mathrm{Succ}_{extX_M}(s, v) \cap B \neq \emptyset\}.$$

Denote by $\mathrm{Pred}^*$ the transitive closure of Pred, let $B = \{s \in S_M \mid \forall t \in S_M : (s, t) \notin \rightarrow_M\}$ be the deadlock states and $B^* = \mathrm{Pred}^*(B)$. Intuitively, these are states from which the environment cannot prevent $M$ from reaching a deadlock.

The *pruning* of $[\![M]\!]$ is then given by the LTS $\rho([\![M]\!]) = (S_M \setminus B^*, S_M^0 \setminus B^*, \rightarrow'_M, L_M)$, where $\rightarrow'_M = \{(s, l, s') \in \rightarrow_M \mid s' \notin B^*\}$. Intuitively, pruning $[\![M]\!]$ removes all bad states and transitions leading to them, which reduces the state-space of $M$ without affecting its language. Note that if $M$ is not well-formed, then $\rho([\![M]\!])$ has no initial states; we then say that the pruning of $M$ is empty.

For guard strengthening at the *syntactic* level, matching the pruning of the semantics, we proceed as follows: for each initial set of guarded commands $\mathit{Init}(A)$ of an atom $A$ in $\mathcal{A}_M$ and $\gamma \in \mathit{Init}(A)$, the predicate $p_\gamma \in \gamma$ is replaced by

$$\tilde{p}_\gamma = \bigvee_{(s_{init}, t \in S_M \setminus B^*, (s,l,t) \in \rightarrow_M, l(A) = p_\gamma)} \bigwedge_{(x \in X_M[waitX_A])} x' = t[x].$$

Similarly, for each update set of guarded commands $Update(A)$ and $\gamma \in Update(A)$, the predicate $p_\gamma \in \gamma$ is replaced by

$$\tilde{p}_\gamma = \bigvee_{(s,t\in S_M\setminus B^*,(s,l,t)\in\to_M,l(A)=p_\gamma)} \bigwedge_{(x\in X_M[waitX_A])} x' = t[x] \wedge \bigwedge_{(x\in X_M[readX_A])} x = s[x].$$

Intuitively, this method amounts to an enumeration, for every guarded command, of possible valuations of read and awaited variables that cannot reach a bad state. Replacing original predicates with associated enumerations prevents exactly bad behaviors. It follows, by construction, that $[\![\mathsf{GS}(M)]\!] \equiv \rho([\![M]\!])$, hence also that $M$ is well-formed if and only if $[\![\mathsf{GS}(M)]\!]$ is not empty. As the trace language $L(M)$ by definition only includes traces which cannot be extended to a deadlock state, we also have $L(M) = L(\mathsf{GS}(M))$.

**Parallel Composition.** We introduce a synchronous parallel composition operation which combines two compatible SI. We say that SI $M$ and $N$ are *composable* if (1) the interface variables of $M$ and $N$ are disjoint; and (2) the await dependencies between the observable variables of $M$ and $N$ are acyclic, that is the transitive closure $(\succ_M \cup \succ_N)^+$ is asymmetric. Observe that the sets of shared variables may overlap, and that private variables are not taken into consideration here: these are not visible from the outside, hence in case of private variables with the same name, we consider that they are different and belong to different name spaces.

We say that two composable SI are *compatible* if there exists an environment in which they can be composed without reaching deadlock states. Informally, the synchronous composition $P$ of $M$ and $N$ consists in the union of their atoms, where some controlled variables in $M$ can constrain external variables in $N$, and vice-versa. An execution of $P$ thus consists in an update of all the remaining external variables, followed by an update of the controlled and shared variables of $M$ and $N$.

**Definition 6.** *Let $M$ and $N$ be two composable SI. Define an intermediate module $P$ by* $\mathrm{privX}_P = \mathrm{privX}_M \cup \mathrm{privX}_N$, $\mathrm{intfX}_P = \mathrm{intfX}_M \cup \mathrm{intfX}_N$, $\mathrm{extX}_P = \mathrm{extX}_M \cup \mathrm{extX}_N \setminus \mathrm{intfX}_P$, $\mathrm{sharedX}_P = \mathrm{sharedX}_M \cup \mathrm{sharedX}_N$, *and finally,* $\mathcal{A}_P = \mathcal{A}_M \cup \mathcal{A}_N$. *$M$ and $N$ are* compatible *if $P$ is well-formed, in which case we define* $M \parallel N = \mathsf{GS}(P)$.

The below theorem shows that parallel composition is *associative*, hence allowing incremental composition. The theorem follows directly from the fact that pruning does not affect the trace language.

**Theorem 1.** *For composable SI $M_1$, $M_2$, $M_3$,* $L(M_1 \parallel (M_2 \parallel M_3)) = L((M_1 \parallel M_2) \parallel M_3)$.

**Refinement.** Refinement of SI allows comparing interfaces. Informally, if $N$ refines $M$, then $N$ works in at least all the environments where $M$ works,

and all the behaviors of $N$ defined in these environments are also behaviors of $M$. Hence refinement for SI is similar to alternating simulation for I/O Automata [4]. For valuations $v \in V[extX_M]$, we define the set $ctr_M(v)$ of shared variables that are controlled by $M$ according to $v$ by $\mathrm{ctr}_M(v) = \{x \in sharedX_M \mid (\bigvee_{A \in M} v[isCtr_x^A]) = \mathtt{t}\}$, and we let $\mathrm{noctrl}_M(v) = extX_M \cup (sharedX_M \setminus \mathrm{ctr}_M(v))$.

**Definition 7.** *Let $M$, $N$ be SI and $[\![M]\!] = (S_M, S_M^0, \to_M)$, $[\![N]\!] = (S_N, S_N^0, \to_N)$. We say that $N$* refines *$M$, written as $N \leq M$, if $\mathrm{extX}_M \subseteq \mathrm{extX}_N$, $\mathrm{intfX}_N \subseteq \mathrm{obsX}_M$, $\mathrm{sharedX}_N = \mathrm{sharedX}_M$, and there exists a relation $R \subseteq S_N \times S_M$ such that $(s_{init}, t_{init}) \in R$ and for all $(s,t) \in R$, we have*

- *$(s,t) \neq (s_{init}, t_{init})$ implies that $s[\mathrm{extX}_M \cup \mathrm{intfX}_N \cup \mathrm{sharedX}_M] = t[\mathrm{extX}_M \cup \mathrm{intfX}_N \cup \mathrm{sharedX}_M]$*
- *for all $v \in V[\mathrm{extX}_M]$ and $v' \in V[\mathrm{sharedX}_M \setminus \mathrm{ctr}_M(v)]$ it holds that if $\mathrm{Succ}_{\mathrm{noctrl}(v)}(t, v \cup v') \neq \emptyset$, then also $\mathrm{Succ}_{\mathrm{noctrl}(v)}(s, v \cup v') \neq \emptyset$, and then for all $s' \in \mathrm{Succ}_{\mathrm{noctrl}(v)}(s, v \cup v')$, there exists $t' \in \mathrm{Succ}_{\mathrm{noctrl}(v)}(t, v \cup v')$ such that $s' R t'$.*

The relation between refinement and trace languages is as follows: for a SI $M$, let $\mathrm{adm}(M) = \{w \in extX_M^* \mid \exists w' \in L(M). w'{\downarrow_{extX_M}} = w\}$ be the set of all *admissible external valuations*; here $w'{\downarrow_{extX_M}}$ denotes the projection of $w'$ to external variables, hence we are collecting all traces of valuations of external variables which *do not block* execution of $M$. Then:

**Theorem 2.** *For SI $N$, $M$ with $N \leq M$ we have $\{w \in L(N) \mid w{\downarrow_{\mathrm{extX}_M}} \in \mathrm{adm}(M)\} \subseteq L(M)$.*

The next theorem shows that SI theory supports *independent implementability*: Refinement is compatible with parallel composition in the sense that components may be refined individually. The proofs of both theorems are in the appendix.

**Theorem 3.** *Given SI $M_1, M_1'$, $M_2$, $M_2'$ with $M_1$ and $M_2$ compatible, if $M_1' \leq M_1$ and $M_2' \leq M_2$, then $M_1'$ is compatible with $M_2'$ and $M_1' \parallel M_2' \leq M_1 \parallel M_2$.*

**Shared Refinement.** We finish this section by mentioning that there also is a notion of *shared refinement* for SI which supports component reuse in different parts of a design; due to space constraints, we defer the formal details to the appendix.

The shared refinement of two SI $M_1$ and $M_2$ is the SI $M = M_1 \wedge M_2$ which is the product of the state spaces in LTSs of $M_1$ and $M_2$, with appropriate transitions ensuring that $M_1$ and $M_2$ evolve synchronously along the same transitions. Hence $M$ accepts inputs that satisfy any of the assumptions from $M_1$ and $M_2$, and it provides outputs that satisfy both guarantees of $M_1$ and $M_2$. In particular, $M$ can be used to implement two different aspects of a single component. Moreover, $M$ is the *smallest* such SI in the sense of the theorem below.

**Theorem 4.** *Given two SI $M_1$ and $M_2$, we have that: (1) $M_1 \wedge M_2 \leq M_1$; (2) $M_1 \wedge M_2 \leq M_2$; and (3) for all SI $M'$ such that $M' \leq M_1$ and $M' \leq M_2$, also $M' \leq M_1 \wedge M_2$.*

## 5 Incremental TTEthernet Scheduling with SI

In this final section we present a methodology for solving scheduling problems using the synchronous interface theory developed in this paper. We concentrate on the particular application of *TTEthernet* scheduling [26], but our framework is sufficiently general that it also allows application to other scheduling and job-shop problems.

A specification of a TTEthernet network consists of a *physical topology*, a set of *frames* and a set of time-triggered scheduling *constraints*. The physical topology is an undirected graph consisting of a set of vertices, corresponding to communicating *devices* (end-systems or switches), and edges, representing bi-directional communication links, called *data-flow links*, between devices. A frame specifies a message that is sent over the network, and is represented by a tree that defines the route for the message delivery from a sender device to a set of receiver devices. Every edge in the tree represents the frame on a particular data-flow link, and is characterized by its *period* (relative deadline for the frame arrival from the sender to its receiver), *length* (value denoting frame delivery duration on the data-flow link) and *offset* (actual time slot at which the frame is sent from a sender to a receiver device). Like in [33], we assume, without loss of generality, that the frame period *Period* is the same for all frames on all data-flow links in the specification. Finally, time-triggered scheduling constraints are defined over the offset values of frames on data-flow links. To simplify presentation, we consider only two most common types of TT scheduling constraints: (1) *contention free* (CF) constraints that impose to any reasonable schedule to forbid simultaneous presence of two frames on the same data-flow link; and (2) *path-dependent* (PD) constraints that impose correct flow of a frame through data-flow links, ensuring that a device cannot send a frame before receiving it.

In a TTEthernet network specification, the only non-fixed values are the offset parameters of frames in data-flow links. A schedule that satisfies the specification corresponds to an assignment of concrete values to offset parameters which satisfies all the constraints. The TT scheduling problem consists in computing such a schedule from a specification.

We introduce our methodology by way of example below, but in essence, it proceeds as follows:

1. Introduce a SI *Clock* which keeps track of time within a period.
2. Model each frame as a SI, including transmission length, path dependency, and shared resources.
3. Use parallel composition and well-formedness check to incrementally reject all non-feasible offset values.
4. If any feasible offset values remain after the preceding step, then any of these constitutes a feasible schedule. Otherwise the problem is unschedulable.

Figure 3 depicts an example of a time-triggered scheduling problem for a particular TTEthernet network specification. The input to the scheduling problem consists of the network topology $N$ (Figure 3(a)) and two frames $f_1$, $f_2$ (Figures 3(b) and (c)). The contention-freedom and path-dependency constraints induced by the frames are depicted in Figure 3(d). Solving the scheduling problem

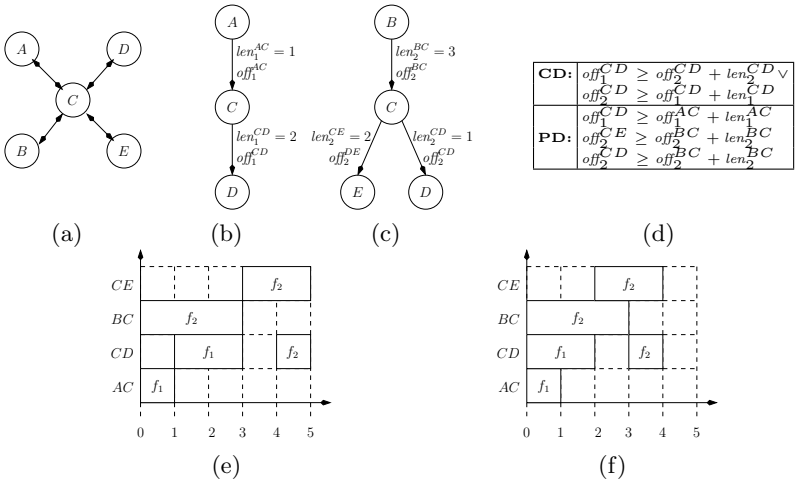



Fig. 3: Specification of a TTEthernet network: (a) network topology $N$; (b), (c) specification of frame routes $f_1$ and $f_2$; (d) constraints on offset values; (e) feasible schedule; (f) infeasible schedule

specified in Figure 3 consists in computing the feasible schedules that satisfy all the requirements of the specification. Figures 3(e) and (f) depict two schedules, one that satisfies and another that violates the specification.

```
module Clock
interface clk_P : ℂ;
atom clk_P reads clk_P
  init
    [] t → clk_P' := 0;
  update
    [] (clk_P ≥ P − 1) → clk_P' := 0;
    [] clk_P < P − 1 → wait;
```

Fig. 4: SI *Clock*

To solve the example scheduling problem, we first introduce a SI *Clock* as depicted in Figure 4 which measures the relative time within every period using an explicit clock variable $clk_P$. This clock is visible to all other interfaces in the system. Then we model the two frames $f_1$ and $f_2$ as two independent interfaces $M_1$ and $M_2$; this will allow to solve the problem incrementally. The application of the well-formedness check operator on the composition $Clock \| M_i$ computes the set of all feasible partial schedules that are consistent with the scheduling (path-dependency) constraints of the frame $f_i$. We then use the parallel composition of *Clock* with $M_1$ and $M_2$ to combine compatible partial schedules for $f_1$ and $f_2$, effectively removing all schedules that violate the contention-freedom constraint.

We now encode the frame $f_1$ as a SI $M_1$, shown in Figure 5. The environment (scheduler) owns the variables $soff_1^{AC}$ and $soff_1^{CD}$ (line 2), that are used to propose in the initial state the offset values for the message of frame $f_1$ on the data flow $AC$ and $CD$, respectively. The interface $M_1$ checks in line 8 whether the proposed values satisfy the path-dependency constraint, and accordingly rejects the offsets, or accepts them and copies them into the controlled variables $off_1^{AC}$ and $off_1^{CD}$. The atom depicted in lines $6-10$ controls a local clock $clk_1^{AC}$, that measures the time length of the message transmitted on the data flow link $AC$

1 **module** $M_1$
2 **external** $soff_1^{AC} : [0, P), soff_1^{CD} : [0, P), clk_P : \mathbb{C}$;
$isCtr^1_{x_{CD}} : \mathbb{B}$;
3 **interface** $off_1^{AC} : [0, P), off_1^{CD} : [0, P)$
4 **interface** $clk_1^{AC} : \mathbb{C}; clk_1^{CD} : \mathbb{C}$;
5 **shared** $x_{CD} : \mathbb{B}$;

6 **atom** $off_1^{AC}, off_1^{CD}$ **awaits** $soff_1^{AC}, soff_1^{CD}$
7 **init**
8 [] $soff_1^{CD\prime} \geq soff_1^{AC\prime} + len_1^{AC} \rightarrow$
$off_1^{AC\prime} := soff_{1AC}{}', off_1^{CD\prime} := soff_1^{CD\prime}$;
9 **update**
10 [] $\mathtt{t} \rightarrow$;

11 **atom** $clk_1^{AC}$ **awaits** $off_1^{AC}, clk_P$
12 **init**
13 [] $off_1^{AC\prime} = 0 \rightarrow clk_1^{AC\prime} := 0$;
14 [] $off_1^{AC\prime} \neq 0 \rightarrow clk_1^{AC\prime} := \bot$;
15 **update**
16 [] $clk_1^{AC\prime} < len_1^{AC} \wedge clk_P{}' < P - 1 \rightarrow$ **wait**;
17 [] $clk_1^{AC\prime} = len_1^{AC} \wedge clk_P{}' < P \rightarrow clk_1^{AC\prime} := \bot$;

18 **atom** $clk_1^{CD}$ **awaits** $off_1^{CD}, clk_P$
19 **init**
20 [] $off_1^{CD\prime} \neq 0 \rightarrow clk_1^{CD\prime} := \bot$;
21 **update**
22 [] $clk_1^{CD\prime} < len_1^{CD} \wedge clk_P{}' < P - 1 \rightarrow$ **wait**;
23 [] $clk_1^{CD\prime} = len_1^{CD} \wedge clk_P{}' < P \rightarrow clk_1^{CD} := \bot$;

24 **atom** $x_{CD}$ **awaits** $clk_1^{CD}, isCtr^1_{x_{CD}}$
25 **initupdate**
26 [] $isCtr^1_{x_{CD}} \wedge clk_1^{CD\prime} \in [0, len_1^{CD}) \rightarrow x_{CD}{}' := \mathtt{t}$;
27 [] $\neg isCtr^1_{x_{CD}} \wedge clk_1^{CD\prime} \notin [0, len_1^{CD}) \rightarrow$

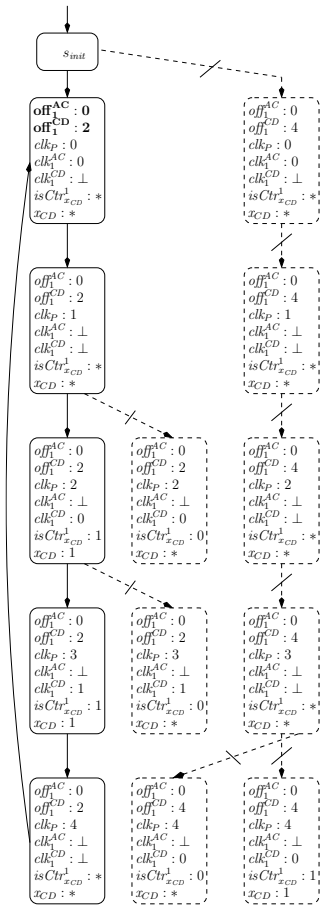



Fig. 5: Synchronous interface $M_1$: (a) syntax; (b) part of its pruned semantics

by the frame $f_1$. The clock $clk_1^{AC}$ is reset when the corresponding offset value is reached, and the atom ensures that the transmission of the message is finished before the end of the period $P$. The atom that controls the local clock $clk_1^{CD}$, depicted in lines $18-23$, does the same monitoring of the message transmitted by $f_1$ on the data flow link $CD$. Finally, the last atom controls the shared variable $x_{CD}$, that models the shared resource (data flow link) $CD$. It ensures that when the frame $f_1$ is given access to the data flow link $CD$ (via the external variable $isCtr^1_{x_{CD}}$), it is not preempted before the message transmission is done.

In order to compute the partial feasible schedules for $f_1$, one needs to apply the well-formedness check on $Clock \| M_1$, which amounts to generating the pruned semantics graph of this composition (also shown (in parts) in Figure 5). The well-formedness checking results in pruning all states that lead to a deadlock, i.e., it removes all states where the offsets that are proposed by the environment result in a violation of a scheduling constraint. The partial feasible schedules are encoded as the valuations of $off_1^{AC}$ and $off_1^{CD}$ in the remaining initial states.

The encoding of the scheduling problem for $f_2$ into a synchronous interface $M_2$, and the corresponding computation of the partial feasible schedules for $f_2$ is done in a similar way. Given the pruned transition systems $\rho([\![Clock \| M_1]\!])$ and $\rho(Clock \| [\![M_2]\!])$, the parallel composition combines the two systems and removes the joint behaviors that are not compatible. In our example, it amounts to remove all the behaviors in which the mutual exclusion property on the access to the shared variable $x_{CD}$ is violated, thus falsifying the contention-freedom scheduling constraint. The pruned transition system of the composition encodes exactly all feasible schedules of the original problem. Figure 6 depicts two fragments of the transition systems for $Clock \| M_1$ and $Clock \| M_2$ and of the pruned semantics of their composition.

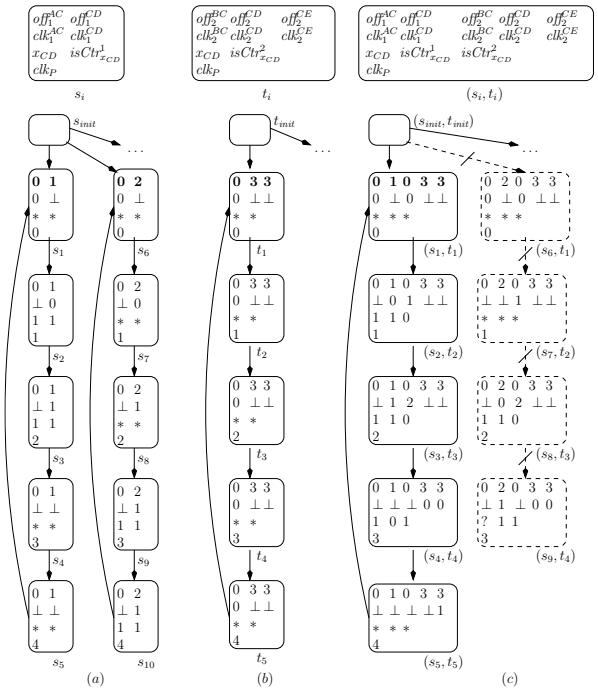



Fig. 6: Composition of $M_1$ and $M_2$: fragment of (a) $\rho([\![M_1]\!])$; (b) $\rho([\![M_2]\!])$ and (c) $\rho(\rho([\![M_1]\!]) \parallel \rho([\![M_2]\!]))$

## 6 Conclusion and Further Work

We present in this paper a simple yet powerful model for *synchronous interfaces*. Contrary to most other interface models one finds in the literature, the modeling language we use is inspired by a specific application domain, resulting in a model that resembles a high-level programming language, hence is easy to use. At the same time, we allow explicit use of time and of shared variables that are treated in

a flexible way, resulting in a rich model which satisfies most common engineering needs. We develop our model into an *interface theory*, allowing for high-level reasoning and component-based design using (shared) refinement, composition and pruning. We propose to use our interface theory as an elegant solution for incremental computation of time-triggered schedules.

In the future, we plan to implement the SI framework and apply it to different scheduling problems. We also believe that the state-based type of analysis on SI makes our approach a good candidate for development of efficient and flexible heuristics, by assigning value functions to states and restricting the search space to the assigned values. Finally, we plan to extend our approach in order to incorporate deeper information about the platform on which the system is running, like in the spirit of recent works [5, 27] done in the context of untimed BIP and UPPAAL frameworks.

## References

1. F. Aarts and F. W. Vaandrager. Learning I/O automata. In *CONCUR*, volume 6269 of *LNCS*, pages 71–85. Springer, 2010.
2. Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. *TCS*, 354(2):272–300, 2006.
3. T. Abdellatif, J. Combaz, and J. Sifakis. Model-based implementation of real-time applications. In *EMSOFT*, pages 229–238. ACM, 2010.
4. R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *CONCUR '98*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998.
5. A. Basu, S. Bensalem, P. Bourgos, M. Bozga, K. Huang, and J. Sifakis. Rigorous system level modeling and analysis of mixed HW/SW systems. In *Memocode*. IEEE, 2011. to appear.
6. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.
7. A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis. Using BIP for modeling and verification of networked systems – a case study on TinyOS-based networks. In *NCA*, pages 257–260. IEEE Computer Society, 2007.
8. S. S. Bauer, P. Mayer, A. Schroeder, and R. Hennicker. On weak modal compatibility, refinement, and the MIO workbench. In *TACAS*, pages 175–189, 2010.
9. S. Bensalem, M. Bozga, S. Graf, D. Peled, and S. Quinton. Methods for knowledge based controlling of distributed systems. In *ATVA'10*, volume 6252 of *LNCS*, pages 52–66. Springer, 2010.
10. A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In *FMCO*, volume 5382 of *LNCS*, pages 200–225. Springer, October 2008.
11. P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, and J. Srba. Infinite runs in weighted timed automata with energy constraints. In *FORMATS*, pages 33–47, 2008.
12. A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. *PRTS*, pages 225 – 248, 1994.
13. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Synchronous and bidirectional component interfaces. In *CAV*, volume 2404 of *LNCS*, pages 414–427, 2002.

14. A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. ECDAR: An environment for compositional design and analysis of real time systems. In *ATVA*, volume 6252 of *LNCS*, pages 365–370. Springer, 2010.
15. A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC*, pages 91–100. ACM ACM, 2010.
16. L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *FroCos*, volume 3717 of *LNCS*, pages 81–105. Springer, 2005.
17. 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*, pages 108–120. Springer, 2007.
18. L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.
19. L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT*, volume 2211 of *LNCS*. Springer, 2001.
20. L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed interfaces. In *EMSOFT*, volume 2491 of *LNCS*, pages 108–122. Springer, 2002.
21. L. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *EMSOFT*, pages 79–88. ACM, 2008.
22. A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hierarchical real-time components. In *EMSOFT*, pages 272–281. ACM, 2006.
23. M. Emmi, D. Giannakopoulou, and C. S. Pasareanu. Assume-guarantee verification for interface automata. In *FM*, volume 5014 of *LNCS*, pages 116–131. Springer, 2008.
24. E. Fersman, P. Krčál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *I&C*, 205(8):1149–1172, 2007.
25. S. Graf, D. Peled, and S. Quinton. Monitoring distributed systems using knowledge. In *FMOODS/FORTE'11*, volume 6722 of *LNCS*, pages 183–197. Springer, 2011.
26. H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered ethernet (TTE) design. In *ISORC'05*, pages 22–33. IEEE Computer Society, 2005.
27. M. Mikučionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard. Schedulability analysis using Uppaal: Herschel-Planck case study. In *ISoLA (2)*, volume 6416 of *LNCS*, pages 175–190. Springer, 2010.
28. S. Palm. Herschel-Planck ACC ASW: sizing, timing and schedulability analysis. Tech. rep., Terma A/S, 2006.
29. S. Quinton and S. Graf. Contract-based verification of hierarchical systems of components. In *SEFM'08*, pages 377–381. IEEE Computer Society, 2008.
30. J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. Modal interfaces: unifying interface automata and modal specifications. In *EMSOFT*, pages 87–96. ACM, 2009.
31. J. I. Rasmussen, K. G. Larsen, and K. Subramani. On using priced timed automata to achieve optimal scheduling. *FMSD*, 29(1):97–114, 2006.
32. I. Shin and I. Lee. Compositional real-time scheduling framework. In *RTSS*, pages 57–67. IEEE Computer Society, 2004.
33. W. Steiner. An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks. In *RTSS*, pages 375–384, 2010.
34. Terma A/S. Software timing and sizing budgets. Tech. rep., Terma A/S, Issue 9.

# Appendix

### Semantics of SI: Examples

*Example 2.* The execution of a SI is illustrated in Figure 7a, where the atoms are executed in a consistent order $A_1, \ldots, A_n$. The SI reads latched variables, updates external variables and writes updated control variables. In order to proceed, some atoms may also need to read updated values of control variables that are controlled by preceding atoms in the consistent order. This is illustrated along with details of the execution of a round in Figure 7b.

*Example 3.* Consider again the SI $\mathrm{M_{ex}}$ from Example 1. A part of the unfolding of the semantics $[\![\mathrm{M_{ex}}]\!]$ of $\mathrm{M_{ex}}$ is given in Figure 8. Observe that state $s_4$ of Figure 8 cannot be reached because it is not safe, as the two atoms are controlling the shared variable $x$ at the same time.

### Proofs from Section 4

*Proof (of Theorem 2).* If the valuations of external variables in $w$ are such that they do not block $M$, then we have $\mathrm{Succ}_{\mathrm{noctrl}(v)}(t, v \cup v') \neq \emptyset$ in every corresponding step of the refinement relation, hence any valuation in $\mathrm{Succ}_{\mathrm{noctrl}(v)}(s, v \cup v')$ can be matched by one in $\mathrm{Succ}_{\mathrm{noctrl}(v)}(t, v \cup v')$. □

*Proof (of Theorem 3).* The conditions on composability of $M_1'$ and $M_2'$ are clearly satisfied, and the environment for which $M_1$ and $M_2$ can be composed without reaching deadlock states has the same property for $M_1'$ and $M_2'$, hence $M_1' \parallel M_2'$ exists. Letting $R_1 \subseteq S_{M_1'} \times S_{M_1}$, $R_2 \subseteq S_{M_2'} \times S_{M_2}$ be the refinement relations, we see that $R_1 \times R_2$ is a refinement relation showing $M_1' \parallel M_2' \leq M_1 \parallel M_2$. □

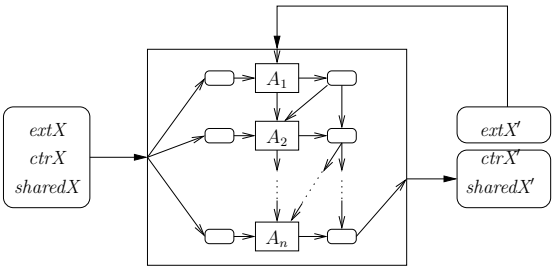
### Shared Refinement

We define shared refinement of SI at the semantics level. The obtained LTS represents the intersection of the observable traces of both original SI.

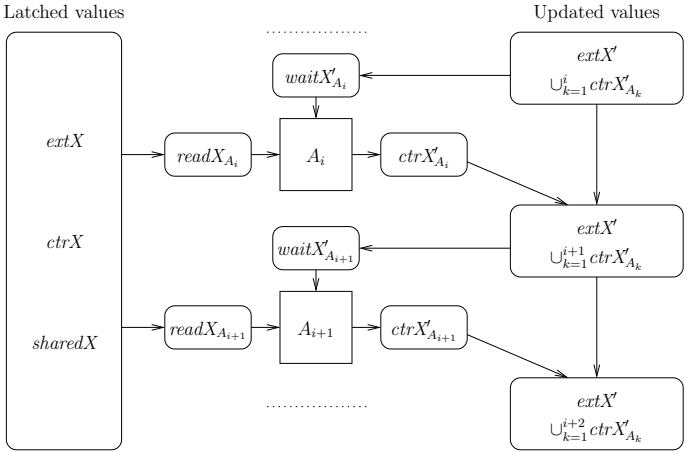
**Definition 8.** *Let $M$, $N$ be two SI with $[\![M]\!] = (S_M, S_M^0, \to_M)$ and $[\![N]\!] = (S_N, S_N^0, \to_N)$, and assume $\mathrm{extX}_M \cap \mathrm{intfX}_N = \mathrm{extX}_N \cap \mathrm{intfX}_M = \emptyset$. The* shared refinement *of $[\![M]\!]$ and $[\![N]\!]$ is the LTS $[\![M \wedge N]\!] = (S, s_0, \to)$ given by*

- *$S \subseteq V[X_M \cup X_N \cup \{m, n\}] \cup \{s_{init}\}$, with $m$ and $n$ two fresh Boolean variables, and such that $s \in S$ iff $s[X_M] \in S_M$ and $s[m] = \mathtt{t}$, or $s[X_N] \in S_N$ and $s[n] = \mathtt{t}$;*
- *$s_0 = s_{init}$ and $s_0[n] = s_0[m] = \mathtt{t}$*
- *for all $s, s' \in S$, $s \to s'$ iff one of the three following conditions holds:*
  *(1): $s[m] = s[n] = \mathtt{t}$ and*

$$\text{either } \begin{cases} s[X_M] \to_M s'[X_M] \wedge s'[m] = \mathtt{t} \\ s[X_N] \to_N s'[X_N] \wedge s'[n] = \mathtt{t} \end{cases}$$




(a) Execution of a SI




(b) Part of a round

Fig. 7: Illustrations of the execution of a SI and details about a round.

$$
or \begin{cases}
s[X_M] \rightarrow_M s'[X_M] \wedge s'[m] = \mathtt{t} \\
s'[\mathrm{intfX}_N \cup (\mathrm{ctr}_N(s'[\mathrm{extX}_N]) \cap \mathrm{ctr}_N(s[\mathrm{extX}_N]))] = \\
\quad s[\mathrm{intfX}_N \cup (\mathrm{ctr}_N(s[\mathrm{extX}_N]) \cap \mathrm{ctr}_N(s'[\mathrm{extX}_N]))] \\
s'[n] = \mathtt{f} \\
\forall t' \in V[X_N], \mathit{if}\ s[X_N] \rightarrow_N t',\ \mathit{then} \\
t'[\mathrm{extX}_N \cup (\mathrm{sharedX}_N \setminus \mathrm{ctr}_N(s'[\mathrm{extX}_N]))] \neq \\
\quad s'[\mathrm{extX}_N \cup (\mathrm{sharedX}_N \setminus \mathrm{ctr}_N(s'[\mathrm{extX}_N]))]
\end{cases}
$$

$$
or \begin{cases}
s[X_N] \rightarrow_N s'[X_N] \wedge s'[n] = \mathtt{t} \\
s'[\mathrm{intfX}_M \cup (\mathrm{ctr}_M(s'[\mathrm{extX}_M]) \cap \mathrm{ctr}_M(s[\mathrm{extX}_M]))] = \\
\quad s[\mathrm{intfX}_M \cup (\mathrm{ctr}_M(s[\mathrm{extX}_M]) \cap \mathrm{ctr}_M(s'[\mathrm{extX}_M]))] \\
s'[m] = \mathtt{f} \\
\forall t' \in V[X_M], \mathit{if}\ s[X_M] \rightarrow_M t',\ \mathit{then} \\
t'[\mathrm{extX}_M \cup (\mathrm{sharedX}_M \setminus \mathrm{ctr}_M(s'[\mathrm{extX}_M]))] \neq \\
\quad s'[\mathrm{extX}_M \cup (\mathrm{sharedX}_M \setminus \mathrm{ctr}_M(s'[\mathrm{extX}_M]))]
\end{cases}
$$

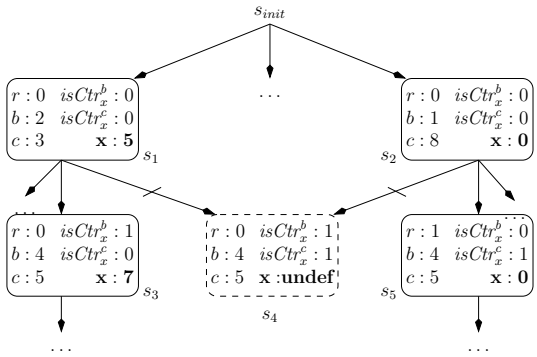



Fig. 8: Semantics $[\![\mathrm{M_{ex}}]\!]$ of the SI $\mathrm{M_{ex}}$

*(2):* $s[m] = \mathtt{t}$, $s[n] = \mathtt{f}$ *and*

$$\begin{cases} s[X_M] \to_M s'[X_M] \wedge s'[m] = \mathtt{t} \\ s[\mathrm{intfX}_N \cup (\mathrm{ctr}_N(s[\mathrm{extX}_N]) \cap \mathrm{ctr}_N(s'[\mathrm{extX}_N]))] = \\ \quad s'[\mathrm{intfX}_N \cup (\mathrm{ctr}_N(s'[\mathrm{extX}_N]) \cap \mathrm{ctr}_N(s[\mathrm{extX}_N]))] \\ s'[n] = \mathtt{f} \end{cases}$$

*(3):* $s[m] = \mathtt{f}$, $s[n] = \mathtt{t}$ *and*

$$\begin{cases} s[X_N] \to_N s'[X_N] \wedge s'[n] = \mathtt{t} \\ s[\mathrm{intfX}_M \cup (\mathrm{ctr}_M(s[\mathrm{extX}_M]) \cap \mathrm{ctr}_M(s'[\mathrm{extX}_M]))] = \\ \quad s'[\mathrm{intfX}_M \cup (\mathrm{ctr}_M(s'[\mathrm{extX}_M]) \cap \mathrm{ctr}_M(s[\mathrm{extX}_M]))] \\ s'[m] = \mathtt{f} \end{cases}$$

The intuition behind this construction is that $m$ and $n$ keep track of whether $M$ and $N$ still can move or previously have reached a deadlock. If no deadlock has been reached previously (1), then either both $M$ and $N$ take a transition, or only one does because the other has reached a deadlock (and then $n$ or $m$ are updated to $\mathtt{f}$). If $N$ previously has reached a deadlock, but $M$ still can take transitions (2), then this information is remembered in $s'[m]$ and $s'[n]$, and we demand (as we do in the deadlock parts of case (1)) that the variables belonging to $N$ do not change their values. Case (3) is symmetric to case (2).

The next theorem shows that shared refinement acts as a *greatest lower bound*: Any behavior admitted by $M$ and $N$ is also admitted by $M \wedge N$ and vice versa.

**Theorem 5.** *Given two SI $M$ and $N$, we have that: (1) $M \wedge N \leq M$; (2) $M \wedge N \leq N$; and (3) for all SI $M'$ such that $M' \leq M$ and $M' \leq N$, also $M' \leq M \wedge N$.*

*Proof.* The first two claims are clear by construction: if there is an environment in which $M$ (or $N$) admits a transition, then $m = \mathtt{t}$ (or $n = \mathtt{t}$) in the corresponding

state in $M \wedge N$ and the corresponding transition exists in $M \wedge N$. For the third claim, let $R_M \subseteq S_{M'} \times S_M$, $R_N \subseteq S_{M'} \times S_N$ be refinement relations and define $R \subseteq S_{M'} \times S_{M \wedge N}$ by

$$R = \{(S^0_{M'}, s_{init})\} \cup \{(v', v) \mid (v', v[X_M]) \in R_M, (v', v[X_N]) \in R_N, \\ m = \mathtt{t} \text{ iff } v[X_M] \rightarrow_M, n = \mathtt{t} \text{ iff } v[X_N] \rightarrow_N\}$$

Here we have used $v[X_M]$ to denote the restriction of the valuation $v$ to variables in $M$ and $v[X_M] \rightarrow_M$ to mean that $M$ is not deadlocked with valuations $v$.

To show that $R$ is a refinement relation witnessing $M' \leq M \wedge N$, let $(v', v) \in R$. If $v$ is not deadlocked, then by definition of $M \wedge N$ also $M$ or $N$ (or both) are not deadlocked in the corresponding valuations. Hence also $M'$ is not deadlocked in $v'$, and then any transition $v' \rightarrow_{M'} \tilde{v}'$ in $M'$ can be matched by corresponding transitions in $M$ and $N$, hence also in $M \wedge N$. □