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

# Compositional Verification of Real-Time Systems Using Ecdar

**Alexandre David and Kim. G. Larsen and Axel Legay and Mikael H. Møller and Ulrik Nyman and Anders P. Ravn and Arne Skou and Andrzej Wąsowski**

Received: date / Accepted: date

**Abstract** We present a specification theory for timed systems implemented in the Ecdar tool. We illustrate the operations of the specification theory on a running example, showing the models and verification checks. In order to demonstrate the power of the compositional verification we perform an in depth case study of a leader election protocol; Modeling it in Ecdar as Timed Input/Output Automata Specifications and performing both monolithic and compositional verification of two interesting properties on it. We compare the execution time of the compositional to the classical verification showing a huge difference in favor of compositional verification.

**Keywords** Timed Input/Output Automata · Compositional Verification · Real-Time Systems · Leader Election Protocol

Authors of this paper has been supported by MT-LAB, A VKR Centre of Excellence in Modeling of IT, and the Sino-Danish Basic Research Center IDEA4CPS.

Alexandre David
Computer Science, Aalborg University, Denmark
E-mail: adavid@cs.aau.dk

Kim G. Larsen
Computer Science, Aalborg University, Denmark
E-mail: kgl@cs.aau.dk

Axel Legay
INRIA/IRISA, Rennes Cedex, France
E-mail: axel.legay@irisa.fr

Mikael H. Møller
Computer Science, Aalborg University, Denmark
E-mail: mikaelhm@cs.aau.dk

Ulrik Nyman
Computer Science, Aalborg University, Denmark
E-mail: ulrik@cs.aau.dk

Anders P. Ravn
Computer Science, Aalborg University, Denmark
E-mail: apr@cs.aau.dk

Arne Skou
Computer Science, Aalborg University, Denmark
E-mail: ask@cs.aau.dk

Andrzej Wąsowski
IT University of Copenhagen, Denmark
E-mail: wasowski@itu.dk

## 1 Introduction

Programs are intrinsically component based, they are built from simple commands, and when we reason about their correctness, we intuitively think in terms of what we can *assume* about the program state before the command is performed and what it *guarantees* about the state afterwards. This simple fact was formalized early on in terms of Floyd assertions [18] and led to Floyd-Hoare logic [20] and is really the foundation of program verification, which led to much fruitful research in the following years. In particular, the challenge of compositional analysis of concurrent programs was pursued first in 1976 by Owicki and Gries [32], who extended Floyd-Hoare logic to parallel programs with shared variables, and later in 1981 by Jones [23], who introduced the rely-guarantee method, allowing for a compositional version of the Owicki-Gries method. Yet, there are larger components in software applications: subroutines from libraries, classes in object-oriented languages, service modules in service-oriented architectures, control modules in embedded systems, etc.

Common for such larger scale components are the characteristics made explicit by Szyperski [36]:

> *a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is a subject to third party composition.*

We shall not consider deployment or component implementation. These are interesting questions, but we get to grips with *composition* at the level of *interfaces*, because this is essential for getting a useful product out of gluing components together and deploying them. Interfaces are essentially specifications of what we assume about the environment of the component and what the implementation guarantees to deliver. In order to have a good theory for reasoning about component interface specification, we expect that for a given specification, we can determine:

*Consistency.* When a specification is satisfied by at least one implementation it is consistent. Consistency is needed to verify that specifications are well-formed and do not contain contradictory statements. Without consistency, we can specify miraculous components which no one can deliver.

*Conjunction.* Specifications are essentially logics, and when composing them using conjunction this should give exactly the intersection of feasible implementations of the constituents. Should the intersection be empty, that is, the conjunction is not consistent, it is useless to put those components together.

*Composition.*When actual components are deployed together they form a new composite component. A similar parallel composition operation is needed for their specifications in order to build systems in a stepwise manner.

*Refinement.*There is a natural partial order on components defined by replacement of one by another while maintaining the functionality of the system as a whole. When such substitutions are possible, the more detailed and constraining specification *refines* the one for the component that is replaced.

Specification theories with refinement were pioneered by Jones [24] in a setting of sequential program components and it has lead to further development of such theories, most recently in the area of object-oriented programming with design by contract and for instance the Java Modeling Language (JML) [28].

However, since we wish to deal with a context of distributed, communicating components, a specification theory with the state given by program variables is not well suited. Specifications for such systems are better built on process algebras [5] and their underlying transition system semantics. Transition systems are also intimately linked to automata models. Since transition systems generate traces of events or actions, specification logics describe properties of traces, and here a very liberal use of assumptions and guarantees may lead to unsound reasoning. Essentially a guarantee can specify that the past is changed to fit an assumption, or an assumption can speak about a future that the guarantee contradicts. This was investigated by Abadi and Lamport [1]. However, since the specification formalism employed here is automata based, it does not suffer from these anomalies.

An interesting question with (parallel) composition of components is whether one can find a strongest specification for an unknown component that composes with a given one to give a desired result. It is the question of finding a *quotient* or a weakest prespecification. This can be done for the current theory, a result that originates in [26,27,4]. Similar results in a logic based refinement theory are found in [21], although this solution is more a proof of existence than an actual construction.

### 1.1 Related Work

In a series of recent work, it has been advocated that specifications can be represented by interface automata, that are automata 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 [30], 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 [16], 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. In this paper, we represent specifications by timed input/output automata [25], i.e., timed automata whose sets of discrete transitions are split into Input and Output transitions. Contrary to [16] and [25] we distinguish between implementations and specifications by adding conditions on the models. This is done by assuming that implementations have fixed timing behavior and they can always advance either by producing an output or delaying. Also, we provide a game-based methodology to decide whether a specification is consistent, i.e. whether it has at least one implementation. An implementation exists when there is a strategy that despite the behavior of the environment will avoid states that cannot possibly satisfy the implementation requirements.

Our theory is rich in the sense that it captures all the good operations for a compositional design theory discussed above. Also, all the algorithms have been implemented in the Ecdar tool set. This implementa-

tion (available at `ecdar.cs.aau.dk`) is build on top of the Uppaal-tiga tool-set [7]. Uppaal-tiga is a tool that implements a series of algorithms for solving timed games [10] as well as checking timed temporal logic properties. Ecdar uses Uppaal-tiga to solve various games that arise in computing the composition operations and refinements.

The first part of the paper presents an overview of the theory implemented in the Ecdar tool set. The second, and maybe most interesting part of the paper, applies Ecdar theory to a leader election protocol. More precisely, we show how compositional design can be used to check two important properties of the protocol in an incremental manner, outperforming classical model checking techniques for timed automata that are working on the entire system directly. The incremental approach used is based on the concept of independent implementability [15], in which a specification can be refined into a more detailed specification independently of what it is composed with. This method is correct because our refinement operator is a precongruence with respect to parallel composition [13].

Another tool supporting refinement is PAT [34,35]. Unlike Ecdar, it builds on CSP with a failures, divergences and refusal semantics which makes a direct comparison difficult. However, the CSP theory does not support quotienting nor simple conjunction of specifications. And thus in contrast to Ecdar, PAT does not support assume/guarantee reasoning about systems.

### 1.2 Structure

The rest of the paper falls in three parts: Theory, Case-study and Conclusion. The theory is presented in Section 2 on page 3. The theory with its definitions is included in order to make the paper self-contained. Theorems and proofs can be found in [13]. The case-study, a leader election protocol, is presented in Section 3 on page 10. While conclusion and future work is given in Section 4 on page 16.

## 2 Timed Input/Output Automata Specifications

Before we proceed to discuss our case study, let us present the main concepts and constructions of the specification theory for real time systems supported by the Ecdar tool. We only focus on the designer-facing aspects of the framework. A reader interested in the theoretical discussions is referred to [13].

The main concept in our modeling framework is that of a specification—an abstract, usually under-specified, description of an implementation of a system. Each specification normally admits multiple implementations that can be derived by different resolutions of detailed design choices.

We use the syntax of *Timed I/O Automata* (TIOA) to represent specifications. We will now recall their definition and only then proceed to define specifications themselves along with a notion of satisfaction of a specification by an implementation, notion of refinement between specifications, and the compositional design operators that allow manipulating and combining specifications.

TIOAs are essentially the usual Timed Automata [2] extended with two types of edges: inputs and outputs. Input edges are drawn as solid arrows labeled by actions followed by a question mark. Output edges are dashed and their actions are suffixed with an exclamation point. Fig. 1 shows an example of a TIOA describing the main research process at a hypothetical university, that, given grants as inputs produces patents as outputs.

The kind of communication an automaton can engage in is limited by its *sort*—a signature of available input and output actions. In Fig. 1, the sort is depicted as incoming arrows (inputs) and outgoing arrows (outputs) incident with the border surrounding the automaton. The initial location is indicated by a doubly circled outline. In this initial location, after the university receives the `grant` input, it will output a `patent`.

The colors used in the figures do not carry semantic meaning but are used consistently in order to increase the readability of the models. These colors—guards (green), resets (navy blue), invariants (violet), and actions (turquoise)—are the same as used in the editor of Ecdar and in related tools such as Uppaal.

Additional labels on edges denote timing constraints over *clocks* (known as guards) and clock *resets*. For example, the grant must be received before the clock $\mathsf{u}$ exceeds two time units ($\mathsf{u} \leq 2$). This clock is reset immediately upon reception of the `grant` ($\mathsf{u} = 0$). Then the patent is issued within 20 time units, as the automaton can only reside in the target location for twenty time units as indicated by the *location invariant* $\mathsf{u} \leq 20$. Any further grants received within this time interval are ignored through the `grant` input loop that has no guard and no resets. When the `patent` is issued the clock $\mathsf{u}$ is again reset.

If the first grant arrives after more than two time units, or if any subsequent grant arrives later than two time units after a patent has been filed, then the behavior of the university automaton becomes *unpredictable*. This is captured by the leftmost location in the figure, a so called *universal* location, in which any communica-

tion can appear at any time; the location has outgoing edges for any available action and imposes no timing constraints. Strictly speaking the behavior of the automaton is still completely specified, but since it provides no guarantees about its output in a *universal* location we also call this unpredictable.

Let us now recall a formal definition of a TIOA:

**Definition 1** 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 with $\mathcal{B}(Clk)$ being a set of clock constraints, $\mathscr{P}(Clk)$ is the set of clocks to reset, $Act = Act_{\mathrm{i}} \uplus Act_{\mathrm{o}}$ is a finite set of actions, partitioned into inputs and outputs respectively, and $Inv : Loc \mapsto \mathcal{B}(Clk)$ is a set of location invariants.

As we have intuitively sketched above, TIOA syntax has a semantic interpretation as a timed execution of a branching process. This is formally captured by a *Timed I/O Transition System* (TIOTS), which is like a usual discrete automaton but infinitely branching and over an infinite state space. In a TIOTS, time delays are modeled as continuously many 'discrete' actions.

**Definition 2 (TIOTS)** A Timed I/O Transition System (TIOTS) is a quadruple $S = (St^S, s_0, \Sigma^S, \rightarrow^S)$, where $St^S$ is an infinite set of states, $s_0 \in St$ is the initial state, $\Sigma^S = \Sigma^S_{\mathrm{i}} \oplus \Sigma^S_{\mathrm{o}}$ is a finite set of actions partitioned into inputs ($\Sigma^S_{\mathrm{i}}$) and outputs ($\Sigma^S_{\mathrm{o}}$) and $\rightarrow^S : St^S \times (\Sigma^S \cup \mathbb{R}_{\geq 0}) \times St^S$ 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 $\mathbb{R}_{\geq 0}$ respectively. We sometimes omit the transition system name ($s \xrightarrow{a} s'$) if obvious from the context and we omit the target location ($s \xrightarrow{a}{}^S$) if we only need to know the existence but not the identity of the target location. 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 $d_1, d_2 \in \mathbb{R}_{\geq 0}$ 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 $s' \in St^S$

A *state* of the TIOTS derived from a TIOA $A$ is a pair $(q, V)$ where $q$ is a location and $V : Clk \mapsto \mathbb{R}_{\geq 0}$ is a *valuation function* that assigns a non-negative value to each clock in $Clk$. We use $u, u'$ to range over clock valuations. We write $u + d$, where $d \in \mathbb{R}_{\geq 0}$ is a delay, to denote a valuation such that for any clock $r$ we have $(u + d)(r) = x + d$ iff $u(r) = x$. Given $c \subseteq Clk$, 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 $c$. We use $\mathbf{0}$ to denote the constant function mapping all clocks to zero. The *initial state* of $A$ is the pair $(q_0, \mathbf{0})$.

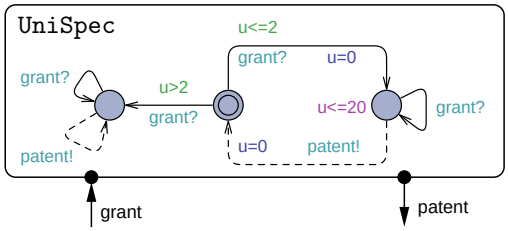


**Fig. 1** University specification `UniSpec`.

The semantics of a TIOA $A = (Loc, q_0, Clk, E, Act, Inv)$ is a TIOTS $[\![A]\!]_{\mathrm{sem}} = (Loc \times (Clk \mapsto \mathbb{R}_{\geq 0}), (q_0, \mathbf{0}), Act, \rightarrow)$, where $\rightarrow$ is the transition relation defined 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 $u \in [Clk \mapsto \mathbb{R}_{\geq 0}]$ 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 $u \in [Clk \mapsto \mathbb{R}_{\geq 0}]$ gives rise to a transition $(q, u) \xrightarrow{d} (q, u + d)$ for each delay $d \in \mathbb{R}_{\geq 0}$ such that $u + d \models Inv(q)$.

We only consider deterministic TIOAs, so TIOAs whose semantics results in a deterministic TIOTS: for each action–state pair at most one action is enabled.

## 2.1 Specifications

We now define specifications in terms of TIOAs.

**Definition 3** A specification automaton is a TIOA that is input-enabled, i.e., in each state all the inputs should be available at all times.

The assumption of input-enabledness, also seen in many interface theories [29,19,33,37,31], 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. The idea is actually quite old, and can be traced to the notion of a CHAOS process in CSP [22].

Input-enabledness encourages explicit modelling of unpredictability, and compositional reasoning about it; for example, deciding if an unpredictable behavior of one component induces unpredictability of the entire system. Observe that it is easy to check whether a TIOA is input-enabled. 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.

We note that our example of Figure 1 can always accept `grant?` from any location. It is also deterministic. Thus `UniSpec` TIOA is a well-formed specification.

### 2.2 Implementations

The role of specifications in a specification theory is to abstract, or under-specify, sets of possible implementations. *Implementations* are concrete executable realizations of systems. We will assume that implementations of timed systems have fixed timing behavior (outputs occur at predictable times) and systems can always advance either by producing an output or delaying. An implementation that cannot voluntarily output or delay would have to block passage of time, which is not realistic.

**Definition 4** An implementation $P$ is a specification whose underlying TIOTS satisfies the following conditions:

1. `Independent progress:` in each state either an output is possible or one can delay until an output is enabled.
   either $(\forall d \geq 0.\, p \xrightarrow{d}{}^{P})$ or
   $\exists\, d \in \mathbb{R}_{\geq 0}.\, \exists\, o! \in \Sigma_{\mathrm{o}}^{P}.\, p \xrightarrow{d} p'$ and $p' \xrightarrow{o!}{}^{P}$.
2. `Output urgency:` an available output cannot be delayed:
   $\forall\, p', p'' \in St^{P}$ if $p \xrightarrow{o!}{}^{P} p'$ and $p \xrightarrow{d}{}^{P} p''$ then $d = 0$ (and consequently, due to determinism and time reflexivity we have $p = p''$)

*Example.* Figure 2(a) specifies a vending machine that can serve tea or coffee. We will use this as a component in our example. A possible implementation of this machine can be found in Figure 2(b). The implementation *refines* the specification, which is defined in the next section. Both automata are deterministic. Note that the output transitions of the implementation `Impl` arrive at a fixed moment in time and cannot be delayed, which guarantees output urgency (the invariant guarantees progress and the guard constrains the transition). Each time the output `tea!` from `Idle` to `Idle` is taken, the clock `y` is reset. Without this reset, independent progress would not be guaranteed for valuations of the clock `y` that are greater than 6.

### 2.3 Satisfaction and Refinement

*Refinement* is always a pivotal element of a specification theory. Akin to entailment for logical specifications, refinement allows to start with very abstract models, and elaborate them towards more specific ones. An early *abstract* specification would typically allow a large set of diverse implementations. This set is monotonically reduced in a stepwise refinement process towards a detailed, more fine grained and *concrete* specification that can be implemented directly.

Any refinement should satisfy the following *substitutability* condition: If $A_S$ refines $A_T$, it should be possible to replace $A_T$ with $A_S$ in every context and obtain a safe system. It is well known from the literature [14, 15, 8] that in order to give these kind of guarantees a refinement should have the flavor of *alternating (timed) simulation* [3].

In our theory we define the refinement between specifications, by requiring a suitable refinement relation in their semantic expansion (TIOTS).

**Definition 5 (Refinement relation)** Let $A_S$ and $A_T$ be two specification automata and $S = (St^S, s_0, \Sigma, \rightarrow^S)$ and $T = (St^T, t_0, \Sigma, \rightarrow^T)$ be their corresponding timed transition systems. We say that $A_S$ refines $A_T$, written $A_S \leq A_T$, iff there exists a binary relation $R \subseteq St^S \times St^T$ containing $(s_0, t_0)$ and for all states $sRt$ implies:

1. Whenever $t \xrightarrow{i?}{}^{T} t'$ for some $t' \in St^T$ then $s \xrightarrow{i?}{}^{S} s'$ and $s'Rt'$ 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 $s'Rt'$ for some $t' \in St^T$
3. Whenever $s \xrightarrow{d}{}^{S} s'$ for $d \in \mathbb{R}_{\geq 0}$ then $t \xrightarrow{d}{}^{T} t'$ and $s'Rt'$ for some $t' \in St^T$

Intuitively, if $A_S$ refines $A_T$ then it can delay at most as much as $A_T$ can, and it can only produce outputs that $A_T$ produces—not others. It, however, may admit more inputs than $A_T$, as long as all $A_T$'s inputs are handled. This construction ensures substitutability, because then, if placed in the same context, $A_S$ will engage in less computations than $A_T$, while maintaining ability to always receive the same inputs. This means that safety properties will be preserved.

In the example of Figure 2, `Machine2` (c), `Machine3` (d), and `Machine4` (e) refine `Machine` (a). `Machine6` (f) refines both `Machine3` (d) and `Machine4` (e). `Machine7` (h) refines `Machine4` (e).

Definition 5 is non-constructive in the sense that it cannot be directly used to decide refinement between two automata. Discussion of a proper efficient refinement checking algorithm is out of scope for this work. See [8, 13] for details.

We relate specifications to implementations using a notion of *satisfaction*. A proper implementation of a specification is said to satisfy it. Technically, in our framework the satisfaction is a special case of the refinement, when the left hand side is an implementation (it

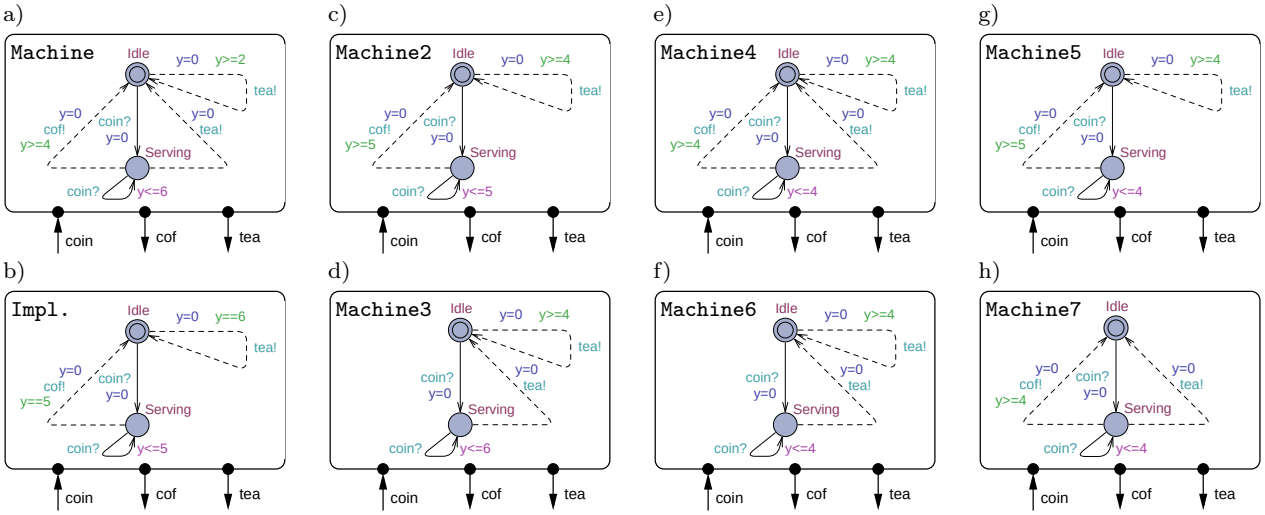


**Fig. 2** a) Specification of a coffee/tea `Machine`, b) an implementation, and c) d) e) f) g) h) more specifications of a coffee/tea machine.

satisfies independent progress and output urgency—see Def. 4).

The set of all implementations of $A$ is denoted $[\![A]\!]_{\text{mod}}$. In [13], we have shown that the refinement relation is complete for our implementation model, i.e., $A_S$ refines $A_T$ if and only if the set of implementations that satisfy $A_S$ is included in the set of implementations that satisfy $A_T$. This is an important usability criterion for tools. It means that if you indeed elaborated $A_T$ into $A_S$ such that any implementation of the latter implements the former, the tool will never report a false positive when checking $A_S \leq A_T$.

*Consistency.* It can happen that a specification cannot be implemented, for example, because it enforces reachability of a stuck state, which violates independent progress. As all implementations satisfy independent progress, they can never satisfy such a specification. We say that a specification which admits at least one implementation is (globally) *consistent*. For example coffee machine of Figure 2, the implementation 2(b) refines 2(a). Since 2(a) admits at least one implementation, it is a consistent specification.

In the example of Figure 2, `Machine5` (g) is in fact inconsistent since, in the state `Serving` no output is available and time cannot diverge, thus violating independent progress.

Inconsistency of a specification in a stepwise design process is normally unintended—an error on behalf of the specifier. Thus it is important for tools to provide feedback on consistency. In [13], we have shown that this question can be answered automatically using an algorithm that decides if there exists a strategy for the system (output) to avoid reaching stuck states in the specification. Furthermore we added a facility called *pruning* that removes from the TIOA all behaviors that are not covered by such a maximal strategy. Pruning thus reduces the size of the TIOA specification by removing inconsistent parts, while maintaining the same set of implementations (Theorem 5 in [13]).

### 2.4 Step-wise Refinement

We decompose and refine our `University` specification of Figure 1 in a top-down manner. The refinement is based on a knowledge of how the system under design is supposed to meet the overall requirements. We decompose our specification into three components in parallel: a Coffee/Tea machine, a Researcher, and an Administration. The `Machine` (Figure 2(a)) needs coins to function and provides the Researcher with coffee and tea. In addition it may offer tea for free. The `Researcher` (Figure 3(a)) produces publications with some guaranteed timing constraints when provided with coffee and tea regularly, otherwise the publication output is not guaranteed any more. The `Administration` is in charge of turning grants into coins to enable the use of the `Machine` *and also* to file patents when publications are produced by the `Researcher`. We could make one TIOA to specify this behavior but it is naturally expressed as a conjunction and making this TIOA manually is error prone. Instead we specify our `Administration` as a conjunction of `HalfAdm1` and `HalfAdm2`, each in charge of one of the tasks. Figure 3(b) shows the alternation between `coin!` and `grant?` while Figure 3(c) shows the alternation between `patent!` and `pub?`. We note that since both automata are parts of the administration, `HalfAdm1` always allows `patent!` and `HalfAdm2` always

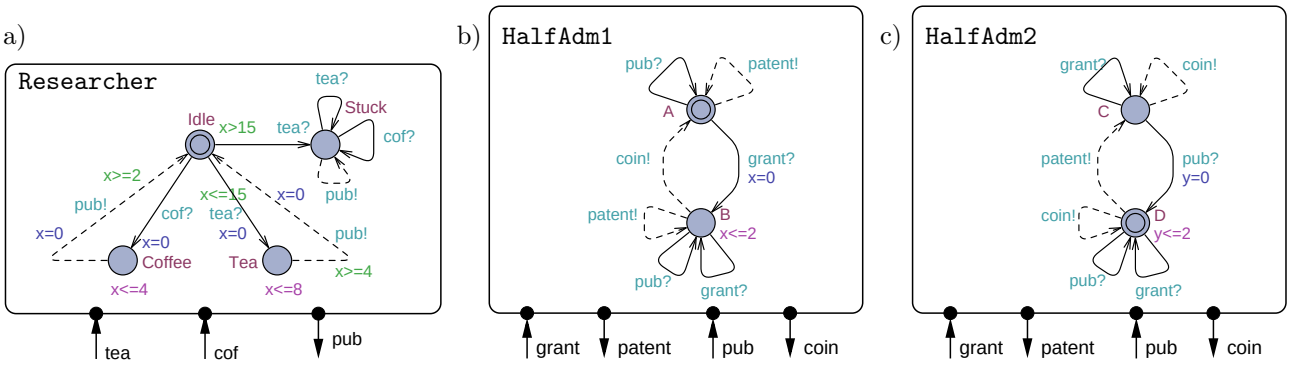


**Fig. 3** Specification of (a) the `Researcher`, and the `Administration` as a conjunction of two components (b) `HalfAdm1` and (c) `HalfAdm2`.

allows `coin!`. Both sub-specifications are also input-enabled and can always accept `grant?` and `pub?`.

Verification of this refinement is carried out step-wise using *pruning* at every step. In this example, the components are checked for consistency individually and pruned to valid behaviors. Then they are combined step-wise, first with the conjunction operator (explained later), the result being pruned, and then with the composition operator, and then pruned. The resulting state graphs for both specifications are finally checked for refinement.

The result here is that the refinement does not hold, which may seem surprising. It turns out that the original specification of Figure 1 does not allow for "free" patents: grants must be received before a patent is produced. However, given that the `Machine` can produce free tea, free publications may appear, and therefor free patents as well, which was not specified. It is possible to correct this by either allowing for free patents or removing free tea in the `Machine`.

In the following sections we will elaborate how the specifications are composed in the framework.

## 2.5 Combining Specifications

In our example we used parallel composition and conjunction intuitively. Now we give more details on all available operators, namely parallel composition, conjunction, and quotient. In the rest of the section, we will consider two specification automata $A_S = (Loc_1, q_0^1, Clk_1, E_1, Act^1, Inv_1)$ and $A_T = (Loc_2, q_0^2, Clk_2, E_2, Act^2, Inv_2)$. For technical reasons, we also assume that $Clk_1 \cap Clk_2 = \emptyset$.

There are two main ways of composing specifications in our framework: conjunction and parallel composition. The latter is the well known structural combination of components—parallel composition is meant to combine specifications of *two separate* interacting components into a single box. In our example the `Researcher` specification is composed with the beverage dispensing `Machine` specification in this manner.

The other operator, conjunction, is meant to combine two different specifications for the *same* component. The two specifications can typically represent requirements from a different viewpoint. In our example `HalfAdm1` represented requirements with respect to providing funding (coins); `HalfAdm2` represented requirements on producing patents.

*Conjunction.* In our framework, conjunction can only be defined if $Act_i^S = Act_i^T$ and $Act_o^S = Act_o^T$ (the extension to dissimilar alphabets is straightforward). The operation reduces to check whether the two specifications can progress in the same way. Formally, the conjunction of $A_S$ and $A_T$, denoted $A_S \wedge A_T$, is the TIOA $A = (Loc, q_0, Clk, E, Act^S, Inv)$ given by: $Loc = Loc_S \times Loc_T$, $q_0 = (q_0^S, q_0^T)$, $Clk = Clk_S \uplus Clk_T$, $Inv((q_S, q_T)) = Inv(q_S) \wedge Inv(q_T)$. The set of edges $E$ is generated by the following rule:

$$\frac{(q_S, a, \varphi_S, c_S, q'_S) \in E_S \qquad (q_T, a, \varphi_T, c_T, q'_T) \in E_T}{((q_S, q_T), a, \varphi_S \wedge \varphi_T, c_S \cup c_T, (q'_S, q'_T)) \in E}$$

The conjunction operator may introduce locally inconsistent states. For example, assume that $A_S$ reaches a state from $s$ where the only available action is the output $a$ and $A_T$ reaches a state $t$ from where the only available action is the output $b$. Assume also that $A_S$ and $A_T$ cannot delay in $s$ and $t$. In $(s, t)$, the conjunction will not issue any output and will not be able to delay, which violates the *independent progress* property. As stated above the locally inconsistent states are removed by Ecdar using the pruning facility.

In the example of Figure 2, `Machine5` (g) is a conjunction of `Machine2` (c) and `Machine4` (e) (though it is an inconsistent conjunction). Furthermore, `Machine6` (f) is a conjunction of `Machine3` (d) and `Machine4` (e).

*Parallel Composition.* This operation computes the classical product between timed specifications [25], where components synchronize on common inputs/outputs. Two components are *composable* iff the intersection between their output alphabets is empty.

Formally, the *parallel composition* of $A_S$ with $A_T$, denoted $A_S||A_T$, is the TIOA $A = (Loc, q_0, Clk, E, Act, Inv)$ given by: $Loc = Loc_S \times Loc_T$, $q_0 = (q_0^S, q_0^T)$, $Clk = Clk_S \uplus Clk_T$, $Inv((q_S, q_T)) = Inv(q_S) \wedge Inv(q_T)$ and the set of actions $Act = Act_i \uplus Act_o$ is given by $Act_i = Act_i^S \setminus Act_o^T \cup Act_i^T \setminus Act_o^S$ and $Act_o = Act_o^S \cup Act_o^T$. The set of edges $E$ is generated by the following rules:

1. Whenever $(q_S, a, \varphi_S, c_S, q'_S) \in E_S$ with $a \in Act_S \setminus Act_T$ then for each $q_T \in Loc_T$ also $((q_S, q_T), a, \varphi_S, c_S, (q'_S, q_T)) \in E$
2. Whenever $(q_T, a, \varphi_T, c_T, q'_T) \in E_T$ with $a \in Act_T \setminus Act_S$ then for each $q_S \in Loc_S$ also $((q_S, q_T), a, \varphi_S, c_S, (q_S, q'_T)) \in E$
3. Whenever $(q_S, a, \varphi_S, c_S, q'_S) \in E_S$ and $(q_T, a, \varphi_T, c_T, q'_T) \in E_T$ with $a \in Act_S \cap Act_T$ then also $((q_S, q_T), a, \varphi_S \wedge \varphi_T, c_S \cup c_T, (q'_S, q'_T)) \in E$.

The first rule represents all the cases where $A_S$ makes an individual move, be it input or output, because $a$ is not in the signature of $A_T$. Similarly the second rule handles all individual moves by the second component $A_T$. The third rule handles all synchronizations between the two components. The possibilities are input/input which again gives an input or input/output which gives an output.

*Quotient.* The operation of *quotienting* is radically different from the other composition operators. It is a differencing operator [17] that can be used to synthesize requirements for missing components in a project. Two fix attention, let's assume that we have an abstract specification $A_T$ for the entire system, and a specification $A_S$ of an existing available component. The quotient synthesizes a specification $A_T \backslash\backslash A_S$ for the missing component—the component that when composed with $A_S$ would implement $A_T$.

The use of quotient simplifies independent design of components. Assume that $X$ is the missing component that needs to be designed by another person, or even another vendor than the rest of the system. The correctness requirement for $X$ is $A_S||X \leq A_T$. In general this requirement might be a rather complicated verification expression. Fortunately, it is sufficient to separate the concerns using quotienting. The new designer does not need to have access to the entire system, nor does he need to perform the verification of the entire system each time he checks his current design for $X$. It suffices to synthesize the quotient $A_T \backslash\backslash A_S$ and he can simply check whether $X \leq A_T \backslash\backslash A_S$. This latter specification effectively captures all contextual requirements for $X$.

Summarizing, quotienting allows for factoring out behavior from a larger component. If one has a large component specification $A_T$ and a small one $A_S$ then $A_T \backslash\backslash A_S$ is the specification of exactly those components that when composed with $A_S$ refine $A_T$.

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

The set of edges $E$ is generated by the following rules:

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

Just like conjunction, the quotient operation may produce (locally) inconsistent specifications. Hence, each quotient operation is followed by pruning.

In the following we will illustrate the quotienting through a very simple example. The example consists of three Timed Input/Output Automata Specifications as shown in Fig. 4. We start with a simple specification, shown in Fig. 4a) of a system with two buttons. The specification states that as long as only `button1` is pressed then only `good` output will be produced. If at some point `button2` is pressed then the system could start to produce `bad` output.

The following definition defines an operator known as weaken or weakening, that is used for easier specification of assume guarantee specifications.

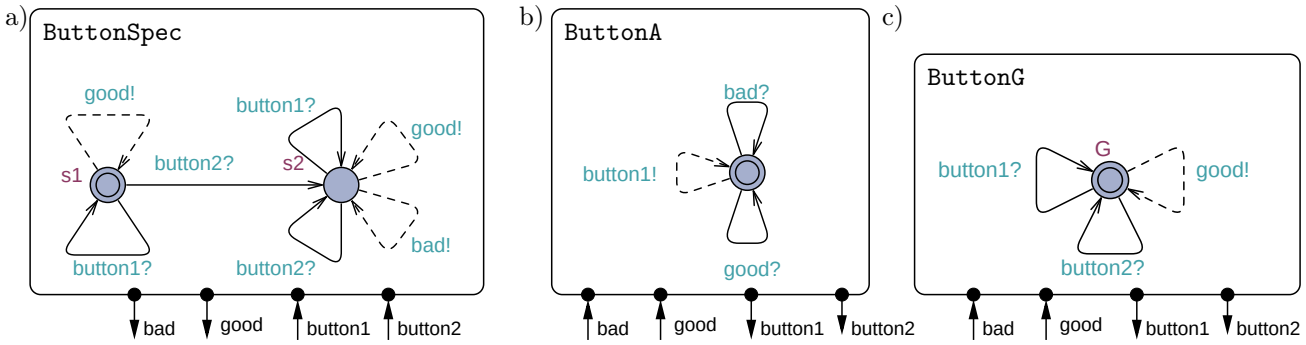


**Fig. 4** Specification of (a) the `ButtonSpec`, (b) the assumption `ButtonA` (c) the guarantee `ButtonG`.

**Definition 6** Weaken $>>$:
For any two Timed Input/Output Automata specifications $A$ and $G$ we define $G >> A$ as follows:

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

In our simple example we would like to express the assumptions and guarantees that we have to the system separately. In Fig. 4b) we specify the assumption that `button2` is never pressed while in Fig. 4c) we specify the guarantee that the system never produces `bad` output. Even though, in this example, our `ButtonSpec` is quite simple the assumption `ButtonA` and guarantee `ButtonG` are even simpler and extremely easy to understand.

For this example we can use Ecdar to prove the following two refinements:

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

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

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

## 2.6 Syntactic Extensions

The Ecdar tool offers a range of syntactic extensions build over the core language described above. These extensions do not affect the theoretical expressiveness of the language, but instead they enable more natural description of systems using primitives such as finite domain types, variables, constants, channels, committed locations, and arrays. These are the same extensions as known from Uppaal, but adapted to the two player semantics.

*Types, variables and constants.* Ecdar allows to introduce finite domain variables ranging over restricted integer types. The variables are more concise descriptions of counters and value placeholders than finite state machines. Named constants allow easy parameterization of models, for example with allowed delays.

*Channels and arrays.* Actions are defined using the syntax: "`broadcast chan a`" which gives both the input label `a?` and the output label `a!`. Actions are, as defined in the theory, broadcast and thus outputs are never blocked.

Channels can be organized in arrays. This is very convenient to encode local communication—for example a two dimensional $n \times n$ array of channels can model individual two-ended channels between $n$ processes.

*Select statements.* The modeling language of Ecdar also allows for using select statements of the form `e:id_t` on a transition. This translates into a set of transitions with `e` having each of the possible values that the type `id_t` can assume. This is only syntactic sugar which allows for much more compact models.

*Templates.* Templates are specifications parameterized with named but unresolved constants. Templates can be instantiated by providing values for constants, and the semantics is given by macro expansion. Templates are useful for instantiating many similar processes, perhaps with different initial conditions. They interplay well with constants and channel array.

Instantiating templates allows not only to change timing properties, but also to configure various communication topologies. For example, parameterize the template with the name (index in an array) of a channel to be used for communication. Then instantiate the parameters so that the instances create trees, rings, and other layouts. We will use this technique to model rings in the case study in the following section.

## 3 The Leader Election Protocol

We analyze a variant of the leader election protocol that operates on a ring topology. The protocol can be instantiated for an arbitrary number of nodes. Each node in the ring has both a place in the ring represented by its *id* and, apart from this, also a unique *priority*. The protocol performs one round of leader election selecting the node with the highest *priority* as the leader. When the protocol is initiated all nodes know that the election has started and can thus start to send their own *priority* to the next node in the ring topology. Figure 5 illustrates an instantiation of the protocol for six nodes, with their initial priorities and the communication channels used between the nodes. If a node receives a *priority* that is lower than its own *priority* it will just discard the received *priority*. If it receives a *priority* that is higher than its own *priority* it will keep a copy of the new *priority* and then send it on at the same time stopping to send its own *priority*. If at some point a node receives its own *priority*, it will know that it is the leader, since this *priority* has traveled one full round on the ring topology without being discarded and thus is greater than all other priorities.

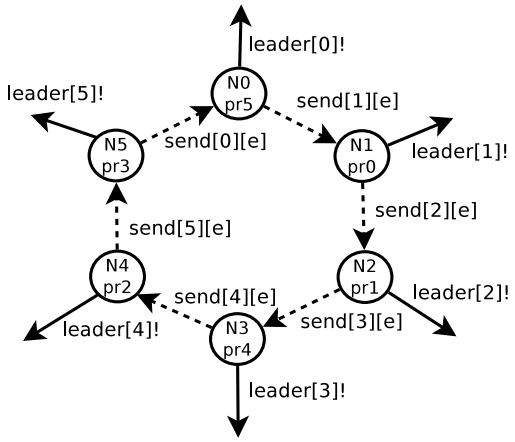


**Fig. 5** Overview of the ring topology and communication channels in a ring with 6 nodes. Each node has both an *id* given by its name (e.g. N0) and a priority (e.g. pr5). Between each set of nodes in the ring there is a set of communication channels used to mimic value passing.

The execution of the protocol is illustrated in Fig. 6 which shows how the information flows in a ring of 6 nodes, in the case where all nodes just happen to send the information at exactly the same time (synchronously).

We proceed to specify the protocol using Timed I/O Automata in the ECDAR tool. Let `N` be a constant that determines the number of nodes in the ring.

```
const int N = 6;
```

We also declare a constant for the maximum delay before a node sends the maximal *priority* that it has seen to the next node in the ring.

```
const int MaxD = 2;
```

Finally we declare a data type `id_t` which is used for all the variables containing *id*s and *priorities*.

```
typedef int[0,N-1] id_t;
```

Using the constant `N` we declare two global arrays of channels that are used to communicate the information in the model.

```
broadcast chan send[N][N];
broadcast chan leader[N];
```

The `send` channel is actually an array of `N` by `N` channels. In the channel expression `send[4][3]!` the first index (in this case node number 4) represents the *id* of the node that is the receiver of the message. The second index (in this case 3) represents the priority *pr* that is being send as the message. This is the standard way of modeling value passing in Timed (I/O) Automata.

### 3.1 Specification model for the nodes

Figure 7 shows the template for specifying the nodes. Each node is instantiated with an identifier `id` and a priority `pr`. Each node uses a local variable `cur` of type `id_t` to store the current priority value, initialized with the value of the `pr` constant:

```
id_t cur := pr;
```

The node consists of three locations. The top location which is also the initial location represents the normal operation of the protocol. This state has an invariant `x<=MaxD` ensuring that the node will send the maximal *priority* that it has seen so far, stored in the local variable `cur` to the next node in the ring with intervals of no more that `MaxD` time units.
Each node receives on the set of channels `send[id][e]?` where *e* can be any priority. Similarly it sends on a set of channels `send[(id+1)%N][e]` to the next node in the ring (the `%` is the modulus operator). On a given edge in the template, say the top leftmost one in Figure 7, the select statement `e:id_t` semantically translates into the instantiated template being able to receive any priority which is then bound to the variable `e`.

The node template has three input transitions in its initial location. The one leading to the second location is taken exactly in the case where the priority received matches the priority of the node itself. If this transition


**Fig. 6** Illustration of one scenario of how the information could be passed around the ring using the protocol. For the sake of illustration every node happens to send the information to the next node at exactly the same time thus giving us six distinct steps. Notice that the maximum *priority* will travel exactly once around the ring. In this case giving a total of 30 messages.

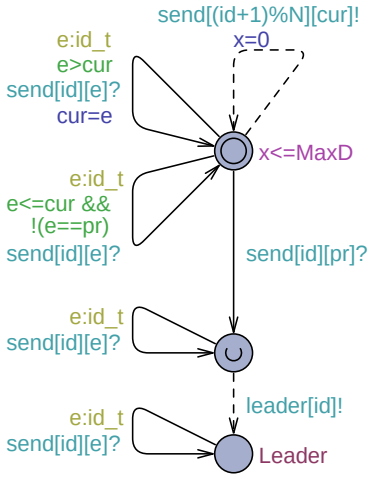


**Fig. 7** Node template used for each of the nodes in the ring topology.

is taken the node will declare itself leader. The other two represents the two cases where the local variable `cur` should be updated or not.

Both the second and third location are input enabled but does nothing with the input. The second location, marked with a $u$ meaning that it is urgent, will immediately send out the `leader[id]!` output.

### 3.2 Verification

The correctness of a ring of **N** nodes we are interested in has both a functional part—i.e. the correct leader is elected—as well as a non-function part—i.e. the leader is elected within an acceptable upper time bound. For this we formulate and verify the two general properties elaborated below.

The first property $\mathcal{S}$ , shown in Fig. 8, states that only the correct node, the one with the lowest priority, can declare itself leader.

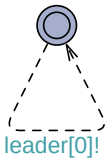


**Fig. 8** The most basic specification $\mathcal{S}$ stating that only the correct node declares itself leader.

The second property $\mathcal{T}$, shown in Fig. 9, states that a leader will be elected within `x<=(N+1)*MaxD` time units, being equal to the maximal priority traveling exactly one round as slowly as possible.


**Fig. 9** A property $\mathcal{T}$ stating that a leader is elected within the specified time-bound.

These overall properties of the ring of nodes can be verified with the following refinement checks:

```
refinement:
 (N0 || N1 || N2 || N3 || N4 || N5) <= S
refinement:
 (N0 || N1 || N2 || N3 || N4 || N5) <= T
```

We call this type of verification monolithic, since it constructs and explores the specification representing the entire systems in order to settle the suggested refinements. In the present case with 6 nodes ECDAR quickly proves the refinements and provides a witnessing strategy which can be exercised interactively. However, it is clear that the monolithic approach will suffer from the exponential growth of the states in the number of nodes in the ring.

### 3.3 Compositional Verification

In order to combat the state-space explosion problem and enable verification of the correctness of the protocol for larger numbers of nodes we will apply *compositional verification* for both the functional correctness property $\mathcal{S}$ and the non-functional correctness property $\mathcal{T}$. The idea is to create $N$ sub-specifications $\mathcal{S}_i$ (and $\mathcal{T}_i$) that may be shown to capture the behavior of the sub-ring $N_N||\ldots||N_i$ inductively, by demonstrating the following sequence of refinements:

$$N_N \leq \mathcal{S}_N \quad (1)$$

$$\mathcal{S}_{i+1}||N_i \leq \mathcal{S}_i \text{ for } i = (N-1)\ldots 1 \quad (2)$$

$$\mathcal{S}_1||N_0 \leq \mathcal{S} \quad (3)$$

As mentioned in the introduction this compositional verification is sound because our refinement operator is a precongruence with regards to parallel composition[13].

Using the that the refinement relation $\leq$ is a precongruence with respect to parallel composition and transitive it may be concluded that the ring is a refinement of $\mathcal{S}$. Given six nodes (1), (2) and (3) amounts to performing the following series of refinement checks:

```
refinement: N5 <= S5
refinement: ( S5 || N4 ) <= S4
refinement: ( S4 || N3 ) <= S3
refinement: ( S3 || N2 ) <= S2
refinement: ( S2 || N1 ) <= S1
refinement: ( S1 || N0 ) <= S
```

The series of refinement checks is illustrated in Fig. 10. Though greater in number than the *single* monolithic verification each of the six refinement checks only involve three small components, thus making the overall verification effort linear rather than exponential in the number of nodes in the ring.

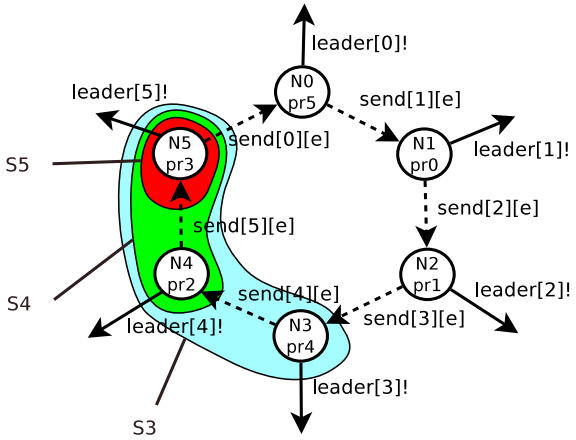


**Fig. 10** Overview of how the induction hypothesis $\varphi 1$ is used to prove the property for a larger and larger set of nodes.

In order to obtain the sub-specifications $\mathcal{S}_i$ and $\mathcal{T}_i$ as instances of general templates, we define the following set of Boolean arrays which are simply used as a reverse look up of which *id*s are included in the set of nodes that a given instantiation of the induction hypothesis covers.

```
const bool S5[N] = { 0, 0, 0, 1, 0, 0};
const bool S4[N] = { 0, 0, 1, 1, 0, 0};
const bool S3[N] = { 0, 0, 1, 1, 1, 0};
const bool S2[N] = { 0, 1, 1, 1, 1, 0};
const bool S1[N] = { 1, 1, 1, 1, 1, 0};
```

These Boolean arrays are then used as input parameters to the corresponding instantiations of the induction hypotheses. The sub-specifications $\mathcal{S}_i$ used to inductively prove the functional property $\mathcal{S}$ is shown in Figure 11, and may be informally described as follows:

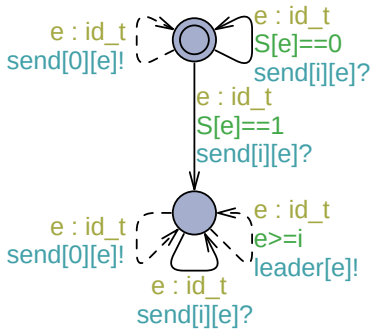


**Fig. 11** The sub-specification $\mathcal{S}_i$. The nodes covered by the sub-specification $(N_N, \ldots, N_i)$ can only declare themselves leader after having received a priority also covered by the sub-specification.

**$\mathcal{S}_i$ first and final version:**
*Whenever the sub-ring $N_N||\ldots||N_i$ receives priorities outside those belonging to one of its nodes, no leader is declared. If a priority belonging to one of the nodes of the sub-ring is received, it is allowed for any of the nodes to declare leadership.*

The sub-specification does not restrict that it has to be the same node that declares itself leader as the one that receives its own *id*. It is worth noting that the sub-specification is this way captures the part of the behavior that is important to prove exactly this property, while ignoring other aspects. In particular, nothing is said about timing of events.

### *3.3.1 Timing property*

Now let us apply the compositional approach to establish the non-functional property $\mathcal{T}$, i.e. that a leader will be elected within `(N+1)*MaxD` time units. Thus, we are searching a (timed) sub-specification $\mathcal{T}_i$, for $i = N \ldots 1$ satisfying the following set of refinements:

$$N_N \leq \mathcal{T}_N \tag{4}$$
$$\mathcal{T}_{i+1}||N_i \leq \mathcal{T}_i \text{ for } i = (N-1)\ldots 1 \tag{5}$$
$$\mathcal{T}_1||N_0 \leq \mathcal{T} \tag{6}$$

The first attempt at defining the timed sub-specification is shown in Fig. 12 and may informally be read as follows:

**$\mathcal{T}_i$ first attempt:**
*Whenever the sub-ring $N_N||\ldots||N_i$ receives a priority larger than any one belonging to one of its nodes, this priority will be delivered to $N_0$ before `(N-i+1)*MaxD` time-units.*

Note the use of the local variable `g` for ensuring that the priority delivered is the one received. However, this proposal for a sub-specification $\mathcal{T}_i$ turned out to be too erroneous (too strong) as it is too strong to be used as the induction hypothesis as it is possible to prove the final step but neither the iterative step nor the base case.

In particular, the base case does not hold as there is no guarantee that a "large" priority received will eventually be delivered to $N_0$ as an even "priority" may be received by the sub-ring in the mean-time. An attempt of correcting this is given in Fig. 13, and may be read informally as follows:

**$\mathcal{T}_i$ second attempt:**
*Whenever the sub-ring $N_N||\ldots||N_i$ receives a priority larger than any one belonging to one of its nodes, this priority will be delivered to $N_0$ before `(N-i+1)*MaxD` time-units,* unless another priority is received before.

As desired, the modified sub-specification validates the refinements required in the base case and the final case. Unfortunately, though seemingly a true property, it turns out that it is too weak for the refinement of the iterative step to hold.

Figure 14 is an attempt of finding a sub-specification for which the refinements of the iterative steps are valid. Here, the behavior after having received a priority and storing it in `g` is made dependent on whether the priority received is equal to the one stored in `g`. Unfortunately this renders all the refinement checks incorrect.

After three (and in fact several) more failing attempts, we finally obtain the satisfactory sub-specification in Fig. 15, that radically differs from the previous in that it only keeps track of what happens to the messages that contains the maximum priority. Informally, the sub-specification reads as follows:

**$\mathcal{T}_i$ final version:**
*Whenever the sub-ring $N_N||\ldots||N_i$ receives the maximum priority before `i*MaxD` time-units - and unless one of the nodes of the sub-ring declares itself leader - the maximum priority will be delivered to $N_0$ before `(N-i+1)*MaxD` time-units.*

Fortunately, this make the sub-specification strong enough to prove the final property $\mathcal{T}$ as well as the iterative refinement steps, yet weak enough to be able to prove the base case and pass the consistency check.

## 3.4 Assume/Guarantee Specifications

In order to make the hunt for the correct sub-specifications easier we will specify $\mathcal{S}$ and $\mathcal{T}$ in the form of a pair

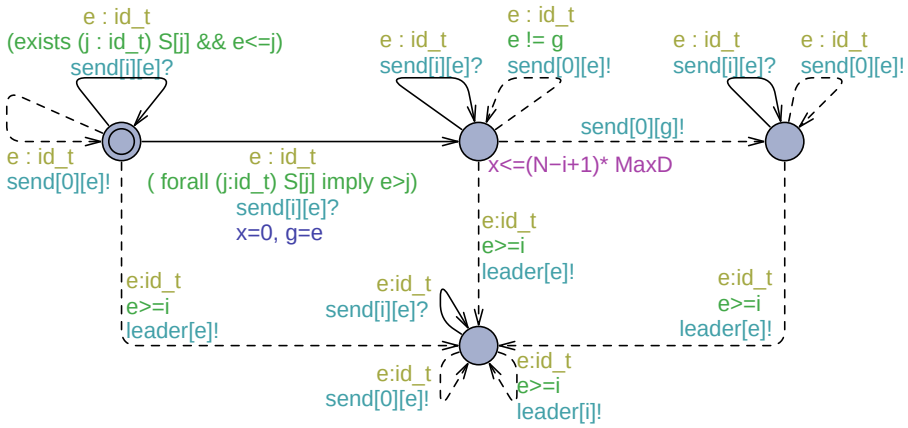


**Fig. 12** The first version of $\mathcal{T}_i$ turned out to be too strong.

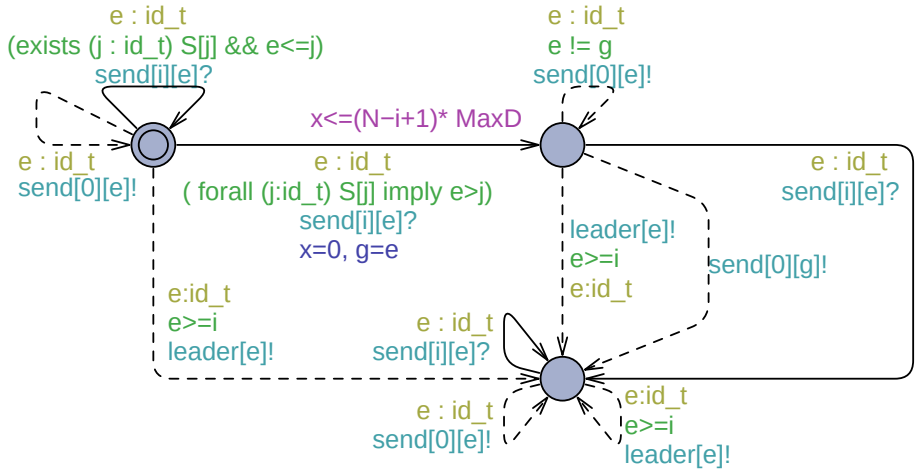


**Fig. 13** The second version of $\mathcal{T}_i$, which turns out to be too weak.

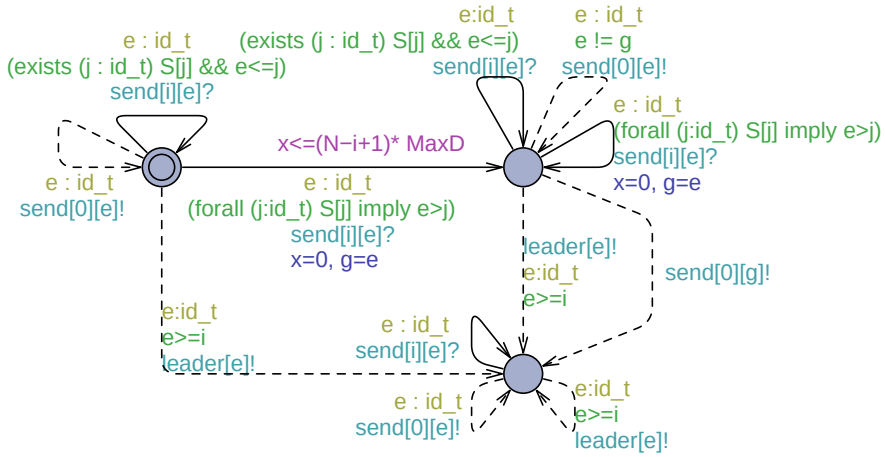


**Fig. 14** Third version of $\mathcal{T}_i$

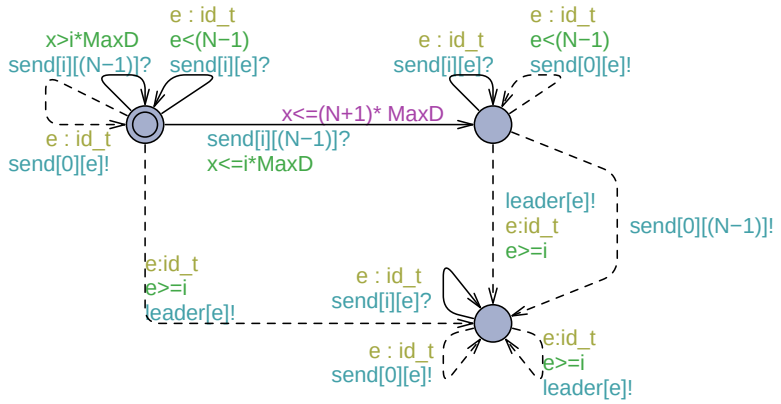


**Fig. 15** Final version of $\mathcal{T}_i$, which only keeps track of the timing regarding messages carrying the maximum priority.

of an assumption and a guarantee part. The assumption and guarantee equivalents of $S$ are shown in Fig. 16 and Fig. 17 respectively.

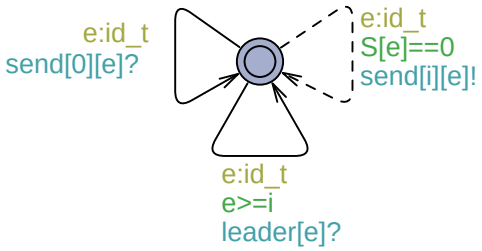


**Fig. 16** The simple assumption $\mathcal{SA}_i$ that no input will be sent with priorities that belong to the set of nodes represented by the sub-specification.

**$\mathcal{SA}_i$ first and final version:**
*We will never send any priority to the sub-ring $N_N||\ldots||N_i$ with priorities belonging to one of its nodes.*

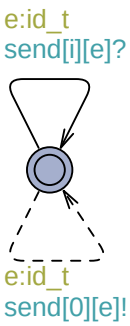


**Fig. 17** The simple guarantee $\mathcal{SG}_i$that no leader output will be generated.

**$\mathcal{SG}_i$ first and final version:**
*The sub-ring $N_N||\ldots||N_i$ will never generate any leader output.*

These two very simple Timed I/O Automata can be combined into a contract using the weakening operator $>>$.

The following two refinements hold (for each $i$):

```
refinement: S1 <= (SG1 >> SA1)
refinement: (SG1 >> SA1) <= S1
```

Thus we have shown that the $S$ that we have come up with is identical to the more easily understandable assumption and guarantee.

The assumption and guarantee equivalents of $T$ are shown in Fig. 18 and Fig. 19 respectively.

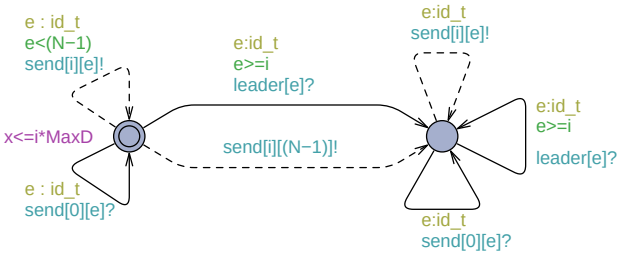


**Fig. 18** The assumption $\mathcal{TA}_i$ that a message with the maximum priority will be delivered to the sub-specification before $i * MaxD$ time units.

**$\mathcal{TA}_i$ first and final version:**
*The maximum priority will be delivered to the sub-ring $N_N||\ldots||N_i$ before $i*MaxD$ time units.*

**$\mathcal{TG}_i$ first and final version:**
*The sub-ring $N_N||\ldots||N_i$ deliver a message with the maximum priority to the node 0 before $(N+1)*MaxD$ time units.*

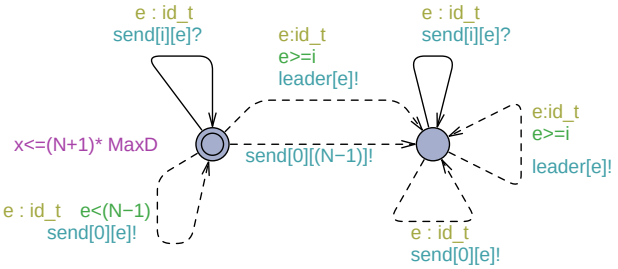


**Fig. 19** The guarantee $\mathcal{TG}_i$ the sub-specification will deliver a message with the maximum priority within $(N+1)*MaxD$ time units.

Similarly as for the other case we can now combine these two specifications into a contract. For this case only one way of the refinements hold (for each $i$):

```
refinement: (TG1 >> TA1) <= T1
```

This means that in this case we can conclude that the composed sub-specification $\mathcal{T}$ that we have come up with refines the contract composed from the assumption and guarantee and thus we can use $\mathcal{T}$ when performing the verification and still rely on the fact the guarantee will hold.

### 3.5 Performance comparison of analysis methods

In order to compare the efficiency of regular monolithic and compositional verification we timed the verification of the two properties $\mathcal{S}$ and $\mathcal{T}$ for several different values of `N`. All the verification was performed on the same machine and all verification instances where allowed a maximum of five minutes to terminate. The choice of exactly five minutes as the upper bound is arbitrary and will not effect the shape of the graphs that we obtain, but only determine the point at which the graphs stop. The upper bound is needed in order to be able to run a large amount of experiments efficiently. The results are listed in Fig. 20. For both the properties in the monolithic cases they took more than five minutes to verify for rings with 7 nodes.

As can be seen from the graph the compositional verification method is capable of handling much larger instances within a reasonable time bound. Besides this the compositional method also has a much larger theoretical upper bound. It will only verify one step at a time and thus will not suffer from lack of available memory as long as a single step can be handled with the available memory.

## 4 Conclusion & Further Work

*Conclusion.* In this paper we have presented the complete specification theory for timed systems underlying the Ecdar tool. Being powered by the game solving engine of the branch Uppaal-tiga, the Ecdar tool provides support for refinement and consistency checking between specifications as well as allow for the logical and structural composition. In particular, as demonstrated in our treatment of the Leader Election Protocol example, the theory and tool allow for efficient *compositional* verification of systems by the exploitation of engineer-provided sub-specifications. As such, the compositional usage of the tool is not *fully* automated, and the design of appropriate sub-specifications – strong enough to entail an overall specification and sufficiently weak to be entailed themselves – is a major challenge. We believe that engineers will always be unfamiliar with any new specification formalism. However, we believe that engineer-provided sub-specifications are not only necessary in the development of realistic systems, but also extremely useful for raising the overall understanding of the systems. In order for the method to be applicable in large scale projects it needs to be supported by a mature tool that is as intuitive as possible to use. As demonstrated in the Leader Election Protocol, tool support is vital in establishing a coherent set of sub-specifications. The need for programmer generated specifications is in no way unique to our approach and is also needed in frameworks such as SPEC# [6] in which assertions (invariants) written by the programmer about a C# program are checked by a range of different analysis techniques.

An important feature of our theory is the existence of a quotient construct (i.e. weakest property transformer with respect to parallel composition), which in particular allows for sub-specifications to be obtained from pairs of assumptions and guarantees. As demonstrated, this often allow for substantially simpler specifications of sub-systems.

*Performance Analysis.* The specification theory presented and the tool Ecdar provide support for establishing hard real-time guaranteed properties from TIOA models. However, as we will sketch in the following, it is possible to also derive soft real-time properties in terms of expected behavior from the same TIOA models. E.g. in the extensive treatment of the Leader Election Protocol of Section 3, we have firmly established that the correct leader is guaranteed to be declared within `(N+1)*MaxD` time-units, given a ring of `N` nodes each implementing the TIOA specification of Fig. 7, i.e. 14 time-units for a ring with 6 nodes. The specification theory presented in

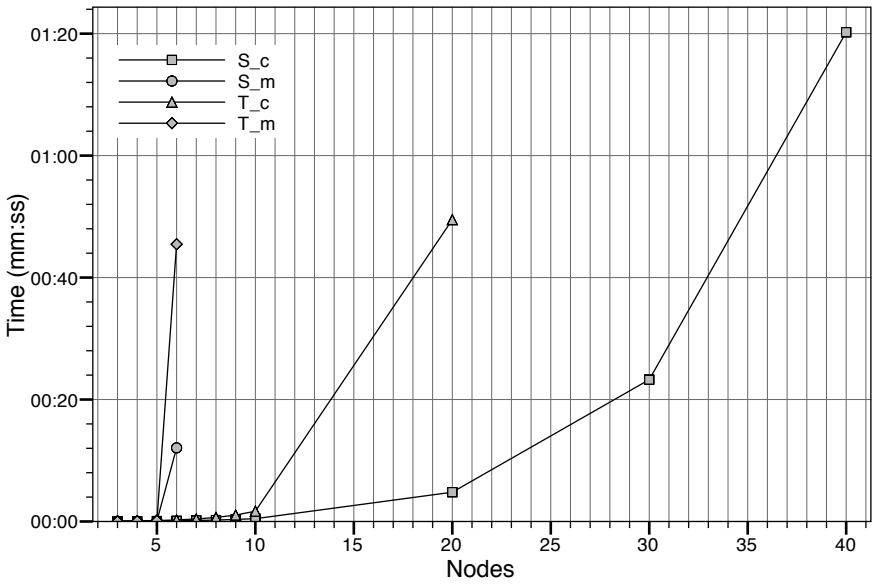


**Fig. 20** Timing results of verification of $\mathcal{S}$ and $\mathcal{T}$ for the compositional and monolithic cases.

this paper, assumes that implementations are concrete executable realizations of specifications. In particular implementations are assumed to have fixed timing behavior, meaning that outputs occur at predictable and exact time moments. However, in a richer setting the timing behavior of implementations could be stochastic, with timing delays of components being chosen by distributions.

In a line of recent work [12,11,9] such a stochastic semantics has been put forward for networks of TIOA, giving a probability measure on sets of runs. This allows for refined probabilistic performance properties to be defined and analyzed, such as the property *"the probability of the set of runs where a leader is declared within 4 time-units is greater than 0.3"*, which could be highly interesting for the Leader Election Protocol. The new Uppaal-smc branch offers a simulation engine allowing to settle such probabilistic properties within desired levels of confidence based on a number of random runs of the system. Assuming that the delay of each node is given by uniform distribution on the interval `[0,MaxD]` Fig. 21 (a) gives the estimated probability, that the leader (node $\mathtt{N}_2$) is declared within `T` time-units, with `T` ranging from 0 to 14. Knowing from our previous verification effort that 14 is the guaranteed upper bound, it is interesting to see that the average time before election is significantly lower, namely 4.42624 time-units Using

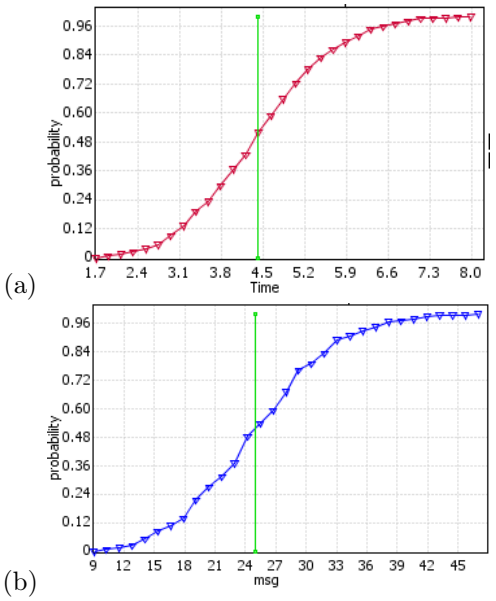


**Fig. 21** Performance Analysis of the Leader Election Protocol, giving the probability that the leader will be declared (a) within `T` time-units and (b) within `M` messages being send, estimated by Uppaal-smc.

Uppaal-smc we obtain $[0.38241, 0.402412]$ as a 95% confidence interval for the probability of that the leader is elected within 4 time-units using 18,445 random runs. On the other hand, directly testing whether this prob-

ability is greater than 0.3 with significance level 0.05 is confirmed with only 266 runs, using the sequential testing method implemented in UPPAAL-SMC.

Extending the model slightly, we may also estimate the *"probability that a leader is declared within a given number $M$ of messages being send"*. Fig. 21 (b) gives an estimation of this probability for $M$ ranging from 0 to 50. We note that on average transmission of some 25 messages is needed.

Following the sketch above for the Leader Election Protocol, we believe that a semantically well-founded extension of the presented TIOA-based specification theory to allow for stochastic implementation would be extremely interesting. In particular, it would enable the refinement of hard real-time guarantees with soft performance statistics in a consistent manner, and allow for the analysis and development of mixed-criticality systems.

## References


1. Martín Abadi and Leslie Lamport. Composing specifications. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 15(1):73–132, 1993.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
3. Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *CONCUR'98*, volume 1466 of *LNCS*. Springer, 1998.
4. Henrik Reif Andersen and Glynn Winskel. Compositional checking of satisfaction. In Kim Guldstrand Larsen and Arne Skou, editors, *CAV*, volume 575 of *Lecture Notes in Computer Science*, pages 24–36. Springer, 1991.
5. Jos C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.
6. Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, LNCS*, volume 3362. Springer, 2004.
7. Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In *CAV*, volume 4590 of *LNCS*. Springer, 2007.
8. Peter Bulychev, Thomas Chatain, Alexandre David, and Kim G. Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *FORMATS*, volume 5813 of *LNCS*, pages 73–87. Springer, 2009.
9. Peter E. Bulychev, Alexandre David, Kim Guldstrand Larsen, Marius Mikucionis, and Axel Legay. Distributed parametric and statistical model checking. In Jiri Barnat and Keijo Heljanko, editors, *PDMC*, volume 72 of *EPTCS*, pages 30–42, 2011.
10. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, 2005.
11. Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, Jonas van Vliet, and Zheng Wang. Statistical model checking for networks of priced timed automata. In Uli Fahrenberg and Stavros Tripakis, editors, *FORMATS*, volume 6919 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2011.
12. Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Zheng Wang. Time for statistical model checking of real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 349–355. Springer, 2011.
13. Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In Karl Henrik Johansson and Wang Yi, editors, *HSCC*, pages 91–100. ACM ACM, 2010.
14. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *FSE*, pages 109–120, Vienna, Austria, September 2001. ACM Press.
15. Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In *In Engineering Theories of Software Intensive Systems, Marktoberdorf Summer School*. Kluwer Academic Publishers, 2004.
16. Luca de Alfaro, Thomas A. Henzinger, and Marielle I. A. Stoelinga. Timed interfaces. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *EMSOFT*, volume 2491 of *LNCS*, pages 108–122. Springer, 2002.
17. Uli Fahrenberg, Axel Legay, and Andrzej Wasowski. Vision paper: Make a difference! (semantically). In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 490–500. Springer, 2011.
18. R. W. Floyd. Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19:19–31, 1967.
19. Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 1998.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
21. C. A. R. Hoare and Jifeng He. The weakest prespecification. *Inf. Process. Lett.*, 24:127–132, January 1987.
22. C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.
23. Cliff B. Jones. Specification as a design base (extended abstract). In A. J. W. Duijvestijn and Peter C. Lockemann, editors, *ECI*, volume 123 of *Lecture Notes in Computer Science*, pages 103–105. Springer, 1981.
24. Cliff B. Jones. *Systematic Software Development using VDM*. Series in Computer Science. Prentice-Hall International, 1986.
25. Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits 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.
26. Kim G. Larsen. *Context-Dependent Bisimulation Between Processes*. PhD thesis, Department of Computer Science, University of Edinburgh, 1986.
27. Kim Guldstrand Larsen and Liu Xinxin. Compositionality through an operational semantics of contexts. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 526–539. Springer, 1990.
28. Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, James Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods: World*

*Congress on Formal Methods in Development of Computer Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer Verlag, 1999.
29. Nancy Lynch. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38, Princeton University, Princeton, N.J., 1988.
30. Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, The MIT Press, November 1988.
31. Rocco De Nicola and Roberto Segala. A process algebraic view of input/output automata. *Theoretical Computer Science*, 138, 1995.
32. Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.
33. Eugene W. Stark, Rance Cleavland, and Scott A. Smolka. A process-algebraic language for probabilistic I/O automata. In *CONCUR*, LNCS, pages 189–2003. Springer, 2003.
34. Jun Sun, Yang Liu, and Jin Song Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008.
35. Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Andre Etienne. Modeling and verifying hierarchical real-time systems using stateful timed csp. *ACM Trans. Softw. Eng. Methodol.*, 2012. Accepted.
36. C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
37. Frits W. Vaandrager. On the relationship between process algebra and input/output automata. In *LICS*, pages 387–398, 1991.