On Implementable Timed Automata

Sergio Feo-Arenis, Milan Vujinović, Bernd Westphal

On Implementable Timed Automata*

Sergio Feo-Arenis[1], Milan Vujinović[2], and Bernd Westphal[2]

[1] Airbus Central R&T
[2] Albert-Ludwigs-Universität Freiburg, Germany

Abstract. Generating code from networks of timed automata is a well-researched topic with many proposed approaches, which have in common that they not only generate code for the processes in the network, but necessarily generate additional code for a global scheduler which implements the timed automata semantics. For distributed systems without shared memory, this additional component is, in general, undesired.
In this work, we present a new approach to the generation of correct code (without global scheduler) for distributed systems without shared memory yet with (almost) synchronous clocks if the source model does not depend on a global scheduler. We characterise a set of implementable timed automata models and provide a translation to a timed while language. We show that each computation of the generated program has a network computation path with the same observable behaviour.

1 Introduction

Automatic code generation from real-time system models promises to avoid human implementation errors and to be cost and time efficient, so there is a need to automatically derive (at least parts of) an implementation from a model. In this work, we consider a particular class of distributed real-time systems consisting of multiple components with (almost) synchronous clocks, yet without shared memory, a shared clock, or a global scheduler. Prominent examples of such systems are distributed data acquisition systems such as data aggregation in satellite constellations [18, 16], the wireless fire alarm system [15], IoT sensors [30], or distributed database systems (e.g. [12]). For these systems, a common notion of time is important (to meet real-time requirements or for energy efficiency) and is maintained up to a certain precision by clock synchronisation protocols, e.g., [17, 23, 24]. Global scheduling is undesirable because schedulers are expensive in terms of network bandwidth and computational power and the number of components in the system may change dynamically, thus keeping track of all components requires large computational resources.

Timed automata, in particular in the flavour of Uppaal [7], are widely used to model real-time systems (see, for example, [14, 32]) and to reason about the correctness of systems as the ones named above. Modelling assumptions of timed automata such as instantaneous updates of variables and zero-time message exchange are often convenient for the analysis of timed system models, yet they, in

* Partly supported by the German Research Council (DFG) under grant WE 6198/1-1.

general, inhibit direct implementations of model behaviour on real-world platforms where, e.g., updating variables take time.

In this work, we aim for the generation of distributed code from networks of timed automata with exactly one program per network component (and no other programs, in particular no implicit global scheduler), where all execution times are considered and modelled (including the selection of subsequent edges), and that comes with a comprehensible notion of correctness. Our work can be seen as the first of two steps towards bridging the gap between timed automata models and code. We propose to firstly consider a simple, iterative programming language with an exact real-time semantics (cf. Section 4) as the target for code generation. In this step, which we consider to be the harder one of the two, we deal with the discrepancy between the atomicity of the timed automaton semantics and the non-atomic execution on real platforms. The second step will then be to deal with imprecise timing on real-world platforms.

Our approach is based on the following ideas. We define a short-hand notation (called *implementable timed automata*) for a sub-language of the well-known timed automata (cf. Section 3). We assume *independency from a global scheduler* [5] as a sufficient criterion for the existence of a distributed implementation. For the timing aspect, we propose not to use platform clocks directly in, e.g., edge guards (see related work below) but to turn model clocks into program variables and to assume a "sleep" operation with absolute deadlines on the target platform (cf. Section 4). In Section 5, we establish the strong and concrete notion of correctness that for each time-safe computation of a program obtained by our translation scheme there is a computation path in the network with the same observable behaviour. Section 6 shows that our short-hand notation is sufficiently expressive to support industrial case studies and discusses the remaining gap towards real-world programming languages like C, and Section 7 concludes.

Generating code for timed systems from timed automata models has been approached before [3, 4, 20, 25, 29]. All these works also generate code for a scheduler (as an additional, explicit component) that corresponds to the implicit, global scheduler introduced by the timed automata semantics [5]. Thus, these approaches do not yield the distributed programs that we aim for. A different approach in the context of timed automata is to investigate discrete sampling of the behaviour [28] and so-called robust semantics [28, 33]. A timed automaton model is then called implementable wrt. to certain robustness parameters. Bouyer et al. [11] have shown that each timed automaton (not a network, as in our case) can be sampled and made implementable at the price of a potentially exponential increase in size. A different line of work is [1, 2, 31]. They use timed automata (in the form of RT-BIP components [6]) as abstract model of the scheduling of tasks. Considering execution times for tasks, a so-called physical model (in a slightly different formalism) is obtained for which an interpreter has been implemented (the *real-time execution engine*) that then realises a scheduling of the tasks. The computation time necessary to choose the subsequent task (including the evaluation of guards) is "hidden" in the execution engine (which

at least warns if the available time is exceeded), and they state the unfortunate observation that time-safety does not imply time-robustness with their approach.

There is an enormous amount of work on so-called *synchronous languages* like Esterel [10], SIGNAL [8], Lustre [19] and *time triggered architectures* such as Giotto/HTL [21]. These approaches provide an abstract programming or modelling language such that for each program, a deployable implementation, in particular for signal processing applications, can be generated.

2 Preliminaries

As modelling formalism (and input to code generation), we consider timed automata as introduced in [7]. In the following, we recall the definition of timed automata for self-containedness. Our presentation follows [26] and is standard with the single exception that we exclude strict inequalities in clock constraints.

A *timed automaton* $\mathcal{A} = (L, A, X, V, I, E, \ell_{ini})$ consists of a finite set of *locations* (including the initial location ℓ_{ini}), sets A, X, and V of *channels*, *clocks*, and *(data) variables*. A *location invariant* $I : L \to \Phi(X)$ assigns a *clock constraint* over X from $\Phi(X)$ to a location. Finitely many *edges* in E are of the form $(\ell, \alpha, \varphi, \vec{r}, \ell') \in L \times A_{!?} \times \Phi(X, V) \times R(X, V)^* \times L$ where $A_{!?}$ consists of input and output actions on channels and the internal action τ, $\Phi(X, V)$ are conjunctions of clock constraints from $\Phi(X)$ and *data constraints* from $\Phi(V)$, and $R(X, V)^*$ are finite sequences of *updates*, an update either resets a clock or updates a data variable. For clock constraints, we exclude strict inequalities as we do not yet support their semantics (of reaching the upper or lower bound arbitrarily close but not inclusive) in the code generation. In the following, we may write $\ell(e)$ etc. to denote the source location of edge e.

The *operational semantics* of a *network* $\mathcal{N} = \mathcal{A}_1 \| \ldots \| \mathcal{A}_n$ of timed automata as *components* – and with pairwise disjoint sets of clocks and variables – is the (labelled) transition system $\mathcal{T}(\mathcal{N}) = (C, \Lambda, \{\xrightarrow{\lambda} | \ \lambda \in \Lambda\}, C_{ini})$ over *configurations*. A configuration $c \in C = \{\langle \vec{\ell}, \nu \rangle \mid \nu \models I(\vec{\ell})\}$ consists of *location vector* $\vec{\ell}$ (an n-tuple whose i-th component is a location of $\mathcal{A}_i$) and a *valuation* $\nu : X(\mathcal{N}) \cup V(\mathcal{N}) \to \mathbb{R}_0^+ \cup \mathcal{D}$ of clocks and variables. The location vector has invariant $I(\vec{\ell}) = \bigwedge_{i=1}^{n} I(\ell_i)$, and we assume a satisfaction relation between valuations and clock and data constraints as usual. Labels are $\Lambda = \{\tau\} \cup \mathbb{R}_0^+$, and the set of *initial configurations* is $C_{ini} = \{\langle (\ell_{ini,1}, \ldots, \ell_{ini,n}), 0 \rangle\} \cap C$. There is a *delay transition* $\langle \vec{\ell}, \nu \rangle \xrightarrow{t} \langle \vec{\ell}, \nu + t \rangle$, $t \in \mathbb{R}_0^+$, if and only if $\nu + t' \models I(\vec{\ell})$ for all $t' \in [0, t]$. There is an *internal transition* $\langle \vec{\ell}, \nu \rangle \xrightarrow{\tau} \langle \vec{\ell}', \nu' \rangle$, if and only if there is an edge $e = (\ell, \tau, \varphi, \vec{r}, \ell')$ enabled in $\langle \vec{\ell}, \nu \rangle$ and ν' is the result of applying e's update vector to ν. An edge is *enabled* in $\langle \vec{\ell}, \nu \rangle$ if and only if its source location occurs in the location vector, its guard is satisfied by ν, and ν' satisfies the destination location's invariant. There is a *rendezvous transition* $\langle \vec{\ell}, \nu \rangle \xrightarrow{\tau} \langle \vec{\ell}', \nu' \rangle$, if and only if there are edges $e_0 = (\ell_0, a!, \varphi_0, \vec{r}_0, \ell_0')$ and $e_1 = (\ell_1, a?, \varphi_1, \vec{r}_1, \ell_1')$ in two different automata enabled in $\langle \vec{\ell}, \nu \rangle$ and ν' is the result of first applying e_0's and then e_1's update vector to ν.

A *transition sequence* of $\mathcal{N}$ is any finite or infinite, initial and consecutive sequence of the form $\langle \vec{\ell}_0, \nu_0 \rangle \xrightarrow{\lambda_1} \langle \vec{\ell}_1, \nu_1 \rangle \xrightarrow{\lambda_2} \cdots$. $\mathcal{N}$ is called deadlock-free if no transition sequence of $\mathcal{N}$ ends in a configuration c such that there are no c', c'' such that $c \xrightarrow{t} c' \xrightarrow{\lambda} c''$ with $t \in \mathbb{R}_0^+$, $\lambda \notin \mathbb{R}_0^+$. A *computation path* of $\mathcal{N}$ is a time stamped transition sequence $\langle \vec{\ell}_0, \nu_0 \rangle, t_0 \xrightarrow{\lambda_1} \langle \vec{\ell}_1, \nu_1 \rangle, t_1 \xrightarrow{\lambda_2} \cdots$ s.t. $t_0 = 0$, $t_{i+1} = t_i + \lambda_{i+1}$ if $\lambda_{i+1} \in \mathbb{R}_0^+$ and $t_{i+1} = t_i$ if $\lambda_{i+1} = \tau$.

Next, Deadline, Boundary. Given an edge e with source location ℓ and clock constraint φ_{clk}, and a configuration $c = \langle \vec{\ell}, \nu \rangle$, we define $\mathit{next}(c, \varphi_{clk}) = \min\{d \in \mathbb{R}_0^+ \mid \nu + d \models I(\ell) \wedge \varphi_{clk}\}$ and $\mathit{deadline}(c, \varphi_{clk}) = \max\{d \in \mathbb{R}_0^+ \mid \nu + \mathit{next}(c, \varphi_{clk}) + d \models I(\ell) \wedge \varphi_{clk}\}$ if minimum/maximum exist and ∞ otherwise. That is, *next* gives the smallest delay after which e is enabled from c and *deadline* gives the largest delay for which e is enabled after *next*. The *boundary* of a location invariant φ_{clk} is a clock constraint $\partial\varphi_{clk}$ s.t. $\nu + d \models \partial\varphi_{clk}$ if and only if $d = \mathit{next}(c, \varphi_{clk}) + \mathit{deadline}(c, \varphi_{clk})$. A simple sufficient criterion to ensure existence of boundaries is to use location invariants of the form $\varphi_{clk} = x \leq q$, then $\partial\varphi_{clk} = x \geq q$.

3 Implementable Timed Automata

In the following, we introduce *implementable timed automata* that can be seen as a definition of a sub-language of timed automata as recalled in Section 2. As briefly discussed in the introduction, a major obstacle with implementing timed automata models is the assumption that actions are instantaneous. The goal of considering the sub-language defined below is to make the execution time of resets and the duration of message transmissions explicit. Other works like, e.g., [13], propose higher-dimensional timed automata where actions take time. We propose to make action times explicit *within* the timed automata formalism.

Definition 1. *An* implementable timed automaton $\mathcal{I} = (L, \ell_{ini}, A, X, V, I, E)$ *consists of locations, initial location, channels, clocks, variables like timed automata, a location invariant* $I : L \to \Phi(X)$ *s.t. each* $I(\ell)$ *has a* boundary $\partial I(\ell)$*, and a finite set* $E = E_\tau \cup E_! \cup E_?$ *of* edges *consisting of*

- internal edges $(\ell, \varphi, \vec{r}_{dat}, \vec{r}_{clk}, \ell') \in E_\tau \subseteq L \times \Phi(X, V) \times R(V) \times R(X) \times L$,
- send edges $(\ell, \varphi, a!, \vec{r}_{clk}, \ell') \in E_! \subseteq L \times \Phi(X, V) \times A_! \times R(X) \times L$,
- receive edges $(\ell, \varphi_{clk}, \{(a_1?, \ell'_1), \ldots, (a_n?, \ell'_n)\}, \vec{r}_{clk}, \ell')$ $n \geq 0$, *in* $E_? \subseteq L \times \Phi(X) \times 2^{A_? \times L} \times R(X) \times L$. ♢

Implementable timed automata distinguish internal, send, and receive edges by action *and* update in contrast to timed automata. An internal edge models (only) updates of data variables or sleeping idle (which takes time on the platform), a send edge models (only) the sending of a message (which takes time), and a receive edge (only) models the ability to receive a message with a timeout. All kinds of edges may reset clocks. Figure 1 shows an example implementable timed automaton using double-outline edges to distinguish the graphical representation from timed automata. The edge from ℓ_0 to ℓ_1, for example, models that

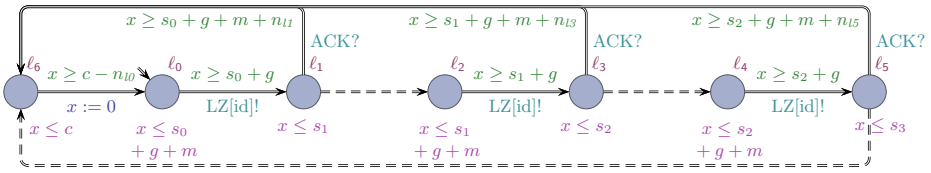
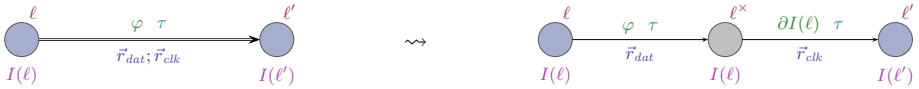
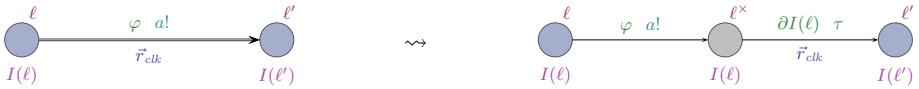


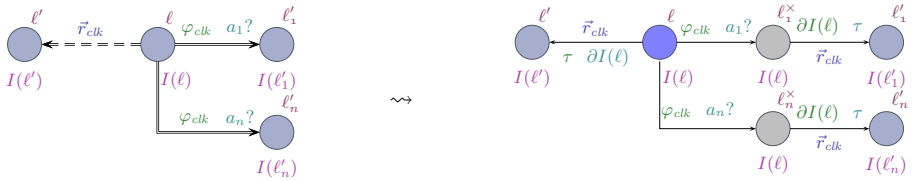
Fig. 1: The LZ-protocol of sensors [15] as implementable timed automaton.



(a) Internal edge $(\ell, \varphi, \vec{r}_{dat}, \vec{r}_{clk}, \ell')$.



(b) Send edge $(\ell, \varphi, a!, \vec{r}_{clk}, \ell')$.



(c) Receive edge $(\ell, \varphi_{clk}, \{(a_1?, \ell'_1), \ldots, (a_n?, \ell'_n)\}, \vec{r}_{clk}, \ell')$.

Fig. 2: Edges of the timed automaton of an implementable timed automaton.

message 'LZ[id]' may be transmitted between time $s_0 + g$ (including guard time g and operating time) and $s_0 + g + m$, i.e., the maximal transmission duration here is m. The time n_{l1} would be the operating time budgeted for location ℓ_1.

The semantics of the *implementable network* $\mathcal{N}$ consisting of implementable timed automata $\mathcal{I}_1, \ldots, \mathcal{I}_n$ is the labelled transition system $\mathcal{T}(\mathcal{A}_{\mathcal{I}_1} \| \ldots \| \mathcal{A}_{\mathcal{I}_n})$. The timed automata $\mathcal{A}_{\mathcal{I}_i}$ are obtained from $\mathcal{I}_i$ by applying the translation scheme in Figure 2 edge-wise. The construction introduces fresh $\ell^\times$-locations. Intuitively, a discrete transition to an $\ell^\times$-location marks the *completion* of a data update or message transmission in $\mathcal{I}$ that started at the *next* time of the considered configuration. After completion of the update or transmission, implementable timed automata always wait up to the deadline. If the update or transmission has a certain time budget, then we need to expect that the time budget may be completely used in some cases. Using the time budget, possibly with a subsequent wait, yields a certain independence from platform speed: if one platform is fast enough to execute the update or transmission in the time budget, then all faster platforms are. Note that the duration of an action may be zero in implementable timed automata (exactly as in timed automata), yet then there will be no time-safe execution of any corresponding program on a real-world platform.

Fig. 3: Artificial example of a non-implementable network if $s_{2,0}+w_2 > s_{1,0}+w_1$.

In [5], the concept of *not to depend on a global scheduler* is introduced. Intuitively, independency requires that sending edges are never blocked because no matching receive edge is enabled or because another send edge in a different component is enabled. That is, the schedule of the network behaviour ensures that at each point in time at most one automaton is ready to send, and that each automaton that is ready to send finds an automaton that is ready for the matching receive. Similar restrictions have been imposed on timed automaton models in [9] to verify the ZeroConf protocol. Whether a network depends on a global scheduler is decidable; for details, we refer the reader to [5].

Figure 3 shows an artificial network of implementable timed automata whose independency from a global scheduler depends on the parameters $s_{1,0} + w_1$ and $s_{2,0} + w_2$. If the location $\ell_{1,1}$ is reached, then the standard semantics of timed automata would (using the implicit global scheduler) block the sending edge until $\ell_{2,1}$ is reached. Yet in a distributed system, the sender should not be assumed to know the current location of the receiver. By choosing the parameters accordingly (i.e., by protocol design), we can ensure that the receiver is always ready *before* the sender so that the sender is never blocked. In this case, we can offer a distributed implementation.

In the following sections, we only consider networks of implementable timed automata that are deadlock-free, *closed component* (no shared clocks or variables, no committed locations (cf. [7])), and do not depend on a global scheduler.

4 Timed While Programs

In this section, we introduce a timed programming language that provides the necessary expressions and statements to implement networks of implementable timed automata as detailed in Section 5. The semantics is defined as a structural operational semantics (SOS) [27] that is tailored towards proving the correctness of the implementations obtained by our translation scheme from Section 5. We use a dedicated time component in configurations of a program to track the execution times of statements and support a snapshot operator to measure the time that passed since the execution of a particular statement. Due to lack of space, we introduce expressions on a strict as-needed basis, including message, location, edge, and time expressions. In a general purpose programming language, the former kinds of expressions can usually be realised using integers (or enumerations), and time expressions can be realised using platform-specific representations of the current system time.

Syntax. Expressions of our programming language are defined wrt. given network variables V and X. We assume that each constraint from $\Phi(X, V)$ or expression

$$S ::= \mathtt{v} \leftarrow \mathit{expr} \mid \mathtt{t} \leftarrow \mathit{texpr} \mid \mathtt{m} \leftarrow \mathit{mexpr} \mid \mathtt{l} \leftarrow \mathit{lexpr} \mid \mathit{sleepto}(\mathit{texpr})$$
$$\mid \mathit{send}(\mathit{mexpr}) \mid \mathtt{m} \leftarrow \mathit{receive}(\mathit{expr}) \mid \mathtt{e}, \mathtt{v}_1, \mathtt{v}_2 \leftarrow \mathit{nextedge}_{\mathcal{I}}([\mathit{mexpr}])$$
$$\mid \mathbf{if} \,\square\, \mathtt{e} = \mathit{eexpr}_1 : \mathcal{S}_1 \ldots \square\, \mathtt{e} = \mathit{eexpr}_n : \mathcal{S}_n \mathbf{fi} \mid \mathbf{while}\ \mathit{expr}\ \mathbf{do}\ \mathcal{S}\ \mathbf{od}$$

$$\mathcal{S} ::= \epsilon \mid S \mid S\triangleleft \mid S; \mathcal{S} \mid S\triangleleft; \mathcal{S} \qquad (\epsilon; \mathcal{S} \equiv \mathcal{S}; \epsilon \equiv \mathcal{S}), \qquad P ::= \mathcal{S}_1 \| \cdots \| \mathcal{S}_n.$$

Table 1: Statements S, statement sequences $\mathcal{S}$, and programs P.

from $\Psi(V)$ over V and X has a corresponding (basic type) program expression and thus that each variable $v \in V$ and each clock $x \in X$ have corresponding (basic type) program variables $\mathtt{v}_v, \mathtt{v}_x \in \mathbb{V}_b$. In addition, we assume typed variables for locations, edges, and messages, and for times (on the target platform). We additionally consider *location variables* $\mathbb{V}_l$ to store the current location, *edge variables* $\mathbb{V}_e$ to store the edge currently worked on, *message variables* $\mathbb{V}_m$ to store the outcome of a receive operation, and *time variables* $\mathbb{V}_t$ to store platform time. *Message expressions* are of the form $\mathit{mexpr} ::= \mathtt{m} \mid a$, $\mathtt{m} \in \mathbb{V}_m$, $a \in A$, *location expressions* are of the form $\mathit{lexpr} ::= \mathtt{l} \mid \ell \mid \mathit{nextloc}_{\mathcal{I}}(\mathit{mexpr})$, $\mathtt{l} \in \mathbb{V}_l$, $\ell \in L$, and *edge expressions* are of the form $\mathit{eexpr} ::= \mathtt{e} \mid e$, $\mathtt{e} \in \mathbb{V}_e$, $e \in E$. A *time expression* has the form $\mathit{texpr} ::= \clockface \mid \mathtt{t} \mid \mathtt{t} + \mathit{expr}$, where $\clockface$ is the *current platform time* and $\mathtt{t} \in \mathbb{V}_t$. Note that time variables are different from clock variables. The values of clock variable $\mathtt{v}_x$ are used to compute a new next time, which is then stored in a time variable, which can be compared to the platform time. Clock variables can be represented by platform integers (given their range is sufficient for the model) while time variables will be represented by platform specific data types like `timespec` with C [22] and POSIX. In this way, model clocks are only indirectly connected (and compared) to the platform clock.

The set of *statements*, *statement sequences*, and *timed programs* are given by the grammar in Table 1. The term $\mathit{nextedge}_{\mathcal{I}}([\mathit{mexpr}])$ represents an implementation of the edge selection in an implementable timed automaton that can optionally be called with a message expression. We denote the empty statement sequence by ϵ and introduce $\triangleleft$ as an artificial *snapshot operator* on statements (see below). The particular syntax with snapshot and non-snapshot statements allows us to simplify the semantics definition below. We use *StmSeq* to denote the set of all statement sequences.

Component Configurations and Interpretation of Expressions. A *component configuration* is a tuple $\pi = \langle \mathcal{S},\ (\beta, \gamma, w, u),\ \sigma \rangle$ consisting of a statement sequence $\mathcal{S} \in \mathit{StmSeq}$, the *operating time* of the current statement $\beta \in \mathbb{R}_0^+$ (i.e., the time passed since starting to work on the current statement), the *time to completion* of the current statement $\gamma \in \mathbb{R}_0^+ \cup \{\infty\}$ (i.e., the time it will take to complete the work on the current statement), the *snapshot time* $w \in \mathbb{R}_0^+$ (i.e., the time since the last snapshot), the *platform clock* value[3] $u \in \mathbb{R}_0^+$, and a type-consistent

[3] Using a real, unbounded value for the platform clock avoids the issue of overflows in executions of programs as defined here. When refining the programs of imple-

$$\text{(R1)}\ \frac{\langle \mathtt{v} \leftarrow \mathit{expr}; \mathcal{S},\ (\beta, 0, w, u),\ \sigma\rangle}{\langle \mathcal{S},\ (0, \gamma', w', u),\ \sigma[\mathtt{v} := \sigma(\mathit{expr})]\rangle} \quad \text{(R5)}\ \frac{\langle \mathit{sleepto}(\mathit{texpr}); \mathcal{S},\ (\sigma(\mathit{texpr}), 0, w, u),\ \sigma\rangle}{\langle \mathcal{S},\ (0, \gamma', w', u),\ \sigma\rangle}$$

$$\text{(R6)}\ \frac{\langle \mathit{send}(\mathit{mexpr}); \mathcal{S},\ (\beta, 0, w, u),\ \sigma\rangle}{\langle \mathcal{S},\ (0, \gamma', w', u),\ \sigma\rangle} \quad \text{(R7)}\ \frac{\langle \mathtt{m} \leftarrow \mathit{receive}(\mathit{expr}); \mathcal{S},\ (\beta, \gamma, w, u), \sigma\rangle}{\langle \mathcal{S},\ (0, \gamma', w', u),\ \sigma[\mathtt{m} := a]\rangle},\ \begin{array}{l} a \in A, \text{ if } \beta \leq \sigma(\mathit{expr}), \\ a = \bot, \text{ if } \beta \geq \sigma(\mathit{expr}), \end{array}$$

$$\text{(R8)}\ \frac{\langle \mathtt{e}, \mathtt{v}_1, \mathtt{v}_2 \leftarrow \mathit{nextedge}_{\mathcal{I}}([\mathit{mexpr}]); \mathcal{S},\ (\beta, 0, w, u),\ \sigma\rangle}{\langle \mathcal{S},\ (0, \gamma', w', u),\ \sigma[\mathtt{e}, \mathtt{v}_1, \mathtt{v}_2 := [\![\mathit{nextedge}_{\mathcal{I}}([\mathit{mexpr}])]\!](\sigma)]\rangle}$$

$$\text{(R9a)}\ \frac{\langle \mathbf{if} \cdots \square\ \mathtt{e} = \mathit{eexpr}_i : \mathcal{S}_i \cdots \mathbf{fi}; \mathcal{S},\ (\beta, 0, w, u),\ \sigma\rangle}{\langle \mathcal{S}_i; \mathcal{S},\ (0, \gamma', w', u),\ \sigma\rangle}, \sigma(\mathtt{e}) = \sigma(\mathit{eexpr}_i)$$

$$\text{(R9b)}\ \frac{\langle \mathbf{if} \square\ \mathtt{e} = \mathit{eexpr}_1 : \mathcal{S}_1 \ldots \square\ \mathtt{e} = \mathit{eexpr}_n : \mathcal{S}_n\ \mathbf{fi}; \mathcal{S},\ (\beta, 0, w, u), \sigma\rangle}{\langle \mathcal{S},\ (0, \gamma', w', u),\ \sigma\rangle}, \begin{array}{l} \forall\, 0 \leq i \leq n \bullet \\ \sigma(\mathtt{e}) \neq \sigma(\mathit{eexpr}_i) \end{array}$$

$$\text{(R10a)}\ \frac{\langle \mathbf{while}\ \mathit{expr}\ \mathbf{do}\ S\ \mathbf{od}; \mathcal{S},\ (\beta, 0, w, u),\ \sigma\rangle}{\langle S; \mathbf{while}\ \mathit{expr}\ \mathbf{do}\ S\ \mathbf{od}; \mathcal{S},\ (0, \gamma', w', u),\ \sigma\rangle}, \sigma(\mathit{expr}) = \mathit{true}$$

$$\text{(R10b)}\ \frac{\langle \mathbf{while}\ \mathit{expr}\ \mathbf{do}\ S\ \mathbf{od}; \mathcal{S},\ (\beta, 0, w, u),\ \sigma\rangle}{\langle \mathcal{S},\ (0, \gamma', w', u),\ \sigma\rangle}, \sigma(\mathit{expr}) = \mathit{false}$$

Table 2: Discrete reductions of the timed programming language. Rules (R2), (R3), and (R4) for time, message, and location assignment are similar to (R1).

valuation σ of the program variables. We will use operating time and time to completion to define computations of timed while programs (with discrete transitions when the time to completion is 0), and we will use the snapshot time w as an auxiliary variable in the construction of predicates by which we relate program and network computations. The valuation σ maps basic type variables from $\mathbb{V}_b$ to values from a domain that includes all values of data variables from $\mathcal{D}$ as used in the implementable timed automaton and all values needed to evaluate clock constraints (see below), i.e. $\sigma(\mathbb{V}_b) \subseteq \mathcal{D}_b$. Time variables from $\mathbb{V}_t$ are mapped to non-negative real numbers, i.e., $\sigma(\mathbb{V}_t) \subseteq \mathbb{R}_0^+$, message variables from $\mathbb{V}_m$ are mapped to channels, i.e., $\sigma(\mathbb{V}_m) \subseteq A \cup \{\bot\}$ or the dedicated value $\bot$ representing 'no message', location variables from $\mathbb{V}_l$ are mapped to locations, i.e., $\sigma(\mathbb{V}_l) \subseteq L$, and edge variables from $\mathbb{V}_e$ are mapped to edges, i.e., $\sigma(\mathbb{V}_e) \subseteq E$.

For the interpretation of expressions in a component configuration we assume that, if the valuation σ of the program variables corresponds to the valuation of data variables ν, then the interpretation $[\![\mathit{expr}]\!](\pi)$ of basic type expression *expr* corresponds to the value of *expr* under ν. Other variables obtain their values from σ, too, i.e. $[\![\mathtt{t}]\!](\pi) = \sigma(\mathtt{t})$, $[\![\mathtt{m}]\!](\pi) = \sigma(\mathtt{m})$, $[\![\mathtt{l}]\!](\pi) = \sigma(\mathtt{l})$, and $[\![\mathtt{e}]\!](\pi) = \sigma(\mathtt{e})$; constant symbols are interpreted by their corresponding value, i.e. $[\![a]\!](\pi) = a$, $[\![\ell]\!](\pi) = \ell$, and $[\![e]\!](\pi) = e$, and we have $[\![\mathtt{t} + \mathit{expr}]\!](\pi) = [\![\mathtt{t}]\!](\pi) + [\![\mathit{expr}]\!](\pi)$.

mentable timed automata to programs on realistic platforms, we need to handle possible overflows in the finitely represented current platform time.

There are two non-standard cases. The $\textcircled{\scriptsize\textsf{T}}$-symbol denotes the platform clock value of π, i.e.. $[\![\textcircled{\scriptsize\textsf{T}}]\!](\pi) = u$, and we assume that $[\![\mathit{nextloc}_{\mathcal{I}}([\mathit{mexpr}])]\!](\pi)$ yields the destination location of the edge that is currently processed (as given by $\mathtt{e}$), possibly depending on a message name given by *mexpr*. If $[\![\mathtt{e}]\!](\pi)$ denotes an internal action or send edge e, this is just the destination location $\ell'(e)$, for receive edges it is $\ell'(e)$ if *mexpr* evaluates to the special value $\perp$, and an ℓ_i from a $(a_i?, \ell_i)$ pair in the edge otherwise. If the receive edge is non-deterministic, we assume that the semantics of $\mathit{nextloc}_{\mathcal{I}}$ resolves the non-determinism.

Program Computations. Table 2 gives an SOS-style semantics with discrete reduction steps of a statement sequence (or component). Note that the rules in Table 2 (with the exception of receive) apply when the time to completion is 0, that is, at the point in time where the current statement completes. Each rule then yields a configuration with the operating time γ' for the new current statement. The new snapshot time w' is 0 if the first statement in $\mathcal{S}$ is a snapshot statement $S\triangleleft$, and w otherwise. Rule (R7) updates $\mathtt{m}$ to a, which is a channel or, in case of timeout, the 'no message' indicator '$\perp$'. Rule (R8) is special in that it is supposed to represent the transition relation of an implementable timed automaton. Depending on the program valuation σ, (R8) is supposed to yield a triple of the next edge to work on, this edge's *next* and *deadline*. For simplicity, we assume that the interpretation of $\mathit{nextedge}_{\mathcal{I}}([\mathit{mexpr}])$ is deterministic for a given valuation of program variables.

A *configuration* of program $P = \mathcal{S}_1 \| \cdots \| \mathcal{S}_n$ is an n-tuple

$$\mathit{\Pi} = (\langle \mathcal{S}_1,\ (\beta_1, \gamma_1, w_1, u_1),\ \sigma_1\rangle, \ldots, \langle \mathcal{S}_n,\ (\beta_n, \gamma_n, w_n, u_n),\ \sigma_n\rangle)$$

of component configurations; $\mathcal{C}(P)$ denotes the set of all configurations of P.

The operational semantics of a program P is the labelled transition system on system configurations defined as follows. There is a *delay transition*

$$\begin{array}{c}(\langle \mathcal{S}_1,\ (\beta_1, \gamma_1, w_1, u_1),\ \sigma_1\rangle, \ldots) \xrightarrow{\delta} \\ (\langle \mathcal{S}_1,\ (\beta_1 + \delta, \gamma_1 - \delta, w_1 + \delta, u_1 + \delta),\ \sigma_1\rangle, \ldots)\end{array}$$

(by delay $\delta \in \mathbb{R}_0^+$) if, for all i, $1 \leq i \leq n$, $\delta \leq \gamma_i$, i.e. if no current statement completes strictly before δ. There is an *internal transition*

$$(\ldots, \langle \mathcal{S}_i,\ (\beta_i, 0, w_i, u_i),\ \sigma_i\rangle, \ldots) \xrightarrow{\tau} (\ldots, \langle \mathcal{S}_i',\ (0, \gamma_i', w_i', u_i),\ \sigma_i'\rangle, \ldots)$$

if for some i, $1 \leq i \leq n$, a discrete reduction rule from Table 2 applies, i.e. if

$$\langle \mathcal{S}_i,\ (\beta_i, 0, w_i, u_i),\ \sigma_i\rangle \vdash \langle \mathcal{S}_i',\ (0, \gamma_i', w_i', u_i),\ \sigma_i'\rangle.$$

There is a *synchronisation transition*

$$\begin{array}{c}(\ldots, \langle \mathcal{S}_i,\ (\beta_i, 0, w_i, u_i),\ \sigma_i\rangle, \ldots \langle \mathcal{S}_j,\ (\beta_j, \gamma_j, w_j, u_j),\ \sigma_j\rangle, \ldots) \xrightarrow{[\![\mathit{mexpr}]\!](\sigma_i)} \\ (\ldots, \langle \mathcal{S}_i',\ (0, \gamma_i', w_i', u_i),\ \sigma_i\rangle, \ldots \langle \mathcal{S}_j',\ (0, \gamma_j', w_j', u_j),\ \sigma_j'\rangle, \ldots)\end{array}$$

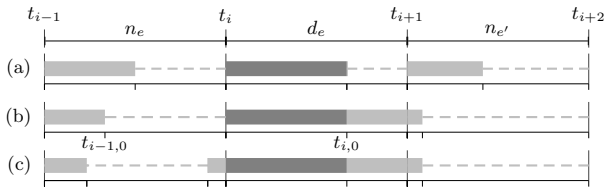


Fig. 4: Scheduling of work and operating time.

if $\langle \mathcal{S}_i,\ (\beta_i, 0, w_i, u_i),\ \sigma_i\rangle \vdash \langle \mathcal{S}'_i,\ (0, \gamma'_i, w'_i, u_i),\ \sigma_i\rangle$ by (R6), and $\langle \mathcal{S}_j, (\beta_j, \gamma_j, w_j, u_j), \sigma_j\rangle \vdash \langle \mathcal{S}'_j,\ (0, \gamma'_j, w'_j, u_j),\ \sigma'_j\rangle$ by (R7), and $\beta_j \geq \beta_i$, i.e. if component j has been listening at least as long as component i has been sending.

Note that this definition of synchronisation allows multiple components to send at the same time (which may cause message collision on a shared medium) and that, similar to the rendezvous communication of timed automata, out of multiple receivers, only one takes the message. In our application domain these cases do not happen because we assume that implementable networks do not depend on a global scheduler. That is, the program of an implementable network never exhibits any of these two behaviours.

A program configuration is called *initial* if and only if the k-th component configuration, $1 \leq k \leq n$, is at $\mathcal{S}_k$, with any β_k, $\gamma_k = 0$, $w_k = 0$, $u_k = 0$, and any σ_k with $\sigma_k(\mathbb{V}_b) = 0$. We use $\mathcal{C}_{ini}(P)$ to denote the set of initial configurations of program P. A *computation* of P is an initial and consecutive sequence of program configurations $\zeta = \Pi_0, \Pi_1, \ldots$, i.e. $\Pi_0 \in \mathcal{C}_{ini}(P)$ and for all $i \in \mathbb{N}_0$ exists $\lambda \in \mathbb{R}_0^+ \cup \{\tau\}$ such that $\Pi_i \xrightarrow{\lambda} \Pi_{i+1}$ as defined above. We need not consider terminating computations of programs here because we assume networks of implementable timed automata without deadlocks.

5 Correct Implementation of Implementable Networks

The program of the network of implementable timed automata $\mathcal{N} = \mathcal{I}_1 \| \ldots \| \mathcal{I}_n$ is $P(\mathcal{N}) = \mathcal{S}(\mathcal{I}_1) \| \ldots \| \mathcal{S}(\mathcal{I}_n)$ (cf. Table 3c). The edges' work is implemented in the corresponding Line 2 of the statement sequences in Tables 3a and 3b. The remaining Lines 3 to 8 include the evaluation of guards to choose the edge to be executed next. The result of choosing the edge is stored in program variable `e` which (by the while loop and the if-statement) moves to Line 1 of the implementation of that edge. The program's timing behaviour is controlled by variable `t` and is thus decoupled from clocks in the timed automata model. After Line 8, the value of `t` denotes the *absolute time* where the execution of the next edge is due. That is, clocks in the program are not directly compared to the platform time (which would raise issues with the precision of platform clocks) but are used to determine points in time that the target platform is supposed to sleep to. By doing so, we also lower the risk of accumulating imprecisions in the sleep operation of the target platform when sleeping for many *relative* durations.

1: $sleepto(\mathtt{t})\triangleleft$; // *sleep to current next at t_i, then snapshot*
2: ($\vec{r}_{dat}$ | $send(a)$); // *from t_i to $t_{i,0}$, work on τ- or a!-edge*
3: $\mathtt{l} \leftarrow \ell'_0$; // *now fictionally at t_{i+1} in destination location*
4: $\mathtt{x} \leftarrow (\mathtt{x} + \mathtt{n} + \mathtt{d})[\vec{r}_{clk}]$; // *fictionally delay to t_{i+1}, reset clocks*
5: $\mathtt{t} \leftarrow \mathtt{t} + \mathtt{d}$; // *new sleep goal (1/2, see below): to t_{i+1} (old deadline)*
6: $\mathtt{e},\mathtt{n},\mathtt{d} \leftarrow nextedge_{\mathcal{I}}()$; // *choose next edge* e *based on current component con-*
7: // *figuration, get next and deadline of* e
8: $\mathtt{t} \leftarrow \mathtt{t} + \mathtt{n}$ // *new sleep goal (2/2): and then to t_{i+2} (new next)*

(a) Implementation $\mathcal{S}(e,\mathcal{I})$ of internal or send edge e in $\mathcal{I}$. Line 2 is the update vector $\vec{r}_{dat}$ if $e \in E_\tau$ (internal edge) and the send action $send(a)$ if $e \in E_!$ (send edge).

2: $\mathtt{m} \leftarrow receive(\mathtt{d})$; // *from t_i to $t_{i,0}$, work on receive edge, i.e. read message*
3: $\mathtt{l} \leftarrow nextloc_{\mathcal{I}}(\mathtt{m})$; // *at $t_{i,0}$, if no* m*-match: treat like timeout*
6: $\mathtt{e},\mathtt{n},\mathtt{d} \leftarrow nextedge_{\mathcal{I}}(\mathtt{m})$; // *choose next edge* e *based on current component confi-*
7: // *guration and message (!), get next and deadline of* e

(b) Implementation $\mathcal{S}(e,\mathcal{I})$ of receive edge e in $\mathcal{I}$; Lines 1, 4-5, and 8 are as in Figure 3a.

1: $\mathtt{t} \leftarrow \mathcal{O}$; // *get beginning of time; assume basic type variables are 0*
2: $\mathtt{l} \leftarrow \ell_{ini}$; // *initialise location*
3: $\mathtt{e},\mathtt{n},\mathtt{d} \leftarrow nextedge_{\mathcal{I}}()$; // *choose next edge* e *based on component configuration*
4: $\mathtt{t} \leftarrow \mathtt{t} + \mathtt{n}$; // *new sleep goal: beginning of time plus next of* e
5: **while** *true* **do if** $\square\ \mathtt{e} = e_0 :\ \mathcal{S}(e_0,\mathcal{I}) \cdots \square\ \mathtt{e} = e_n :\ \mathcal{S}(e_n,\mathcal{I})$ **fi od**

(c) Implementation $\mathcal{S}(\mathcal{I})$ of implementable timed automaton $\mathcal{I}$.

Table 3: Implementation scheme for implementable timed automaton.

The idea of scheduling work and operating time is illustrated by the timing diagram in Figure 4. Row (a) shows a naïve schedule for comparison: From time t_{i-1}, decide on the next edge to execute and determine this edge's *next* time at t_i (light grey phase: operating time, must complete within the next edge's *next* time n_e), then sleep up to the *next* time (dashed grey line), then execute the edge(s) actions (dark grey phase: work time, must complete within the edge's deadline d_e), then sleep up to the edge's deadline at t_{i+1}, and start over. The program obtained by our translation scheme implements the schedule shown in Row (b). The program begins with determining the next edge *right after* the work phase and then has only one sleep phase up to, e.g., t_{i+2} where the next work phase begins. In this manner, we require only one interaction with the execution platform that implements the sleep phases. Row (c) illustrates a possible extension of our approach where operating time is needed right before the work phase, e.g., to prepare the platform's transceiver for sending a message.

We call the program $P(\mathcal{N})$ a correct implementation of network $\mathcal{N}$ if and only if for each observable behaviour of a *time-safe* execution of $P(\mathcal{N})$ there is a *corresponding* computation path of $\mathcal{N}$. In the following, we provide our notion of time-safety and then elaborate on the above mentioned correspondence between program and network computations.

Intuitively, a computation of $P(\mathcal{N})$ is not time-safe if either the execution of an edge's statement sequence takes longer than the admitted deadline or if the *next* time of the subsequent edge is missed, e.g., by an execution platform that is too slow. Note that in a given program computation, the performance of the platform is visible in the operation time β and time to completion γ.

We write $\Pi^k{:}\mathrm{L}_n^e$ to denote that the program counter of component k is at Line n of the statement sequence of edge e. We use $\sigma|_{X\cup V}$ to denote the (network) configuration encoded by the values of the corresponding program variables. We assume[4] that for each program variable `v`, the old value, i.e., the value before the last assignment in the computation is available as `@v`.

Definition 2. *A computation $\Pi_0, \Pi_1, \ldots$ of $P(\mathcal{N})$ is* time-safe *if and only if, for each component k, $0 \leq k \leq n$ and all $i \in \mathbb{N}_0$,*

1. *$\Pi_i^k{:}L_2^e \wedge \gamma_{i,k} = 0 \implies w_k \leq \mathit{deadline}(\langle \sigma_{i,k}(\mathtt{l}), \sigma_{i,k}|_{X\cup V}\rangle, \sigma_{i,k}(\mathtt{e}))$, i.e., if the i-th configuration completes ($\gamma_{i,k} = 0$) Line 2 of an edge's statement sequence, not more time than admitted by its deadline has been used (w_k),*
2. *$\Pi_i^k{:}L_1^e \wedge \gamma_{i,k} = 0 \implies w_k = \sigma_{i,k}(\mathtt{@d}) + \mathit{next}(\langle \sigma_{i,k}(\mathtt{l}), \sigma_{i,k}|_{X\cup V}\rangle, \sigma_{i,k}(\mathtt{e}))$, i.e., the sleepto statement in Line 1 completes exactly after the deadline of the previously worked on edge plus the current edge's* next *time.* ♢

Note that, by Definition 2, operating times may be larger than the subsequent edge's *next* time in a time-safe computation (if the execution of the current edge completes before its deadline). Stronger notions of time-safety are possible.

For correctness of $P(\mathcal{N})$, recall that we introduced Timed While Programs to consider the computation time that is needed to compute the transition relation of an implementable network on the fly. In addition, program computations have a finer granularity than network computations: In network computations, the current location and the valuation of clocks and variables are updated atomically in a transition. In the program $P(\mathcal{N})$, these updates are spread over three lines.

We show that, for each time-safe computation ζ of program $P(\mathcal{N})$, there is a computation of network $\mathcal{N}$ that is related to ζ in a well-defined way. The relation between program and network configurations decouples both computations in the sense that at some times (given by the respective timestamp) the, e.g., clock values in the program configuration are "behind" network clocks (i.e., correspond to an earlier network configuration), at some times they are "ahead", and there are points where they coincide.

Figure 5 illustrates the relation for one edge e. The top row of Figure 5 gives a timing diagram of the execution of the program for edge e of one component. The rows below show the values over time for each program variable `v` up to `e`, `n`, and `d`. For example, the value of `l` will denote the source location ℓ of e until Line 3 is completed, and then denotes the destination location ℓ'. Similarly, v' and x' denote the effects of the update vector of e on data variables and clocks. Note that, during the execution of Line 3, we may observe combinations of values

[4] Without loss of generality, since the program could be augmented by an auxiliary variable `@v` for each variable `v` that provides the old value of `v`.

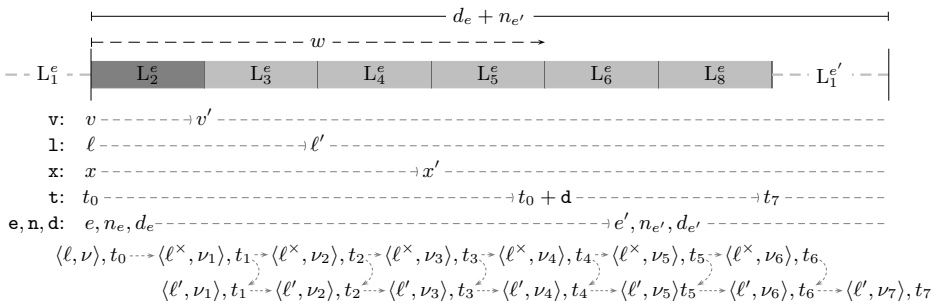
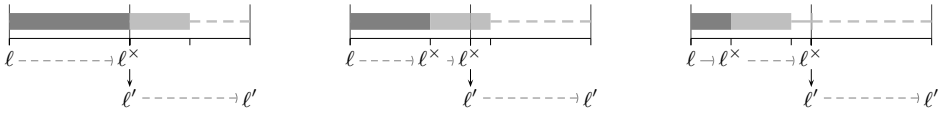


Fig. 5: Relating program and network computations for one component.



(a) At deadline. (b) During operating time. (c) During sleep time.

Fig. 6: Cases of changing from intermediate location to destination location.

for v and l that are never observed in a network computation due to the atomic semantics of networks.

The two bottom lines of Figure 5 show related network configurations aligned with their corresponding program lines. Note that the execution of each line except for Line 1 may be related to two network configurations depending on whether the program timestamp is before or after the current edge's deadline. Figure 6 illustrates the three possible cases: The execution of program Line 2 (work time, dark gray) is related to network configurations with the source location ℓ of the current edge. Right after the work time, the network location $\ell^\times$ is related and at the current edge's deadline the destination location ℓ' is related. In the related network computation, the transition from $\ell^\times$ to ℓ' always takes place at the current edge's deadline. This point in time may, in the program computation, be right after work time (Figure 6a, no delay in $\ell^\times$), in the operating time (Figure 6b), or in the sleep time (Figure 6c).

The relation between program and network configurations as illustrated in Figure 5 can be formalised by predicates over program and network configurations, one predicate per edge and program line.[5] The following lemma states the described existence of a network computation for each time-safe program computation. The relation gives a precise, component-wise and phase-wise relation of program computations to network computations. In other words, we obtain a precise accounting of *which* phases of a time-safe program computation cor-

[5] Details on these predicates and a detailed proof of Lemma 1 are provided in a corresponding technical report.

respond to a network computation and *how*. We can argue component-wise by the closed component-assumption from Section 3.

Lemma 1. *For each time-safe computation $\zeta = \Pi_0, \Pi_1, \ldots$ of $P(\mathcal{N})$, there exists a computation path $\xi = c_{0,0}, \ldots, c_{0,m_0}, c_{1,0}, \ldots$ of $\mathcal{N}$ s.t. each network configuration $c_{i,j}$ is properly related to program configuration Π_i.* ♢

Proof (sketch). The proof is a technical check of the predicates mentioned above during an inductive construction of computation path ξ. For the base case, we show that the initialisation statements in Lines 1 to 4 of Table 3c reach the Line 2 of a send or receive edge (cf. Table 3a and 3b) and establish a related network configuration. For the induction step, we need to consider delays and discrete steps of the program. From time-safety of ζ we can conclude to possible delays in $\mathcal{N}$ for the related configurations with a case-split wrt. the deadline (cf. Figure 6). When the program time is at the current edge's deadline, the network may delay up to the deadline in an intermediate location $\ell^{\times}$, take a transition to the successor location ℓ', and possibly delay further. For discrete program steps, we can verify that $\mathcal{N}$ has enabled discrete transitions that reach a network configuration that is related to the next program line. Here, we use our assumptions from the program semantics that update vectors have the same effect in the program and the network. And we use the convenient property of our program semantics that the effects of statements only become visible with the discrete transitions. For synchronisation transitions of the program, we use the assumption that the considered network of implementable timed automata does not depend on a global scheduler, in particular that send actions are never blocked, or, in other words, that whenever a component has a send edge locally enabled, then there is a receiving edge enabled on the same channel. □

Our main result in Theorem 1 is obtained from Lemma 1 by a projection onto observable behaviour (cf. Definition 3). Intuitively, the theorem states that at each point in time with a discrete transition to Line 2, the program configuration exactly encodes a configuration of network $P(\mathcal{N})$ right before taking an internal, send, or receive edge.

Definition 3. *Let $\xi^k = \langle \ell^k_{0,0}, \nu^k_{0,0} \rangle \xrightarrow{\lambda_{0,1}} \ldots \xrightarrow{\lambda_{0,m_0}} \langle \ell^k_{1,0}, \nu^k_{1,0} \rangle \ldots$ be the projection of a computation path ξ of the implementable network $\mathcal{N}$ onto component k, $1 \leq k \leq n$, labelled such that each configuration $\langle \ell^k_{i,0}, \nu^k_{i,0} \rangle$ is initial or reached by a discrete transition to a source location of an internal, send, or receive edge.*

The sequence $\xi^k_{obs} = \langle \ell^k_{0,i_0}, \nu^k_{0,i_0} + d_0 \rangle, \langle \ell^k_{1,i_1}, \nu^k_{1,i_1} + d_1 \rangle, \ldots, d_j \geq 0$, where (j, i_j) is the largest index such that between $c := \langle \ell^k_{j,0}, \nu^k_{j,0} \rangle$ and $\langle \ell^k_{j,i_j}, \nu^k_{j,i_j} + d_j \rangle$ exactly next(c) time units have passed, is called the observable behaviour of component k in ξ. ♢

Theorem 1. *Let $\mathcal{N}$ be an implementable network and $\zeta^k = \pi_{0,0}, \ldots, \pi_{0,n_0}, \pi_{1,0}, \ldots$ the projection onto the k-th component of a time-safe computation ζ of $P(\mathcal{N})$ labelled such that $\pi_{i,n_i}, \pi_{i+1,0}$ are exactly those transitions in ζ from a Line 1 to the subsequent Line 2. Then $(\langle \sigma_{i,0}(\mathtt{l}), \sigma_{i,0}|_{X \cup V} \rangle, u_{i,0})_{i \in \mathbb{N}_0}$ is an observable behaviour of component k on some computation path of $\mathcal{N}$.* ♢

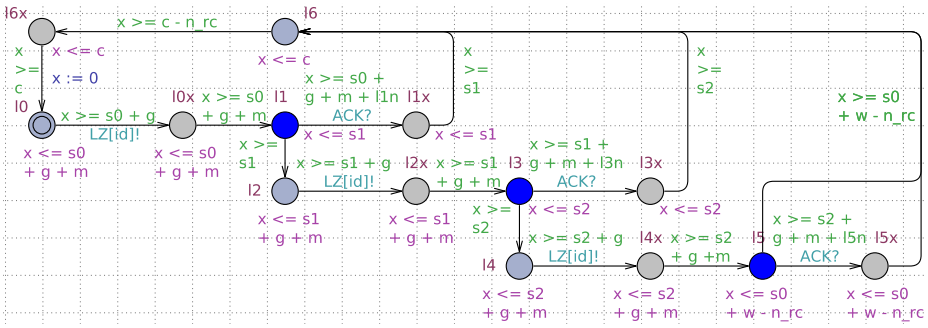


Fig. 7: Timed automaton of the implementable timed automaton (after applying the scheme from Figure 2) for the LZ-protocol of sensors [15].

6 Evaluation and Discussion

The work presented here was motivated by a project to support the development of a new communication protocol for a distributed wireless fire alarm system [15], without shared memory, only assuming clock synchronisation and message exchange. We provided modelling and analysis of the protocol *a priori*, that is, before the first line of code had been written. In the project, the engineers manually implemented the model and appreciated how the model indicates exactly which action is due in which situation. Later, we were able to study the handwritten code and observed (with little surprise) striking regularities and similarities to the model. So we conjectured that there exists a significant sub-language of timed automata that is *implementable*. In our previous work [5], we identified independency from a global scheduler as a useful precondition for the existence of a distributed implementation (cf. Section 2).

For this work, we have modelled the LZ-protocol of sensors in the wireless fire alarm system from [15] as an implementable timed automaton (cf. Figure 1; Figure 7 shows the timed automaton obtained by applying the scheme from Figure 2). Hence our modelling language supports real-world, industry case-studies. Implementable timed automata also subsume some models of time-triggered, periodic tasks that we would model by internal edges only.

From the program obtained by the translation scheme given in Table 3, we have derived an implementation of the protocol in C. Clock, data, location, edge, and message variables become enumerations or integers, time variables use the POSIX data-structure `timespec`. The implementation runs timely for multiple days. Although our approach with sleeping to absolute times reduces the risk of drift, there is jitter on real-world platforms. The impact of timing imprecision needs to be investigated per application and platform when refining the program of a network to code, e.g., following [11]. In our case study, jitter is much smaller than the model's time unit. Another strong assumption that we use is synchrony of the platform clocks and synchronised starting times of programs which can in general not be achieved on real-world platforms. In the wireless

fire alarm system, component clocks are synchronised in an initialisation phase and kept (sufficiently) synchronised using system time information in messages. Robustness against limited clock drift is obtained by including so-called *guard times* [23, 24] in the protocol design. In the model, this is constant g: Components are ready to receive g time units before message transmission starts in another component.

Note that Theorem 1 only applies to time-safe computations. Whether an implementation is time-safe needs to be analysed separately, e.g., by conducting worst-case execution time (WCET) analyses of the work code and the code that implements the timed automata semantics. The C code for the LZ-model mentioned above actually implements a *sleepto* function that issues a warning if the target time has already passed (thus indicating non-time-safety). The translation scheme could easily be extended by a statement between Lines 2 and 3 that checks whether the deadline was kept and issues a warning if not. Then, Theorem 1 would strengthen to the statement that all computations of $P(\mathcal{I})$ either correspond to observable behaviour of $\mathcal{I}$ or issue a warning. Note that, in contrast to [1, 2, 31], our approach has the practically important property that time-safety implies time-robustness, i.e., if a program is time-safe on one platform then it is time-safe on any 'faster' platform. Furthermore, we have assumed a deterministic choice of the next edge to be executed for simplicity and brevity of the presentation. Non-deterministic models can be supported by providing a non-deterministic semantics to the $nextedge_{\mathcal{I}}$ function in the programming language and the correctness proof.

7 Conclusion

We have presented a shorthand notation that defines a subset of timed automata that we call implementable. For networks of implementable timed automata that do not depend on a global scheduler, we have given a translation scheme to a simple, exact-time programming language. We obtain a distributed implementation with one program for each network component, the programs are supposed to be executed concurrently, possibly on different computers. We propose to not substitute (imprecise) platform clocks for (model) clocks in guards and invariants, but to rely on a sleep function with absolute deadlines. The generated programs do not include any "hidden" execution times, but all updates, actions, and the time needed to select subsequent edges are taken into account. For the generated programs, we have established a notion of correctness that closely relates program computations to computation paths of the network. The close relation lowers the mental burden for developers that is induced by other approaches that switch to a slightly different, e.g., robust semantics for the implementation.

Our work decomposes the translation from timed automata models to code into a first step that deals with the discrepancy between atomicity of the timed automaton semantics and the non-atomic execution on real platforms. The second step, to relate the exact-time program to real platforms with imprecise timing is the subject of future work.

References

1. Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: Carloni, L.P., Tripakis, S. (eds.) EMSOFT. pp. 229–238. ACM (2010), https://doi.org/10.1145/1879021.1879052
2. Abdellatif, T., Combaz, J., Sifakis, J.: Rigorous implementation of real-time systems - from theory to application. Math. Struct. Comput. Sci. 23(4), 882–914 (2013), https://doi.org/10.1017/S096012951200028X
3. Abdullah, J., Mohaqeqi, M., Yi, W.: Synthesis of Ada code from graph-based task models. In: Seffah, A., Penzenstadler, B., Alves, C., Peng, X. (eds.) SAC. pp. 1467–1472. ACM (2017), https://doi.org/10.1145/3019612.3019681
4. Amnell, T., Fersman, E., Pettersson, P., Sun, H., Yi, W.: Code synthesis for timed automata. Nord. J. Comput. 9(4), 269–300 (2002)
5. Arenis, S.F., Vujinovic, M., Westphal, B.: On global scheduling independency in networks of timed automata. In: Abate, A., Geeraerts, G. (eds.) FORMATS. LNCS, vol. 10419, pp. 42–57. Springer (2017), https://doi.org/10.1007/978-3-319-65765-3_3
6. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM. pp. 3–12. IEEE (2006), https://doi.org/10.1109/SEFM.2006.27
7. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT. LNCS, vol. 3185, pp. 200–236. Springer (2004), https://doi.org/10.1007/978-3-540-30080-9_7
8. Benveniste, A., Le Guernic, P., Jacquemot, C.: Synchronous programming with events and relations: the SIGNAL language and its semantics. Sci. Comput. Program. 16(2), 103–149 (1991), https://doi.org/10.1016/0167-6423(91)90001-E
9. Berendsen, J., Vaandrager, F.W.: Compositional abstraction in real-time model checking. In: Cassez, F., Jard, C. (eds.) FORMATS. LNCS, vol. 5215, pp. 233–249. Springer (2008), https://doi.org/10.1007/978-3-540-85778-5_17
10. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Program. 19(2), 87–152 (1992), https://doi.org/10.1016/0167-6423(92)90005-V
11. Bouyer, P., Larsen, K.G., Markey, N., Sankur, O., Thrane, C.R.: Timed automata can always be made implementable. In: Katoen, J., König, B. (eds.) CONCUR. LNCS, vol. 6901, pp. 76–91. Springer (2011), https://doi.org/10.1007/978-3-642-23217-6_6
12. Corbett, J.C., Dean, J., Epstein, M., et al.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. 31(3), 8:1–8:22 (2013), https://dl.acm.org/citation.cfm?id=2491245
13. Fahrenberg, U.: Higher-dimensional timed automata 51(16), 109–114 (2018), https://doi.org/10.1016/j.ifacol.2018.08.019
14. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Automated analysis of AODV using UPPAAL. In: Flanagan, C., König, B. (eds.) TACAS. LNCS, vol. 7214, pp. 173–187. Springer (2012), https://doi.org/10.1007/978-3-642-28756-5_13
15. Feo-Arenis, S., Westphal, B., Dietsch, D., Muñiz, M., Andisha, A.S., Podelski, A.: Ready for testing: ensuring conformance to industrial standards through formal verification. Formal Asp. Comput. 28(3), 499–527 (2016)
16. Feo-Arenis, S., Westphal, B.: Parameterized verification of track topology aggregation protocols. In: Beyer, D., Boreale, M. (eds.) FORTE. LNCS, vol. 7892, pp. 35–49. Springer (2013), https://doi.org/10.1007/978-3-642-38592-6_4

17. Flammini, A., Ferrari, P.: Clock synchronization of distributed, real-time, industrial data acquisition systems. In: Vadursi, M. (ed.) Data Acquisition, chap. 3. IntechOpen, Rijeka (2010), https://doi.org/10.5772/10458
18. Gobriel, S., Khattab, S.M., Mossé, D., Brustoloni, J.C., Melhem, R.G.: Rideshare: Fault tolerant aggregation in sensor networks using corrective actions. In: SECON. pp. 595–604. IEEE (2006), https://doi.org/10.1109/SAHCN.2006.288516
19. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (1991)
20. Hendriks, M.: Translating Uppaal to not quite C (2001), http://repository.ubn.ru.nl/bitstream/handle/2066/19058/19058.pdf?sequence=1
21. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE 91(1), 84–99 (2003)
22. ISO/IEC: 9899:2018, Programming Languages – C, 4th edn. (2018)
23. Jubran, O., Westphal, B.: Formal approach to guard time optimization for TDMA. In: Auguin, M., de Simone, R., Davis, R.I., Grolleau, E. (eds.) RTNS. pp. 223–233. ACM (2013), https://doi.org/10.1145/2516821.2516849
24. Jubran, O., Westphal, B.: Optimizing guard time for TDMA in a wireless sensor network - case study. In: LCN. pp. 597–601. IEEE Computer Society (2014), https://doi.org/10.1109/LCNW.2014.6927708
25. Kristensen, J., Mejlholm, A., Pedersen, S.: Automatic translation from Uppaal to C (2005), http://mejlholm.org/uni/pdfs/dat4.pdf
26. Olderog, E.R., Dierks, H.: Real-time systems - formal specification and automatic verification. Cambridge University Press (2008)
27. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
28. Puri, A.: Dynamical properties of timed automata. Discrete Event Dynamic Systems 10(1-2), 87–113 (2000), https://doi.org/10.1023/A:1008387132377
29. Senthooran, I., Watanabe, T.: On generating soft real-time programs for non-real-time environments. In: Nishizaki, S.y., Numao, M., Caro, J., Suarez, M.T. (eds.) Theory and Practice of Computation, pp. 1–12. Springer Japan, Tokyo (2013)
30. Tirado-Andrés, F., Rozas, A., Araujo, Á.: A methodology for choosing time synchronization strategies for wireless IoT networks. Sensors 19(16), 3476 (2019), https://doi.org/10.3390/s19163476
31. Triki, A., Combaz, J., Bensalem, S., Sifakis, J.: Model-based implementation of parallel real-time systems. In: Cortellessa, V., Varró, D. (eds.) FASE. LNCS, vol. 7793, pp. 235–249. Springer (2013), https://doi.org/10.1007/978-3-642-37057-1_18
32. Wibling, O., Parrow, J., Pears, A.N.: Ad hoc routing protocol verification through broadcast abstraction. In: Wang, F. (ed.) FORTE. LNCS, vol. 3731, pp. 128–142. Springer (2005), https://doi.org/10.1007/11562436_11
33. Wulf, M.D., Doyen, L., Raskin, J.: Almost ASAP semantics: from timed models to timed implementations. Formal Asp. Comput. 17(3), 319–341 (2005), https://doi.org/10.1007/s00165-005-0067-8