Modelling and analysis of real-time coordination patterns

Kemper, S.

Chapter 2

System Models

When designing real-time systems, one—if not *the*—major design decision is the choice of time domain. In [AD94], the authors phrased this problem as finding the answer to the question "What is the nature of time?".

Reasoning on a *discrete* time scale allows time values in the domain of natural numbers $\mathbb{N}$, that means events can happen at equidistant points in time, or multiples thereof. Reasoning on a *continuous* (or *dense*) time scale allows time values in the domain of real numbers $\mathbb{R}_{\geq 0}$, that means events can happen at any positive (to express that time "starts at 0" and does not "run backwards") real-valued point in time. While discrete time is very close to physical hardware implementation of real-time systems (for example, a discrete time step can be taken with every trailing edge), continuous time is a more realistic approach, in that events do not always happen at integer-valued times, even if the base value for "one time unit" is small.

To use a discrete time scale, the occurrence times of events would either have to be restricted to integer values, or be approximated with the "closest" integer value. The former is unrealistic, the latter comprises a serious information loss with respect to sequential order and (a)synchrony of events. Therefore, in this thesis, we choose continuous time as the time domain `Time` of real-time systems, that means,

$$\texttt{Time} = \mathbb{R}_{\geq 0}.$$

In the remainder of this chapter, we introduce the state-based system models that we use to model real-time systems. We begin by introducing some general notions of real-time systems in Section 2.1. In subsequent Sections, we introduce Timed Automata (TA, Section 2.2), Timed Constraint Automata (TCA, Section 2.3), and Timed Network Automata (TNA, Section 2.4). We conclude each Section with a discussion about the advantages and disadvantages of the respective system model. We conclude the Chapter in Section 2.5 with a clear identification of the parts that represent original work in this thesis, and parts that are based on previous results.

2.1 Preliminaries

In this section, we introduce some general notions and concepts for modelling real-time systems.

Notation 2.1.1 (Operator Precedence). To reduce the number of parenthesis, we establish the following precedence rules. Among logical operators, $\neg$ has a higher precedence than $\{\wedge, \vee\}$, which have a higher precedence than $\{\rightarrow, \leftrightarrow\}$. Among arithmetical operators, - (unary minus) has a higher precedence than $\{+, -\}$ (binary minus), which have a higher precedence than $\{<, \leq, =, \geq, >\}$.

For operators with the same precedence, we may still have to add parenthesis, for example, we need to make clear whether $a \wedge b \vee c$ means $(a \wedge b) \vee c$ or $a \wedge (b \vee c)$. We do not need to define precedence rules between arithmetical and logical operators though, since it is clear from the context which variables or constants the operators bind to. For example, for integer variables x_1, x_2, x_3, x_4, it is clear that $x_1 = x_2 \wedge x_3 \geq x_4$ means $(x_1 = x_2) \wedge (x_3 \geq x_4)$.

2.1.1 Time

As explained above, we work with a dense time domain $\texttt{Time} = \mathbb{R}_{\geq 0}$. To measure the passage of time, each model of a real-time systems is equipped with a finite set of real-valued clocks $\mathcal{X}$. All clocks evolve with the same slope 1 (they all "run with the same speed"), i.e., after d time units have passed, the value of each clock has advanced by d. Components of the system may be associated with clock constraints, which restrict the behaviour of the system. The syntax of clock constraints is defined as follows

Definition 2.1.2 (Clock Constraint). Let $\mathcal{X}$ be a finite set of real-valued variables, called clocks. *Clock constraints* $cc \in \mathcal{CC}(\mathcal{X})$ *over* $\mathcal{X}$ are defined as follows:

$$cc ::= \texttt{true} \mid x \sim c \mid x - y \sim c \mid cc_1 \wedge cc_2, \qquad \text{with } x, y \in \mathcal{X}, c \in \mathbb{Z}, \text{ and } \sim \in \{<, \leq, =, \geq, >\}$$

We write $\mathcal{X}|_{cc} \subseteq \mathcal{X}$ to denote the set of clocks that occur in a clock constraint cc.

Remark 2.1.3 (Time Domain in Clock Constraints). Though clocks are real-valued, we need to restrict the domain of constants in clock constraints, in order to obtain/preserve decidability results (cf. [AD94]). For example, with real-valued constants, reachability would become undecidable.

To preserve decidability results, it would be enough to restrict the domain to $\mathbb{Q}$ (rational numbers) though. Yet, for simplicity, we further restrict it to integer numbers. This does not reduce expressiveness (with respect to rational numbers): if clock constraints involve rational constants, we can multiply all constants by the least common multiple of their denominators to obtain clock constraints with integer constants only.

The validity ("semantics") of clock constraints is evaluated under a certain valuation of the clock variables.

Definition 2.1.4 (Clock Valuation). Let $\mathcal{X}$ be a finite set of clocks, $X \subseteq \mathcal{X}$, $x \in \mathcal{X}$ and $t \in \mathtt{Time}$. A *clock valuation* $\nu \in \mathcal{V}(\mathcal{X})$ *over* $\mathcal{X}$ is a mapping $\nu : \mathcal{X} \to \mathtt{Time}$, assigning to each clock $x \in \mathcal{X}$ an element from the time domain Time, its current value. The *restriction* $\nu|_X$ *of* ν *from* $\mathcal{X}$ *to* X is a valuation that agrees with ν on clocks $x \in X$, and is undefined otherwise, that means $\nu|_X(x) = \nu(x)$ iff $x \in X$, and $\nu|_X(x) =$ undefined otherwise.

We may write $\mathcal{V}$ if $\mathcal{X}$ is clear from the context.

We use $\models$ for the standard satisfaction relation on clock constraints. For example, $\nu \models (x \sim c)$ iff $\nu(x) \sim c$, $\sim \in \{<, \leq, =, \geq, >\}$.

To model the semantics of real-time systems, we need the following operations on valuations.

Definition 2.1.5 (Timeshift, Update). Let $\mathcal{X}$ be a finite set of clocks, $x, y \in \mathcal{X}$, $t \in \mathtt{Time}$ and $\nu \in \mathcal{V}(\mathcal{X})$.

The *timeshift operation* $\nu + t$ (or simply *timeshift*) increases (with respect to ν) the values of all clocks simultaneously by the same amount of time t, that means, $(\nu + t)(x) = \nu(x) + t$ for all $x \in \mathcal{X}$.

An *update map* $\lambda \in \Lambda(\mathcal{X})$ *over* $\mathcal{X}$ [AM04] is a mapping $\lambda : \mathcal{X} \to (\mathcal{X} \cup \mathbb{N})$. The *update operation* $\nu[\lambda]$ (or simply *update*) modifies the values of clocks under valuation ν, by either setting them to the value of another clock or to a natural number, according to the update map λ (λ is the identity for clocks not meant to be modified). That is, $\nu[\lambda](x) = \nu(y)$ iff $\lambda(x) = y$,[1] and $\nu[\lambda](x) = n$ iff $\lambda(x) = n \in \mathbb{N}$.

We do not allow negation on clock constraints (cf. [Alu99, AM04]), which results in clock constraints being *convex*.

Remark 2.1.6 (Convexity of Clock Constraints). Clock constraints defined according to Definition 2.1.2 are *convex* under timeshift and update (Definition 2.1.5), for any clock valuation $\nu \in \mathcal{V}(\mathcal{X})$ (Definition 2.1.4).

Intuitively, convexity of clock constraints means that if the value of a clock satisfies the same clock constraint under two different valuations, then it also satisfies the clock constraint for all valuations "in between". Formally: for a clock constraint cc and two valuations ν and ν' over clock $x \in \mathcal{X}|_{cc}$, such that $\nu(x) < \nu'(x)$, if $\nu \models cc$ and $\nu' \models cc$, then $\nu'' \models cc$ for all ν'' with $\nu(x) < \nu''(x) < \nu'(x)$.

Convexity of clock constraints is an important property used for efficient representation (and verification) of real-time systems in Chapter 3.

Note that we do not impose any semantic constraints on clock constraints. For example, they may "overlap" on a clock, like $(x \leq 4) \wedge (x \geq 3)$. Yet, due to convexity, this simply reduces the number of satisfying valuations.

[1] In case of an update $\lambda(x) = y$, $\lambda(y) = n$, the equation $\nu[\lambda](x) = \nu(y)$ ensures that x is assigned the value of y *before* y is updated.

2.1.2 Data

Communication in real-time systems may involve exchange of data values. We assume a global (possibly infinite but) countable data domain $\mathcal{D}ata$, with a special element $\bot \in \mathcal{D}ata$ representing "no data", which we use in Chapter 3 to explicitly represent *absence of data*. Real-time systems use a finite set of ports $\mathcal{P}$, through which they exchange data values with the environment. Further, they can make use of a finite set of data variables $\mathcal{D}$. The intended idea is that ports are used to exchange data values with the environment, while data variables are used to store and exchange data values within the real-time system.

To restrict the admissible data values to be exchanged, components of real-time systems may be associated with data constraints. These restrict the behaviour of the system, by reasoning about the data values exchanged through ports, or stored in data variables. The syntax of data constraints is defined as follows.

Definition 2.1.7 (Data Constraint). Let $\mathcal{P}$ be a finite, nonempty set of ports, $\mathcal{D}$ a finite, nonempty set of data variables, $\mathcal{D}ata$ a data domain. *Data constraints* $dc \in DC(\mathcal{P} \cup \mathcal{D})$ *over* $\mathcal{P}$ *and* $\mathcal{D}$ are defined as

$$dc ::= \texttt{true} \mid \mathbf{d}{=}\mathbf{d}' \mid dc_1 \wedge dc_2 \mid \neg dc, \text{ with } \mathbf{d}, \mathbf{d}' \in \mathcal{P} \cup \mathcal{D} \cup \mathcal{D}ata$$

If there exists a total order $\leqslant$ on $\mathcal{D}ata \backslash \bot$, we also allow data constraints of the form

$$dc ::= \mathbf{d} \leqslant \mathbf{d}', \text{ with } \mathbf{d}, \mathbf{d}' \in \mathcal{P} \cup \mathcal{D} \cup \mathcal{D}ata \backslash \bot$$

If on top of $\leqslant$ there exists an operation $+$ (addition) on $\mathcal{D}ata \backslash \bot$,[2] we also allow data constraints of the form

$$dc ::= \mathbf{D} \sim \mathbf{D}', \text{ with } \sim \in \{=, \leqslant\}, \text{ and}$$
$$\mathbf{D} ::= \mathbf{d} \mid \mathbf{D_1} + \mathbf{D_2} \mid (\mathbf{D_1} - \mathbf{D_2}), \text{ with } \mathbf{d}, \mathbf{d}' \in \mathcal{P} \cup \mathcal{D} \cup \mathcal{D}ata \backslash \bot$$

We write $\mathcal{D}|_{dc} \subseteq \mathcal{D}$ to denote the set of data variables that occur in a data constraint dc. Equivalently, $\mathcal{P}|_{dc} \subseteq \mathcal{P}$ denotes the set of ports that occur in dc, and $\mathcal{D}ata|_{dc} \subseteq \mathcal{D}ata$ denotes the set of data values that occur in dc.

We may write $DC(\mathcal{P},\mathcal{D})$ instead of $DC(\mathcal{P} \cup \mathcal{D})$, and we use $DC(\mathcal{P})$ as a shorthand for $DC(\mathcal{P}, \emptyset)$, equivalently we use $DC(\mathcal{D})$ as a shorthand for $DC(\emptyset, \mathcal{D})$.

Other data constraints, like for example $p \in A$ (for some set $A \subseteq \mathcal{D}ata$), $dc_1 \vee dc_2$, or $dc_1 \rightarrow dc_2$, are defined as abbreviations ("syntactic sugar") in the standard way.

The validity ("semantics") of data constraints is evaluated under a certain data assignment. Data assignments describe the data values which are pending at ports, or stored in data variables.

[2] Formally, $+$ is required to be an operation such that $(\mathcal{D}ata \backslash \bot, +)$ is an abelian group, i.e. a group which satisfies the abelian group axioms (1) closure, (2) associativity, (3) existence of identity element, (4) existence of inverse element, and (5) commutativity. As usual, the operation "$-$" (negation) is a shorthand for addition of the inverse.

Definition 2.1.8 (Data Assignment). Let $\mathcal{P}$ and $\mathcal{D}$ be as in Definition 2.1.7, and $\mathcal{D}ata$ a data domain. A *data assignment* $\delta \in DA(\mathcal{P} \cup \mathcal{D})$ *over* $\mathcal{P}$ *and* $\mathcal{D}$ is a mapping $\delta : (\mathcal{P} \cup \mathcal{D}) \to \mathcal{D}ata$, assigning to each port $p \in \mathcal{P}$ the data value which is currently pending at p, and to each data variable $d \in \mathcal{D}$ the data value which is currently contained in d.

If $\delta(p) = \bot$ ("no dataflow through p"), p is called *inactive*. Otherwise, p is called *active*. If $\delta(d) = \bot$, d is called *empty*.

The *restriction* $\delta|_{\mathcal{A}}$ *of* δ *from* $(\mathcal{P} \cup \mathcal{D})$ *to any subset* $\mathcal{A} \subseteq \mathcal{P} \cup \mathcal{D}$ is a data assignment that agrees with δ on elements $a \in \mathcal{A}$, and is undefined otherwise.

We may write $DA(\mathcal{P}, \mathcal{D})$ instead of $DA(\mathcal{P} \cup \mathcal{D})$, and we use $DA(\mathcal{P})$ as a shorthand for $DA(\mathcal{P}, \emptyset)$, equivalently we use $DA(\mathcal{D})$ as a shorthand for $DA(\emptyset, \mathcal{D})$.

Definition 2.1.7 allows for trivial data constraints involving only data constants (i.e., elements from $\mathcal{D}ata$), for example $\mathbb{d}_1 = \mathbb{d}_2$, with $\mathbb{d}_1, \mathbb{d}_2 \in \mathcal{D}ata$. Since the validity of such data constraints does not depend on a specific data assignment δ, they can be evaluated statically (to either `true` or $\texttt{false} \stackrel{\text{def}}{=} \neg\texttt{true}$). Therefore, we assume that every data constraint involves at least one port or one data variable. We use $\models$ for the standard satisfaction relation of data assignments on data constraints. For example, $\delta \models (p = q)$ iff $\delta(p) = \delta(q)$, and $\delta \models (p \leqslant q)$ iff $\delta(p) \leqslant \delta(q)$.

Remark 2.1.9 (Use of $\bot$ in Data Constraints). Notice that we only allow the special value $\bot$ in simple data constraints of the form $(\mathbf{d} = \mathbf{d}')$. The idea is that a data constraint $(p = \bot)$, with $p \in \mathcal{P}$, represents a "check" whether port p is inactive. Equivalently, a data constraint $(d = \bot)$, with $d \in \mathcal{D}$, represents a check whether data variable d is empty.

We do not allow $\bot$ to be used in combination with $\leqslant$, since it is not clear how to define the result of such a comparison. One possible solution would be to define $\bot$ as supremum or infimum of $\mathcal{D}ata$; yet, the constraints involving $\bot$ could be simplified to `true` or `false` in this case. Apart from that, many countably infinite sets have neither a supremum nor an infimum (take for example $\mathbb{Z}$), and actually we do not consider a comparisons of the form $(\mathbb{d} \leqslant \bot)$, with $\mathbb{d} \in \mathcal{D}ata$, useful at all.

A similar argumentation holds for the use of $\bot$ in combination with $+$.

2.2 Timed Automata

In this section, we present the first and most basic system model for modelling real-time systems: Timed Automata.

Timed Automata (TA) were introduced in the seminal paper of Alur and Dill in 1994 [AD94], and have been studied intensively since. TA are finite automata, extended with real-valued clock variables, that can measure the passing of time. Their behaviour consists of a sequence of events (or actions) happening over time. Conceptually, this is represented by an infinite sequence of events, which is paired with an infinite sequence of time instants, with the intended meaning that a specific events takes place at the specific time.

To model this behaviour, TA comprise two kinds of events: visible (external) and invisible (internal) events. The former are used for synchronisation with other automata, while the latter are used for internal activities of a single automaton, independent from others.

The underlying idea is that transitions (location changes) are instantaneous, time may only elapse while the automaton remains in one of its locations. The firing of transitions, and the dwell time in locations, are restricted by constraints on the clocks—called clock guards and clock invariants, respectively—which the current clock values have to satisfy. That means, a TA is only allowed to fire a transition if the associated clock guard is satisfied, and is only allowed to stay in a location as long as the associated clock invariant is satisfied. In addition, transitions may update the values of (a subset of the) clocks to a natural number or to the value of another clock [AM04].

In the literature, many slightly different variants of TA can be found. Our definitions are essentially based on [AD94]. Please refer to Section 2.2.4 for a discussion of other variants.

2.2.1 Syntax of Timed Automata

Each transition of a TA is labelled with a distinct event, which captures "what is being performed" when the transition is fired. In this work, we distinguish between two types of events: visible (external) and invisible (internal) actions, cf. [BK08, KP07, AM04]. The former are used to synchronise with other automata when considering systems of TA, while the latter are used for internal steps of a single automaton, independent from other automata. Since internal actions can be regarded as "being of no further interest" [BK08], they are commonly denoted by the single distinguished action symbol τ. We denote the set of visible events of a TA by Σ_v, and the set of all events (visible and invisible) by Σ, i.e., $\Sigma{=}\Sigma_v \dot\cup \tau$. For a discussion of other possibilities to define the set of admissible events, please refer to Section 2.2.3.

Recalling Definitions 2.1.2, 2.1.4 and 2.1.5, we define the syntax of TA as follows.

Definition 2.2.1 (Timed Automaton). A *TA* is a tuple $\mathfrak{A}{=}(S, s_0, \Sigma, \mathcal{X}, I, E)$, with S a finite set of locations, $s_0{\in}S$ the initial location, Σ a finite set of visible events, $\mathcal{X}$ a finite set of real-valued clocks, $I{:}S{\to}\mathcal{CC}(\mathcal{X})$ a function assigning a clock constraint (*location invariant*) to every location, and $E{\subseteq}(S{\times}\Sigma{\times}\mathcal{CC}(\mathcal{X}){\times}\Lambda(\mathcal{X}){\times}S)$ the finite set of transitions.

The idea of transitions of TA is as follows: an element $e{=}(s, \mathfrak{a}, cc, \lambda, s'){\in}E$ describes a transition from the *source location* s to the *target location* s' on occurrence ("execution") of *action* $\mathfrak{a}$. The firing of the transition is restricted by the *(clock) guard* cc, and updates clocks according to the *update map* λ. For every such transition, we require cc to be satisfiable. If $\mathfrak{a}{\in}\Sigma_v$, we call e an *external (or visible)* transition, otherwise (i.e., if $\mathfrak{a}{=}\tau$), e is called *internal (or invisible).*

Example 2.2.2 (Timed Automaton). Two examples for TA are given in Figures 2.1 and 2.2. The TA in Figure 2.1 models an "intelligent light switch": it consists of three locations *off*, *light* and *bright*, representing the corresponding states of the light, and a clock x. In the initial location (marked by the incoming arrow), the light is off. If the switch is pressed (modelled by action *press*), the light turns on (location *light*), and the clock x is reset to measure the temporal difference to the next *press* action. If the switch is pressed again before the value of x reaches 3 (modelled by the guard $x{\leq}3$), the light becomes bright, otherwise (i.e., if the value of x is greater than 3, modelled by the guard $x{>}3$), it switches off again.

The automaton in Figure 2.2 shows a "user" of the light switch. The automaton consists of a single location and a clock y. The location has an invariant $y{\leq}4$, which forces the automaton to leave the location after having delayed there for at most 4 time units. Together with the guard $y{\geq}2$ and the update $y{:=}0$ on the transition, this models a user which executes the *press* action every 2 to 4 time units.

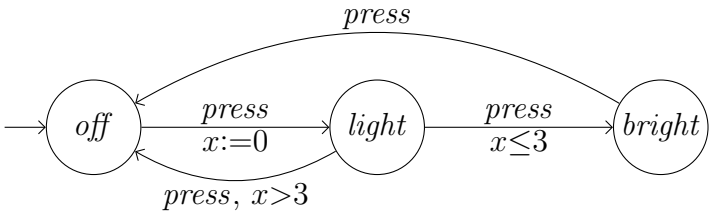


Figure 2.1: Intelligent Light Controller

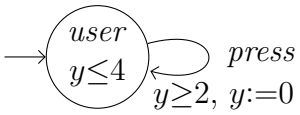


Figure 2.2: User pressing the Light Switch

In the graphical representation of TA, we use assignment rather than functional notation for the updates, and we omit guards and invariants equal to `true` as well as identity updates of the form $\lambda(x){=}x$.

Notation 2.2.3 (TA). If not stated otherwise, we shall assume the constituents of a TA $\mathfrak{A}$ to be denoted as $\mathfrak{A}{=}(S, s_0, \Sigma, \mathcal{X}, I, E)$, and of a TA $\mathfrak{A}_i$ to be denoted as $\mathfrak{A}_i{=}(S_i, s_{0,i}, \Sigma_i, \mathcal{X}_i, I_i, E_i)$, for $i{\in}\mathbb{N}$.

By $\mathcal{CC}(\mathcal{X})|_{\mathfrak{A}}$, we denote the set of clock constraints (over clock set $\mathcal{X}$) that occur in a TA $\mathfrak{A}$ (invariants or guards).

2.2.2 Semantics of Timed Automata

As mentioned above, the idea of TA is that transitions are instantaneous, time only elapses while the automaton remains in one of its locations. The semantics of a TA $\mathfrak{A}$ is defined as the set of runs of the associated labelled transition system (LTS) $\mathfrak{S}_{\mathfrak{A}}$, cf. for example [BK08].

Definition 2.2.4 (Associated Labelled Transition System). Let $\mathfrak{A}$ be a TA. The *associated LTS* $\mathfrak{S}_{\mathfrak{A}}$ is a tuple $\mathfrak{S}_{\mathfrak{A}}{=}(\mathcal{Q}, q_0, \rightarrow)$, with $\mathcal{Q}{\subseteq}(S{\times}\mathcal{V}(\mathcal{X}))$ the set of *configurations*, such that $\nu{\models}I(s)$ for every $(s,\nu){\in}\mathcal{Q}$, $q_0{=}\langle s_0, \mathbf{0}\rangle$ the *initial configuration*, with $\mathbf{0}(x){=}0$ for all $x{\in}\mathcal{X}$, and the transition relation $\rightarrow\,{\subseteq}(\mathcal{Q}{\times}(\mathtt{Time}{\cup}\Sigma){\times}\mathcal{Q})$ is given in (2.1) and (2.2).

$$\frac{(s,\mathfrak{a},cc,\lambda,s')\in E,\quad \nu\models cc,\nu[\lambda]\models I(s')}{\langle s,\nu\rangle\xrightarrow{\mathfrak{a}}\langle s',\nu[\lambda]\rangle}\qquad(2.1)$$

$$\frac{s\in S,t\in\mathtt{Time},t>0,\quad \forall t',t\geq t'\geq 0:\nu+t'\models I(s)}{\langle s,\nu\rangle\xrightarrow{t}\langle s,\nu+t\rangle}\qquad(2.2)$$

A *run of* $\mathfrak{S}_{\mathfrak{A}}$ *(starting in configuration* q*)* is an infinite sequence of transitions $q_0\xrightarrow{\mathfrak{a}_0}q_1\xrightarrow{\mathfrak{a}_1}\ldots$, with $\mathfrak{a}_i\in(\Sigma\cup\mathtt{Time})$ for all $i>0$. A run is called *initial* if it starts in the initial configuration q_0, it is called *loop-free* if all configurations are different.

Rule (2.1) describes an *action transition* (with action $\mathfrak{a}$) of $\mathfrak{S}_{\mathfrak{A}}$, based on a transition $e\in E$ of $\mathfrak{A}$. The valuation ν of the source configuration $\langle s,\nu\rangle$ needs to satisfy ("enable") the clock guard cc, and the updated valuation $\nu[\lambda]$ after the execution of the transition needs to satisfy the invariant of location s' (otherwise, the automaton could not enter location s'). On execution of the transition, the values of the clocks of $\mathfrak{A}$ are updated according to the update map λ. Rule (2.2) describes a *delay transition* (in location s) of $\mathfrak{S}_{\mathfrak{A}}$: the invariant $I(s)$ of the location needs to be satisfied at all times (for all t'), and the clock values of all clocks increase by the same amount of time t.

Definition 2.2.5 (Semantics of Timed Automata). Let $\mathfrak{A}$ be a TA, $\mathfrak{S}_{\mathfrak{A}}$ the associated LTS as defined in Definition 2.2.4. The *trace semantics of* $\mathfrak{A}$ is given by the set $Run_{\mathfrak{A}}$ of initial runs of $\mathfrak{S}_{\mathfrak{A}}$. With $Run_{\mathfrak{A},k}$, we denote the set of finite prefixes of elements of $Run_{\mathfrak{A}}$ of (at most) length k.

Note that we do not require clock guards of outgoing transitions of a location to be complete, i.e., cover every possible valuation. This may lead to so-called *timelocks* [Tri99]: consider a location with invariant $x\leq 3$ and a single outgoing transition with clock guard $x\leq 2$. The location cannot be left once $\nu(x)>2$, and as soon as $\nu(x)=3$, from a theoretical point of view, time is not allowed to progress anymore, since the automaton is neither allowed stay in the location nor allowed to leave it. However, using the above definitions, such behaviour is excluded from the semantics.

Example 2.2.6 (Run of a Timed Automaton). In (2.3), we show a run of the intelligent light switch from Figure 2.1 of length 10.

$$\begin{aligned}&\langle \mathit{off},x{=}0\rangle\xrightarrow{2}\langle \mathit{off},x{=}2\rangle\xrightarrow{press}\langle \mathit{light},x{=}0\rangle\xrightarrow{2.5}\langle \mathit{light},x{=}2.5\rangle\xrightarrow{press}\\ &\quad\langle \mathit{bright},x{=}2.5\rangle\xrightarrow{1}\langle \mathit{bright},x{=}3.5\rangle\xrightarrow{press}\langle \mathit{off},x{=}3.5\rangle\xrightarrow{press}\\ &\quad\langle \mathit{light},x{=}0\rangle\xrightarrow{6}\langle \mathit{light},x{=}6\rangle\xrightarrow{3}\langle \mathit{light},x{=}9\rangle\xrightarrow{press}\langle \mathit{off},x{=}9\rangle\end{aligned}\qquad(2.3)$$

Convexity of clock constraints (cf. Remark 2.1.6) gives rise to the following property for sequences of delay transitions, cf. [Alu99].

Remark 2.2.7 (Time-Additivity). For two consecutive delay transitions in a run, *time-additivity* holds. That means, for a TA $\mathfrak{A}$ and associated LTS $\mathfrak{S}_{\mathfrak{A}}=(\mathcal{Q},q_0,\rightarrow)$, with configurations $q_1,q_2,q_3\in\mathcal{Q}$, and $t_1,t_2\in\mathtt{Time}$, if $q_1\xrightarrow{t_1}q_2\in\rightarrow$ and $q_2\xrightarrow{t_2}q_3\in\rightarrow$, then also $q_1\xrightarrow{t_1+t_2}q_3\in\rightarrow$.

2.2.3 Systems of Timed Automata

In this section, we present a product construction for TA which is *compositional*, and therefore allows to build complex and/or distributed systems by first designing the individual components separately, and then combining them with the product operation.

The intended idea of a system of TA is that the automata work in parallel, while synchronising via transitions labelled with the same event. In this work, we assume TA to perform *joint broadcast synchronisation* on visible events (cf. [Alu99]). That means, if some visible event $\mathfrak{a}$ occurs, every automaton $\mathfrak{A}_i$ in the system, with $\mathfrak{a}\in\Sigma_i$ ("knowing about $\mathfrak{a}$") must execute a transition labelled with $\mathfrak{a}$, while an automaton $\mathfrak{A}_i$ with $\mathfrak{a}\notin\Sigma_i$ performs a zero-delay step ("nothing"). If, instead, event τ occurs, automata may decide to either execute a transition labelled with τ or do a zero-delay step. Delay steps with delay $t>0$ have to be executed synchronously by all automata. The product automaton for two TA $\mathfrak{A}_1$ and $\mathfrak{A}_2$ is defined as follows.

Definition 2.2.8 (Product of TA). Let $\mathfrak{A}_1, \mathfrak{A}_2$ be TA, with $\mathcal{X}_1\cap\mathcal{X}_2=S_1\cap S_2=\emptyset$ (can be achieved by renaming the constituents in one of the TA). The *product of* $\mathfrak{A}_1$ *and* $\mathfrak{A}_2$ is a new TA $\mathfrak{A}_1\bowtie\mathfrak{A}_2=(S, s_0, \Sigma, \mathcal{X}, I, E)$, with $S=S_1\times S_2$, $s_0=(s_{0,1}, s_{0,2})$, $\Sigma=\Sigma_1\cup\Sigma_2$, $\mathcal{X}=\mathcal{X}_1\cup\mathcal{X}_2$, $I{:}S_1\times S_2\to\mathcal{CC}(\mathcal{X})$ such that for $s=(s_1, s_2)\in S$, $I(s)=I(s_1)\wedge I(s_2)$, and E is defined in (2.4) and (2.5), and the symmetric rule of the latter.

$$\frac{\begin{array}{c}(s_1, \mathfrak{a}, cc_1, \lambda_1, s_1')\in E_1,\\ (s_2, \mathfrak{a}, cc_2, \lambda_2, s_2')\in E_2,\end{array}}{((s_1, s_2), \mathfrak{a}, cc_1\wedge cc_2, \lambda_1\circ\lambda_2, (s_1', s_2'))} \quad (2.4) \qquad \frac{\begin{array}{c}(s_1, \mathfrak{a}, cc_1, \lambda_1, s_1')\in E_1,\\ (\mathfrak{a}\notin\Sigma_2) \text{ or } (\mathfrak{a}=\tau), s_2\in S_2\end{array}}{((s_1, s_2), \mathfrak{a}, cc_1, \lambda_1, (s_1', s_2))} \quad (2.5)$$

Rule (2.4) describes synchronisation, that means the automata execute a transition labelled with the same event (note that this can be a visible event as well as the internal action τ) in parallel. The resulting transition in the product involves a location change in both underlying TA, the transition is guarded by the combined guard $cc_1\wedge cc_2$, and clocks are updated according to the combined update maps $\lambda_1\circ\lambda_2$. Note that since $\mathcal{X}_1\cap\mathcal{X}_2=\emptyset$, we have $\lambda_1\circ\lambda_2=\lambda_2\circ\lambda_1$, see also Proposition 2.2.9. Rule (2.5) describes the execution of a local transition (visible or invisible) in one of the TA, while the other automaton remains in its current location.

Proposition 2.2.9 (Product of TA). The product of TA is commutative and associative, up to isomorphy of location names.

Proof.

1. Commutativity follows from the commutativity of $\wedge$ on clock constraints, and the commutativity of $\circ$ (function composition) on update maps over disjoint clock sets.

2. Associativity follows from the associativity of $\wedge$ on clock constraints, and the associativity of $\circ$ on update maps over disjoint clock sets. □

Example 2.2.10 (Product of TA). An example for the product construction can be found in Figure 2.3. The automaton shows the product TA for the intelligent light switch and user, as presented in Example 2.2.2.

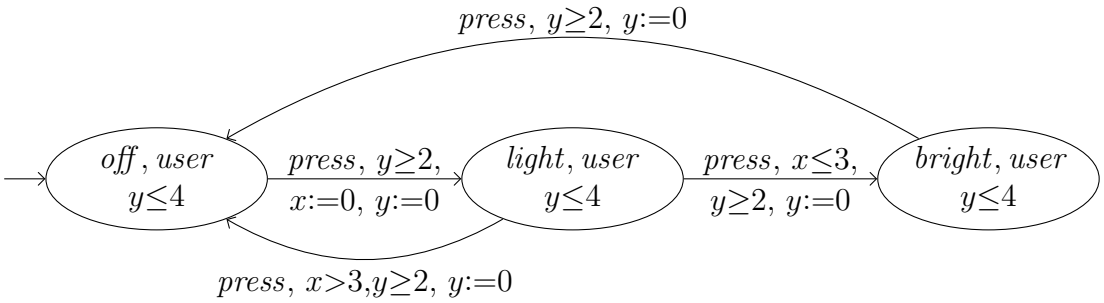


Figure 2.3: Product of Light Switch and User

The semantics of a system of two TA $\mathfrak{A}_1$ and $\mathfrak{A}_2$ (with $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$ and $S_1 \cap S_2 = \emptyset$, as required in Definition 2.2.8) is defined as the semantics (Definition 2.2.5) of the corresponding product automaton $\mathfrak{A}_1 \bowtie \mathfrak{A}_2$ (Definition 2.2.8), i.e., the set of runs of the associated LTS $\mathfrak{S}_{\mathfrak{A}_1 \bowtie \mathfrak{A}_2}$ (Definition 2.2.4). In (2.6), we show a run of the product automaton from Figure 2.3 of length 9.

$$\begin{aligned}
&\langle (\mathit{off}, \mathit{user}), \tfrac{x=0}{y=0} \rangle \xrightarrow{2} \langle (\mathit{off}, \mathit{user}), \tfrac{x=2}{y=2} \rangle \xrightarrow{\mathit{press}} \langle (\mathit{light}, \mathit{user}), \tfrac{x=0}{y=0} \rangle \xrightarrow{2.5} \\
&\quad \langle (\mathit{light}, \mathit{user}), \tfrac{x=2.5}{y=2.5} \rangle \xrightarrow{\mathit{press}} \langle (\mathit{bright}, \mathit{user}), \tfrac{x=2.5}{y=0} \rangle \xrightarrow{1} \\
&\quad \langle (\mathit{bright}, \mathit{user}), \tfrac{x=3.5}{y=1} \rangle \xrightarrow{\mathit{press}} \langle (\mathit{off}, \mathit{user}), \tfrac{x=3.5}{y=0} \rangle \xrightarrow{\mathit{press}} \langle (\mathit{light}, \mathit{user}), \tfrac{x=0}{y=0} \rangle \xrightarrow{4} \\
&\quad \langle (\mathit{light}, \mathit{user}), \tfrac{x=4}{y=4} \rangle \xrightarrow{\mathit{press}} \langle (\mathit{off}, \mathit{user}), \tfrac{x=4}{y=0} \rangle
\end{aligned} \tag{2.6}$$

Remark 2.2.11 (Size of the Product). The product automaton is exponential in the size of the underlying TA in the worst case. Therefore, for model checking/verification, we provide a technique to avoid the explicit construction of the exponential cross product in Section 3.1.2.

2.2.4 Discussion

TA were introduced a long time ago, and are by now well-studied. As a result, TA have been extended and adapted in many ways and for many purposes, like for example Timed Automata with Deadlines [BS00, GS05], Task Automata [FKPY07], or Probabilistic Timed Automata [Bea03, KNSS02]. Yet, even for the "basic" version of TA, slightly different variants of how to define them can be found in the literature, each of which has advantages and disadvantages. We now discuss the major distinctions, and motivate our design decisions.

2.2.4.1 Internal Actions

The original work [AD94, Alu99] did not consider internal actions, but only considered visible actions in the set of admissible events. As a consequence, every transition of a TA could synchronise with a transition in another TA, if a transition labelled

with the same event was enabled. In this work, in addition to visible events, we also allow for invisible internal actions. We consider this more realistic: typically, the behaviour of a real-time system not only depends on the environment (modelled by synchronisation via visible actions), but also on internal state changes, which cannot be influenced by other TA in any way. Consider for example a deadline expiration, after which the behaviour of the system changes.

It has been shown in [AM04] that internal actions add to the expressive power of TA, i.e., a TA with internal actions is more expressive than a TA without internal actions. As an example, consider the TA in Figure 2.4. The automaton consists of a single location s, a clock x, and can synchronise with other automata via action $\mathfrak{a}$. The automaton requires all actions to occur at integer times, and no two actions occur at the same time: both transitions become enabled exactly one time unit after the system started, modelled by the guard $x{=}1$. At this time, the invariant $x{\leq}1$ forces the automaton to leave location s, by performing either of the transitions. If no synchronisation via action $\mathfrak{a}$ (left transition) is possible, the automaton executes the internal action τ (right transition). Both transitions reset clock x to zero, such that the automaton can reenter location s after the execution of the transition.

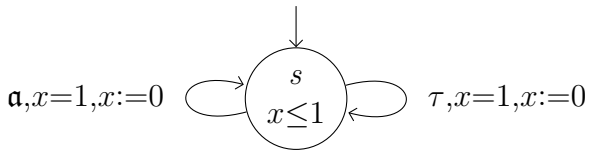


Figure 2.4: Use of Invisible Transitions

This behaviour cannot be modelled with a TA which does not allow for τ-transitions (cf. [AM04]): if the largest constant occurring in guards and invariants was c,[3] the TA could not distinguish between occurrences of action $\mathfrak{a}$ which happen at times $c{+}1$ and $c{+}1.1$ (after the start of the computation).

2.2.4.2 Synchronisation

Within systems of TA, we assume *joint broadcast synchronisation* (cf. Section 2.2.3), that means, on occurrence of a visible event $\mathfrak{a}$, all TA "knowing about" $\mathfrak{a}$ have to synchronise on this event. In particular, there is no restriction on how many automata can participate in a single synchronisation, the number is determined only by the fact whether $\mathfrak{a}$ is contained in the event set of an automaton. Another possibility of defining synchronisation (which is closer to the approach taken in process algebras) is to assume the synchronisation to be *binary*: under this assumption, *exactly two* automata synchronise, i.e., execute a transition labelled with some event $\mathfrak{a}$ in parallel. If more than two automata are ready to synchronise on $\mathfrak{a}$, the choice which pair—i.e., which two automata—executes the synchronisation transitions is made nondeterministically.

The expressiveness of these two ways of modelling synchronisation is the same. We have chosen for joint broadcast synchronisation, since we consider this closer

[3]Such largest constant always exists, since both the set of locations as well as the set of transitions of TA are finite, cf. Definition 2.2.1

to the nature of component-based systems. The ideas of modelling each approach with the respective other can roughly be sketched as follows: using TA with joint broadcast synchronisation to model binary synchronisation is straightforward, by using each action in exactly two automata only. Using TA with binary synchronisation to model joint broadcast synchronisation uses the fact that transitions are instantaneous, therefore, multiple transitions (all labelled with the same action) can take place at the same time instant.

2.2.4.3 Input- and Output Actions

A special case of binary synchronisation is the distinction between input and output actions. While we consider two types of actions (visible/external and invisible/internal), some authors prefer to further divide the set of visible actions into input and output actions, cf. for example [BDL04]. Usually, input and output actions are used to distinguish between actions that are controlled by the TA, and actions that are controlled by the environment. Since we do not make this distinction here, we regard two types of actions (external and internal) to be sufficient.

2.2.4.4 Clock Constraints and Updates

In the original work presented in [AD94, Alu99], so-called *diagonal* clock constraints of the form $x-y\sim c$ (i.e., involving clock differences, cf. Definition 2.1.2) were not allowed. However, it has been shown by Alur and Madhusudan [AM04] that adding these constraints does not add to the expressive power of TA. Therefore, we allow diagonal clock constraints, since they may be used for more concise modelling.

Moreover, [AD94, Alu99] only allowed transitions to reset a (sub)set of clocks to zero. It has been shown in [AM04] that this restriction can be relaxed in favour of more general *update maps*, without adding to the expressiveness. We follow their approach and use the update maps presented in [AM04], cf. Definition 2.1.5.

In contrast to [AD94], but following [Alu99], we do not allow negation in clock constraints, cf. Definition 2.1.2. This is done to obtain convex clock constraints (cf. Remark 2.1.6), which are used for efficient representation in Chapter 3. Yet, non-convex clock constraints can be simulated by splitting locations (for non-convex invariants) and transitions (for non-convex guards).

These considerations on clock constraints and update maps directly carry over to TCA (Section 2.3) and TNA (Section 2.4).

2.3 Timed Constraint Automata

In this section, we define the second system model for modelling real-time systems: Timed Constraint Automata (TCA). TCA [ABdBR07, Kem11] arise from combining the concepts of constraint automata [ABRS04] (CA) and timed automata (TA, Section 2.2). CA were originally defined as a semantical model for the channel-based coordination language $\mathcal{R}\mathsf{eo}$ [Arb04], and consequently, TCA were intended to serve as a semantical model of a timed variant of $\mathcal{R}\mathsf{eo}$. Yet, TCA offer a powerful coordination mechanism for channel-based coordination languages in general. They

are specially tailored for implementing coordinating connectors in networks where timed components communicate by exchanging data through multiple channels. The behaviour of the network is given by synchronisation between channel ends (ports).

For this, TCA allow for two types of transitions: internal (invisible) location changes caused by some timing constraints, and external (visible) transitions representing data flow at some of the ports. Transitions in TCA are labelled with sets of actions (ports). The underlying general idea is that all actions which happen at the same time (i.e., atomically) collapse into a single transition. As a consequence, a positive amount of time elapses before every visible transitions.

The major conceptual difference to TA and other action-based coordination models, like e.g. finite state machines, or timed I/O automata [KLSV03a, KLSV03b], is thus the "non-instantaneous" handling of transitions. While in TA, every transition can be fired immediately (provided that the guard is satisfied), TCA require a positive delay before every visible transition. Moreover, TA permit only a single action per transition, such that synchrony—and concurrent execution in the parallel composition—of different actions is reduced to arbitrary interleavings plus nondeterminism. This is unintuitive, since it imposes a sequential order on actions which conceptually happen at the same time. Moreover, from a technical point of view, the presence of all possible interleavings amplifies the state explosion problem. TCA permit *true concurrency*, as they directly model truly atomic synchronous communication through different ports.

2.3.1 Syntax of Timed Constraint Automata

The syntactic concepts for handling real-time (i.e., clocks, guards, invariants), as defined in the previous section, directly carry over from TA to TCA. For handling data values, each TCA is equipped with a finite set of ports, through which it exchanges data values with other TCA. Transition of TCA are labelled with a subset of these ports, with the intended meaning that data flows through these ports (the ports are active) when the transition is fired. Transitions can be labelled with a data constraint (cf. Definition 2.1.7), restricting the admissible data values flowing through the active ports. In addition, the coordination pattern implemented by a TCA may depend on data values which have been exchanged *before* (i.e., through ports that were active in previous steps). To keep track of this, we extend the basic definition of TCA, presented in [Kem11], with *location memory*, cf. also [PSHA09]: each TCA is equipped with a finite set of data variables, which can be used by locations to store data values, cf. Section 2.1.2. We call these data variables associated to locations *memory cells*.[4]

The underlying idea is that memory cells enable the *current* location to store data values. The same memory cell can be used by both source and target location of the same transition, in this case, its contents carry over unchanged, unless they are updated by the transition. The contents of memory cells which are used by the source locations but not by the target location are discarded, i.e., unused memory

[4]In the remainder of this Section, we may use the terms *memory cell* and *data variable* interchangeably.

cells are empty by definition. Consequently, data constraints on transitions may only refer to memory cells used by the source or target location.

Recalling Definitions 2.1.2, 2.1.4, 2.1.5, 2.1.7 and 2.1.8, TCA are defined as follows.

Definition 2.3.1 (Timed Constraint Automaton). A *TCA over data domain* $\mathcal{D}ata$ is a tuple $\mathfrak{T}=(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, with S a finite set of locations, $s_0 \in S$ the initial location, $\mathcal{P}$ a finite set of ports, $\mathcal{X}$ a finite set of clocks, $I: S \to \mathcal{CC}(\mathcal{X})$ a function assigning a clock constraint (*location invariant*) to every location, $\mathcal{D}$ a finite set of data variables, called *memory cells*, $\#: S \to 2^{\mathcal{D}}$ a function assigning to each location the set of memory cells it may use, and $E \subseteq (S \times 2^{\mathcal{P}} \times \mathcal{DC}(\mathcal{P}, \mathcal{D}) \times \mathcal{CC}(\mathcal{X}) \times \Lambda(\mathcal{X}) \times S)$ the finite set of transitions.

For every element $e=(s, P, dc, cc, \lambda, s') \in E$, we require that the *data guard* dc only reasons about active ports, and only about memory cells of the *source location* s and the *target location* s', i.e., $dc \in \mathcal{DC}(P, \#(s) \cup \#(s'))$. We require both dc and cc (*clock guard*) to be satisfiable. If $P=\emptyset$, transition e is called *invisible*, otherwise, it is called *visible*.

The idea of invisible transitions is that they do not represent observable data flow: the port set P is empty, so no ports are active. The data constraint dc of an invisible transition may thus only reason about memory cells, i.e., $dc \in \mathcal{DC}(\emptyset, \#(s) \cup \#(s'))$. In this way, invisible transitions only serve for internal synchronisation purposes, for example by changing to another location, or updating clocks. Visible transitions, on the other hand, correspond to observable behaviour: on location change from s to s', data flows through all ports in the *port set* P. After the TCA has delayed in location s for a positive amount of time,[5] during which the invariant $I(s)$ of s needs to be satisfied, it executes the transition and moves to location s', provided that the data values pending at the active ports (in P) and contained in memory cells of the source location s satisfy the data guard dc, and the clock values satisfy the *clock guard* cc and the invariant $I(s')$ of the target location s'. The firing of the transition, i.e., the location change from s to s', is considered to be instantaneous. On execution of the transition, all clocks are updated according to the update map λ. These informally described timing constraints will be made explicit in the definition of semantics (Definition 2.3.6).

Remark 2.3.2 (Use of Memory Cells). We do not impose any restrictions on the set of memory cells used by locations. In particular, the same memory cell m may be used by both source and target location of a transition. This may lead to an ambiguous data constraint dc in case dc reasons about m. Yet, such behaviour is necessary for example when it comes to product definition (cf. Definition 2.3.9). To avoid ambiguities, we indicate in the data constraint to which location a memory cell m belongs, i.e., whether the occurrence of m refers to its value before or after the execution of the transition. We add to m the prefix "s." if it refers to the value used

[5] As explained in the beginning of the section, the general idea of TCA is that all data flow actions which happen at the same time atomically collapse to a single transition, therefor, every such transition is preceded by a positive time delay.

by the source location, and the prefix "$t.$" if it refers to the value used by the target location. For example, instead of $(m{=}\mathbb{d}_1){\wedge}(m{=}\mathbb{d}_2)$ (which would obviously evaluate to `false`), we write $(s.m{=}\mathbb{d}_1){\wedge}(t.m{=}\mathbb{d}_2)$. Any data assignment δ will consider $s.m$ and $t.m$ to be different elements.

Our definition allows to leave the value of a memory cell m used by the target location unspecified, i.e., neither does the source location use m, nor does m occur in the data constraint. In this case, we assume m to take a random value.

While unused memory cells are empty be definition (cf. the intuition at the beginning of this section, and also Definition 2.3.5), we may explicitly require memory cells used by the current location to be empty as well, using data constraints of the form $(m{=}\bot)$ (cf. Definition 2.1.7).

Example 2.3.3 (Timed Constraint Automaton). An example for a TCA is given in Figure 2.5. It models a FIFO buffer with capacity 1 and expiration: the buffer consists of two locations *empty* and *full*, uses clock x to measure the time until the data expires, and two ports p and q for exchanging data values. The *full* location has an associated memory cell m, which is denoted $[m]$ in the graphical representation. In the initial location, the buffer is empty. On receiving a data value through port p, the TCA moves to location *full*, updates (resets) clock x to 0, and stores the data value in the location memory of *full*, indicated by the data guard $p{=}t.m$. Since we do not impose any further data constraints, this means that any value can be received through p. If data flow through port q is possible before 3 time units have elapsed (clock guard $x{<}3$), the TCA sends the data item from the memory cell through port q, and moves to back to the initial location. If data flow through q is not possible, at time 3, the invariant of location *full* forces the TCA to leave that location, and execute the invisible transition back to the initial location. No data is transmitted, which models "loosing" the stored data item.

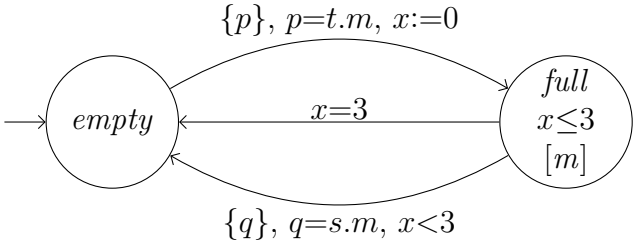


Figure 2.5: 1-bounded FIFO Buffer with Expiration

Notation 2.3.4 (TCA). If not stated otherwise, we shall assume the constituents of a TCA $\mathfrak{T}$ to be denoted as $\mathfrak{T}{=}(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, and of a TCA $\mathfrak{T}_i$ to be denoted as $\mathfrak{T}_i{=}(S_i, s_{0,i}, \mathcal{P}_i, \mathcal{X}_i, I_i, \mathcal{D}_i, \#_i, E_i)$, for $i{\in}\mathbb{N}$. We lift $\#$ to reason about sets of locations, and we may omit curly braces: $\#(s, s') \stackrel{\text{def}}{=} \#(s){\cup}\#(s')$.

By $CC(\mathcal{X})|_{\mathfrak{T}}$, we denote the set of clock constraints (over clock set $\mathcal{X}$) that occur in a TCA $\mathfrak{T}$ (invariants or guards). Equivalently, by $DC(\mathcal{P},\mathcal{D})|_{\mathfrak{T}}$, we denote the set of data constraints over port set $\mathcal{P}$ and memory cells $\mathcal{D}$ that occur in a TCA $\mathfrak{T}$.

As for TA, in the graphical notation of TCA, we use assignment rather than functional notation for updates, and we omit guards and invariants equal to `true`, as well as identity updates. We denote a set of memory cells $\{m_1, \ldots, m_n\}$ associated to a location as a bracketed list $[m_1, \ldots, m_n]$.

2.3.2 Semantics of Timed Constraint Automata

TCA model true concurrency, by allowing sets of ports on each transition. As a consequence, a positive amount of time has to elapse before every visible transition (while invisible transitions may be instantaneous). The underlying idea is that all actions which happen at the same time are truly atomic and thus collapse to a single transition. The semantics of a TCA $\mathfrak{T}$ is defined as the set of runs of the associated labelled transition system (LTS) $\mathfrak{S}_{\mathfrak{T}}$.

Definition 2.3.5 (Associated LTS). Let $\mathfrak{T}$ be a TCA. The *associated LTS* $\mathfrak{S}_{\mathfrak{T}}$ is a tuple $\mathfrak{S}_{\mathfrak{T}}{=}(\mathcal{Q}, q_0, \rightarrow)$, with $\mathcal{Q}{\subseteq}(S{\times}DA(\mathcal{D}){\times}\mathcal{V}(\mathcal{X}))$ the set of configurations, where for every $(s, \delta, \nu){\in}\mathcal{Q}$, $\delta{\in}DA(\mathcal{D})$, with $\delta(m){=}\perp$ if $m{\notin}\#(s)$, and $\nu{\models}I(s)$. The initial configuration is $q_0{=}\langle s_0, \mathbf{0}, \mathbf{0}\rangle$, with $\mathbf{0}(x){=}0$ for all $x{\in}\mathcal{X}$ and $\mathbf{0}(m){=}\perp$ for all $m{\in}\#(s_0)$. The transition relation $\rightarrow\ {\subseteq}(\mathcal{Q}{\times}2^{\mathcal{P}}{\times}DA(\mathcal{P},\mathcal{D}){\times}\mathtt{Time}{\times}\mathcal{Q})$ is given in (2.7) and (2.8).

$$\frac{\begin{array}{l}(s, P, dc, cc, \lambda, s'){\in}E\\ t{>}0, t{\geq}t'{\geq}0 : \nu{+}t'{\models}I(s), \nu{+}t{\models}cc, (\nu{+}t)[\lambda]{\models}I(s')\\ \bar{\delta}{\in}DA(\mathcal{P},\mathcal{D}) : \bar{\delta}{\models}dc\\ \bar{\delta}(s.m){=}\begin{cases}\delta(m), & \text{if } m{\in}\#(s)\\ \perp, & \text{otherwise}\end{cases}\\ \bar{\delta}(t.m){=}\delta(m) \text{ if } m{\in}\#(s), m{\in}\#(s'), t.m{\notin}\mathcal{D}|_{dc}\\ \bar{\delta}(m){=}\perp \text{ if } m{\in}\mathcal{D}{\setminus}\#(s, s'),\\ \bar{\delta}(p){=}\perp \text{ iff } p{\in}\mathcal{P}{\setminus}P,\\ \delta'(m){=}\bar{\delta}(t.m) \text{ for all } m{\in}\#(s')\end{array}}{\langle s, \delta, \nu\rangle \xrightarrow{P,\bar{\delta},t} \langle s', \delta', \nu{+}t[\lambda]\rangle} \tag{2.7}$$

$$\frac{\begin{array}{l}(s, \emptyset, dc, cc, \lambda, s'){\in}E\\ \nu{\models}cc, \nu[\lambda]{\models}I(s')\\ \bar{\delta}{\in}DA(\mathcal{P},\mathcal{D}) : \bar{\delta}{\models}dc\\ \bar{\delta}(s.m){=}\begin{cases}\delta(m), & \text{if } m{\in}\#(s)\\ \perp, & \text{otherwise}\end{cases}\\ \bar{\delta}(t.m){=}\delta(m) \text{ if } m{\in}\#(s), m{\in}\#(s'), t.m{\notin}\mathcal{D}|_{dc}\\ \bar{\delta}(m){=}\perp \text{ if } m{\in}\mathcal{D}{\setminus}\#(s, s'),\\ \bar{\delta}(p){=}\perp \text{ iff } p{\in}\mathcal{P},\\ \delta'(m){=}\bar{\delta}(t.m) \text{ for all } m{\in}\#(s')\end{array}}{\langle s, \delta, \nu\rangle \xrightarrow{\emptyset,\bar{\delta},0} \langle s', \delta', \nu[\lambda]\rangle} \tag{2.8}$$

A *run of* $\mathfrak{S}_{\mathfrak{T}}$ *(starting in configuration* q*)* is a sequence of transitions $q \xrightarrow{P,\delta,t} q_1 \xrightarrow{P',\delta',t'} \dots$ which is either time divergent (i.e. infinite, and $t+t'+\dots=\infty$) or finite and ends in a *terminal configuration* $\langle s, \delta, \nu \rangle$ (i.e. without outgoing transitions, allowing for infinite passage of time: $\forall t{>}0{:}\nu+t \models I(s)$). A run is called *initial* if it starts in the initial configuration q_0, it is called *loop-free* if all configurations are different.

Rule (2.7) captures the constraints described after Definition 2.3.1, for both visible and invisible transitions: before the transition can be fired, a positive amount of time ($t{>}0$) has to elapse, during which the invariant $I(s)$ needs to be satisfied at all times.[6] At time t, the transition is fired, and updates all clocks according to the update map λ, provided that the clock guard cc is satisfied *before* the resetting of clocks, and the invariant of the target location is satisfied (after the time delay and) *after* the resetting of clocks (all second row). The data values have to satisfy the data guard (third row), the values of memory cells before the execution of the transition remain unchanged if used by the source location s, otherwise, they are empty (fourth row). The values of memory cells which are used in both source and target location carry over to s' if they are not modified in the data constraint (fifth row). Note that if a memory cell is not used in the target location, its value $t.m$ after the execution of the transition is unspecified, cf. also Remark 2.3.2. All other memory cells are empty (sixth row), data is only pending at active ports (seventh row), and the data assignment on the transition determines the data values of the memory cells of the target location (eighth row).

Rule (2.8) captures the fact that invisible transitions may be instantaneous; it can be seen as a simplification of (2.7) for $P{=}\emptyset$ and $t{=}0$. Note that data may not be pending at any port (seventh row), yet data constraints of invisible transitions may still reason about memory cells of the involved locations, for example to copy values from one memory cell to another, or to model data loss.

As mentioned in Remark 2.3.2, memory cells which are used by the target location, but are neither used by the source location nor occur in the data constraint, can take a random value. As can be seen from the fifth rows in (2.7) and (2.8), we do not impose any constraints on the values ($t.m$) of these memory cells, which indeed allows $\bar{\delta}$ to assign a random (up to the fact that $\bar{\delta} \models dc$ has to hold) value to them.

Definition 2.3.6 (Semantics of Timed Constraint Automata). Let $\mathfrak{T}$ be a TCA, $\mathfrak{S}_{\mathfrak{T}}$ the associated LTS as defined in Definition 2.3.5. The *trace semantics of* $\mathfrak{T}$ is given by the set $Run_{\mathfrak{T}}$ of initial runs of $\mathfrak{S}_{\mathfrak{T}}$. With $Run_{\mathfrak{T}k}$, we denote the set of finite prefixes of elements of $Run_{\mathfrak{T}}$ of (at most) length k.

Note that as for TA, we do not require clock guards to be complete, nor do we require this for data guards, cf. Page 14.

Example 2.3.7 (Run of a Timed Constraint Automaton). In (2.9), we show a run of the 1-bounded FIFO buffer from Figure 2.5 of length 4.

[6] Due to convexity, these constraints can be relaxed in the representation, since it is enough to check the invariant at the beginning and at the end of the time delay.

$$\langle empty, \bot, x{=}0\rangle \xrightarrow{\{p\},\ p=2,\ t.m=2,\ 4} \langle full, m{=}2, x{=}0\rangle \xrightarrow{\{q\},\ q=2,\ s.m=2,\ 2.5}$$
$$\langle empty, \bot, x{=}2.5\rangle \xrightarrow{\{p\},\ p=5,\ t.m=5,\ 10} \langle full, m{=}5, x{=}0\rangle \xrightarrow{\emptyset,\bot,3}$$
$$\langle empty, \bot, x{=}3\rangle \tag{2.9}$$

Here, we omit data assignments on transitions which evaluate to $\bot$, and we abuse the single symbol $\bot$ to denote the data assignment assigning $\bot$ to all elements.

Remark 2.3.8 (Maximal Progress). The semantics of TCA, as defined above, requires *maximal progress with respect to active ports*: on execution of a (visible) transition, data is pending on all active ports, and on none of the non-active ports, ensured by the constraints $\bar{\delta}(p){=}\bot$ iff $p{\in}\mathcal{P}\backslash P$ in (2.7) and $\bar{\delta}(p){=}\bot$ iff $p{\in}\mathcal{P}$ in (2.8).

For example, if data is pending at ports p and q, and two transitions with port sets $\{p\}$ and $\{p, q\}$, respectively, are ready to fire, then only the latter transition is enabled.

2.3.3 Systems of Timed Constraint Automata

In this section, we present a *compositional* product construction for TCA, which allows to easily build complex connectors by composing simpler ones. We also present a hiding operation on ports, which allows to hide ports on which two (or more) TCA synchronise from the environment. The idea is that the ports become "internal" to the new (composed) connector.

The idea of TCA synchronisation is as follows: within a system of TCA, two automata synchronise (communicate, exchange data values) if the port sets of the involved transitions coincide on common ports. Invisible transitions, and local visible transitions (i.e., transitions involving ports known to only one automaton) can be executed independently of other automata. This gives rise to the following definition.

Definition 2.3.9 (Product of TCA). Let $\mathfrak{T}_1$, $\mathfrak{T}_2$ be TCA over $\mathcal{D}ata_1$ and $\mathcal{D}ata_2$, respectively, with $\mathcal{X}_1{\cap}\mathcal{X}_2{=}\emptyset$, $\mathcal{D}_1{\cap}\mathcal{D}_2{=}\emptyset$, and $S_1{\cap}S_2{=}\emptyset$ (can be achieved by renaming the constituents in one of the TCA). The *product of* $\mathfrak{T}_1$ *and* $\mathfrak{T}_2$ is a new TCA $\mathfrak{T}_1{\bowtie}\mathfrak{T}_2{=}(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$ over data domain $\mathcal{D}ata_1{\cup}\mathcal{D}ata_2$, with $S{=}S_1{\times}S_2$, $s_0{=}(s_{0,1},s_{0,2})$, $\mathcal{P}{=}\mathcal{P}_1{\cup}\mathcal{P}_2$, $\mathcal{X}{=}\mathcal{X}_1{\cup}\mathcal{X}_2$, $I{:}S{\to}\mathcal{CC}(\mathcal{X}_1,\mathcal{X}_2)$, with $I(s){=}I_1(s_1){\wedge}I_2(s_2)$ for $s{=}(s_1, s_2){\in}S$, $\mathcal{D}{=}\mathcal{D}_1{\cup}\mathcal{D}_2$, $\#{:}S{\to}2^{\mathcal{D}}$, with $\#((s_1, s_2)){=}\#_1(s_1){\cup}\#_2(s_2)$, and E is defined in (2.10) and (2.11), and the symmetric rule of the latter.

$$\frac{\begin{array}{c}(s_1, P_1, dc_1, cc_1, \lambda_1, s_1'){\in}E_1\\ (s_2, P_2, dc_2, cc_2, \lambda_2, s_2'){\in}E_2\\ P_1{\cap}\mathcal{P}_2 = P_2{\cap}\mathcal{P}_1, P_1{\neq}\emptyset, P_2{\neq}\emptyset, dc_1{\wedge}dc_2{\neq}\texttt{false}\end{array}}{((s_1, s_2), P_1{\cup}P_2, dc_1{\wedge}dc_2, cc_1{\wedge}cc_2, \lambda_1{\circ}\lambda_2, (s_1', s_2')){\in}E} \tag{2.10}$$

$$\frac{(s_1, P_1, dc_1, cc_1, \lambda_1, s_1'){\in}E_1, P_1{\cap}\mathcal{P}_2 = \emptyset, s_2{\in}S_2}{((s_1, s_2), P_1, dc_1, cc_1, \lambda_1, (s_1', s_2)){\in}E}, \tag{2.11}$$

Rule (2.10) captures the synchronisation of visible transitions: the nonempty port sets have to coincide on common ports, i.e. data flows through the same set of shared ports on both transitions. The case where $P_1 \cap \mathcal{P}_2 = P_2 \cap \mathcal{P}_1 = \emptyset$ (i.e., the set of shared ports is empty) represents a system step where each automaton performs a *local* visible transition (concurrent execution of independent actions). Rule (2.11) describes the execution of a local transition (visible or invisible) in one automaton, while the other automaton remains in its current location (is idle). The semantics of TCA, in particular the fifth and sixth row of (2.7), ensures that the values stored in memory cells of the idle automaton correctly carry over to the next location. Note that in case such a local transition is preceded by a time delay, the idle automaton actually performs a delay transition.

Example 2.3.10 (Product of TCA). An example for the product construction can be found in Figure 2.7: it shows the product TCA of two instances of the 1-bounded FIFO buffer (cf. Example 2.3.3), as show in Figures 2.5 and 2.6 (the two instances are identical except for renaming). The resulting TCA models a FIFO buffer with capacity 2, where each of the buffer cells has its own expiration timer (clocks x and x', respectively). The buffer accepts data through port p, and releases it through port r. The synchronisation on port q models the step where the data from the first buffer cell is transferred to the second buffer cell.

For readability, we have abbreviated the location names in Figure 2.7 with e, e', f and f', for *empty*, *empty*′, *full* and *full*′, respectively.

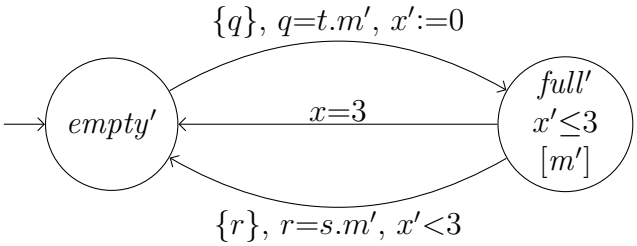


Figure 2.6: 1-bounded FIFO Buffer with Expiration, Second Instance

Proposition 2.3.11 (Product of TCA). The product of TCA is commutative and associative, up to isomorphy of location names.

Proof.

1. Commutativity follows from the commutativity of $\cup$ on port sets, the commutativity of $\circ$ on update maps over disjoint clock sets, and the commutativity of $\wedge$ on data and clock constraints.

2. Associativity follows from the associativity of $\cup$ on port sets, the associativity of $\circ$ on update maps over disjoint clock sets, the associativity of $\wedge$ on data and clock constraints, and the fact that for $P_i \subseteq \mathcal{P}_i$, $i=1,2,3$, if we have $P_2 \cap \mathcal{P}_3 = P_3 \cap \mathcal{P}_2$, $P_1 \cap (\mathcal{P}_2 \cup \mathcal{P}_3) = \mathcal{P}_1 \cap (P_2 \cup P_3)$ and $P_1 \cap \mathcal{P}_2 = P_2 \cap \mathcal{P}_1$, then holds $P_3 \cap (\mathcal{P}_1 \cup \mathcal{P}_2) = \mathcal{P}_3 \cap (P_1 \cup P_2)$.

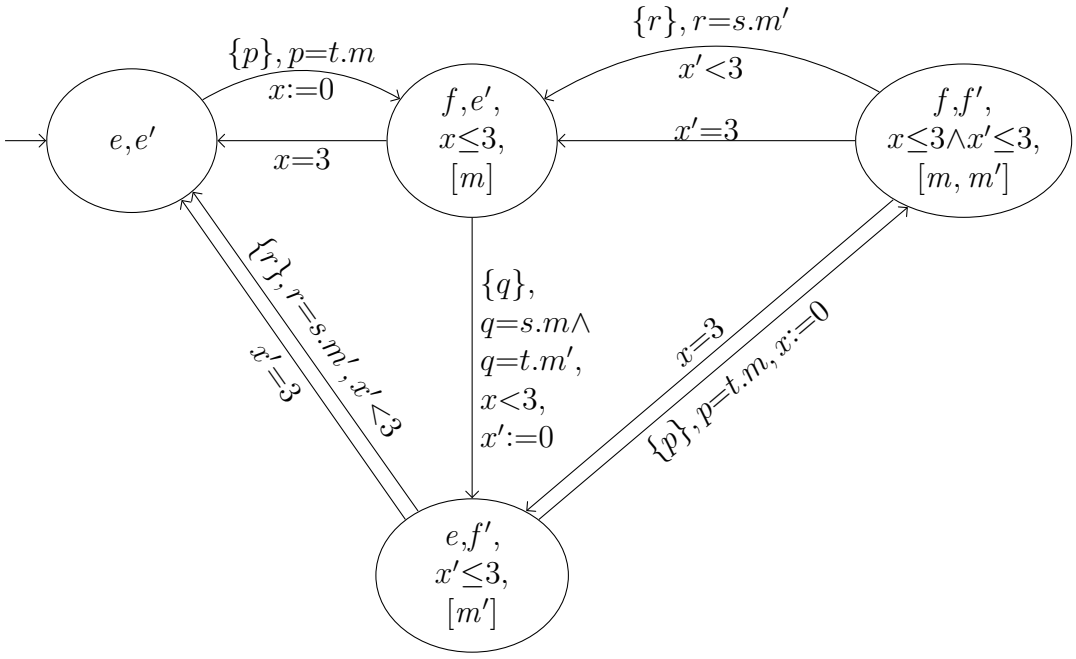


Figure 2.7: Product of two 1-bounded FIFO Buffers

□

It may be required to hide some ports of a TCA from the environment. For example, in the product construction, the common ports (on which the TCA synchronise) could be considered to become "internal" ports, and thus not be visible from the outside anymore. Consider for example port q in Example 2.3.10. The hiding operation removes all information about a set of ports $O{\subseteq}\mathcal{P}$ from a TCA. To ensure correct timed behaviour of transitions with port sets $P{\subseteq}O$—namely that such transitions may only be taken after a positive amount of time—we need to introduce an additional clock.

Definition 2.3.12 (Hiding in TCA). Let $\mathfrak{T}$ be a TCA, $x{\notin}\mathcal{X}$ a fresh clock, and $O{\subseteq}\mathcal{P}$. The *hiding of O in $\mathfrak{T}$* yields a new TCA $\mathfrak{T}\backslash_O{=}(S, s_0, \mathcal{P}\backslash O, \mathcal{X}{\cup}x, I, \mathcal{D}, \#, E')$, where E' is given in (2.12) and (2.13).

$$\frac{(s, P, dc, cc, \lambda, s'){\in}E, ((P{=}\emptyset) \vee (P\backslash O{\neq}\emptyset))}{(s, P\backslash O, dc\backslash_O, cc, \lambda_x, s'){\in}E} \tag{2.12}$$

$$\frac{(s, P, dc, cc, \lambda, s'){\in}E, \emptyset{\neq}P{\subseteq}O}{(s, \emptyset, \texttt{true}, cc{\wedge}(x{>}0), \lambda_x, s'){\in}E'} \tag{2.13}$$

Here, $dc\backslash_O$ denotes the data constraint which is derived from dc by replacing all literals dc', with $dc'{=}(\mathbf{D}{\sim}\mathbf{D}')$ or $dc'{=}{\neg}(\mathbf{D}{\sim}\mathbf{D}')$, by $\texttt{true}$ iff $p{\in}\mathcal{P}|_{dc'}$ (cf. Definition 2.1.7), for all $p{\in}O$. The update map λ_x updates the new clock x to zero, and agrees with λ on other clocks: $\lambda_x(x'){=}\lambda(x')$ if $x'{\in}\mathcal{X}$, and $\lambda_x(x){=}0$ otherwise.

The basic idea is to obtain transitions of $\mathfrak{T}\backslash_O$ from transitions of $\mathfrak{T}$ by reducing the port set (and data constraints) to ports not contained in O (2.12). If the resulting transition in $\mathfrak{T}\backslash_O$ is invisible, while the underlying transition in $\mathfrak{T}$ is visible (2.13), the new clock x is used to ensure correct timed behaviour: since x is updated to zero on all transitions, the additional constraint (x>0) ensures the elapse of a positive amount of time before the (now invisible) transition can be taken.

The semantics of a system of two TCA $\mathfrak{T}_1$ and $\mathfrak{T}_2$ (with disjoint sets of clocks, locations and memory cells, as required in Definition 2.3.9) is defined as the semantics (Definition 2.3.6) of the corresponding product automaton $\mathfrak{T}_1 \bowtie \mathfrak{T}_2$ (Definition 2.3.9), i.e., the set of runs of the associated LTS $\mathfrak{S}_{\mathfrak{T}}$ (Definition 2.3.5). In (2.14), we show a run of the product automaton from Figure 2.7 of length 5.

$$
\begin{aligned}
&\langle (e, e'), \bot, \tfrac{x=0}{x'=0} \rangle \xrightarrow{\{p\},\ \substack{p=3\\ t.m=3},\ 4} \langle (f, e'), m{=}3, \tfrac{x=0}{x'=4} \rangle \xrightarrow{\substack{q=3\\ \{q\},\ s.m=3,\ 2.5\\ t.m'=3}} \\
&\qquad \langle (e, f'), m'{=}3, \tfrac{x=2.5}{x'=0} \rangle \xrightarrow{\substack{p=5\\ \{p\},\ t.m=5\\ s.m'=3,\ 0.5\\ t.m'=3}} \\
&\qquad \langle (f, f'), \tfrac{m=5}{m'=3}, \tfrac{x=0}{x'=0.5} \rangle \xrightarrow{\emptyset,\ \substack{s.m=5\\ t.m=5},\ 2.5} \\
&\qquad \langle (f, e'), m{=}5, \tfrac{x=2.5}{x'=3} \rangle \xrightarrow{\emptyset, \bot, 0.5} \langle (e, e'), \bot, \tfrac{x=3}{x'=3.5} \rangle
\end{aligned}
\tag{2.14}
$$

This run correspond to an execution of the buffer as follows: first, after a delay of 4, it receives data element 3 through port p, and stores it in memory cell m of target location (f,e'). Next, it transfers the data item to memory cell m' of (e,f') through port q, then receives data item 5 through port p and stores it in the memory cell m of target location (f,f'). The buffer is now completely full. In location (f,f'), the buffer delays for 2.5, which—together with previous delays—increases the value of clock x' to 3, such that the buffer takes the invisible transition to location (f,e'), loosing data item 3. After delaying for another 0.5, the value of clock x reaches the threshold 3 as well, the second data item is lost as well, and the buffer returns to the initial location (e,e').

Remark 2.3.13 (Size of the Product). The result of the product construction for TCA, as presented in Definition 2.3.9, is exponential in the worst case. For model checking/verification, we present a linear technique to avoid the explicit construction of the product automaton in Section 3.1.3.

2.3.4 Discussion

TCA arise from combining the real-time concepts of TA, as presented in Section 2.2, with the coordination concepts of constraint automata (CA) [ABRS04]. We further extend the basic definition from [ABdBR07, Kem11] with location memory, which allows to reason about and base the coordination pattern on data values which have been exchanged prior to the current step.

In this way, TCA are specially tailored for implementing coordinating connectors in networks where timed components communicate by exchanging data values through multiple channels. The behaviour of the network is given by synchronisation

between channel ends (ports). While the functionality of channels is often limited to synchrony and (FIFO) buffering, TCA allow connectors with arbitrary behaviour. These connectors provide exogenous coordination, by imposing a certain communication pattern—for example reordering or delays—on associated components. TCA are compositional, which allows to easily build complex connectors out of simpler ones.

One of the major advantages of TCA (which is also one of the major distinctions to TA) is the fact that they provide *true concurrency*. Most action-based (coordination) models, like e.g. finite state machines, timed I/O automata, but also TA, permit only a single action per transition. As a consequence, synchrony, and concurrent execution of actions in the parallel composition, is reduced to arbitrary interleavings plus nondeterminism. Especially for timed systems involving exchange of data values—aside from being unintuitive—this does not correctly capture the nature of distributed systems, since it imposes a sequential order on actions which conceptually happen at the same time. Furthermore, the sequential order might influence the availability of data items, depending on the execution order of transitions. What is more, from a technical point of view, the presence of all possible interleavings amplifies the state explosion problem. In contrast, TCA allow sets of actions on each transition, which permits *true concurrency*, as this directly models (truly atomic) synchronous communication through different ports.

Under true concurrency, all actions which happen at the same time (atomically) collapse into a single transition. As a consequence, a positive amount of time has to elapse before every visible transition (i.e., transitions involving visible data flow). This allows for a more "concise" semantics compared to TA: transitions in the associated LTS $\mathfrak{S}_{\mathfrak{A}}$ of a TA $\mathfrak{A}$ correspond to *either* the execution of a transition of $\mathfrak{A}$, *or* to a system delay,[7] cf. Definition 2.2.4. In contrast, *every* transition in the associated LTS $\mathfrak{S}_{\mathfrak{T}}$ of a TCA $\mathfrak{T}$ corresponds to the execution of a transition in $\mathfrak{T}$, possibly preceded by a time delay, cf. Definition 2.3.5. Thus, on average, runs of $\mathfrak{S}_{\mathfrak{T}}$ are shorter than runs of $\mathfrak{S}_{\mathfrak{A}}$ while containing the same number of visible events, i.e., providing the same "information content".

2.4 Timed Network Automata

In this section, we define the third system model: Timed Network Automata (TNA). TNA [Kem10] can be seen as an extension of TCA with *environmental constraints* (see for example [CCA07, Cos10]). Such constraints are imposed on the TNA by the surrounding network (hence the name), and capture information about whether the environment is ready to communicate. Thus, presence and absence of dataflow in the connector (which is modelled by the TNA) no longer depend on the internal state of the automaton only, but also take into account whether the environment is ready to communicate. Thereby, the behaviour of the environment is represented through constraints on the transitions of the TNA, i.e., there is *no need* to specify the environment explicitly. In this way, TNA provide a modular framework for compositional construction of a real-time generalisation of dataflow networks.

[7] Remember that subsequent delays can be combined into a single transition, cf. Remark 2.2.7)

The underlying idea of TNA is that absence of dataflow needs a reason. For example, a simple empty buffer (without any constraints on the input) should always be ready to accept data, communication can only be delayed if the reason comes from the outside the connector (if the environment does not provide a data item, i.e., is not ready to communicate). To capture where the reason for delaying the communication comes from, TNA distinguish between input ports and output ports.

2.4.1 Syntax of Timed Network Automata

The (externally visible) behaviour of a TNA is given by the possible dataflow through its (externally visible) ports, which not only depends on the TNA itself, but also on the environment it occurs in. In particular, there needs to be a reason for the absence of dataflow, from either the TNA or the environment; if both are ready to communicate, dataflow cannot be delayed. Consequently, we define the environmental constraints over the ports of a TNA. Essentially following the three-colouring idea presented in [CCA07], we define three different states of ports—called colours[8]—which not only capture presence and absence of dataflow, but in case of no dataflow also describe where the reason for delaying the communication comes from.

Definition 2.4.1 (Colours, Colourings). Let $\mathcal{P}$ be a finite set of ports, $\mathcal{Q}\subseteq\mathcal{P}$, and $p\in\mathcal{P}$. A *colouring* $\mathbb{c}\in\mathbb{C}(\mathcal{P})$ *over* $\mathcal{P}$ is a mapping $\mathbb{c}:\mathcal{P}\to\mathbb{C}lr$, assigning to each port $p\in\mathcal{P}$ a colour from the *set of colours* $\mathbb{C}lr=\{\text{—},\text{–!–},\text{–?–}\}$. We denote the colouring of a port p (i.e., the colour assigned to that port) by p:—, p:–!– and p:–?–, respectively. Port p is called *active (under colouring* $\mathbb{c}$*)* iff $\mathbb{c}(p)=$—, and *inactive (under colouring* $\mathbb{c}$*)* otherwise.

The *restriction* $\mathbb{c}|_{\mathcal{Q}}$ *of colouring* $\mathbb{c}$ *from* $\mathcal{P}$ *to* $\mathcal{Q}$ is a colouring that agrees with $\mathbb{c}$ on ports in $\mathcal{Q}$, and is undefined otherwise, i.e., $\mathbb{c}|_{\mathcal{Q}}:\mathcal{Q}\to\mathbb{C}lr$, $\mathbb{c}|_{\mathcal{Q}}(p)=\mathbb{c}(p)$, $p\in\mathcal{Q}$.

We may write colourings in either orientation (i.e., p:— or —:p), and we may omit the port name p if it is clear from the context. Further, we may write $\mathbb{C}$ if $\mathcal{P}$ is clear from the context.

The intended idea of the colourings of a port p is to denote *dataflow through* p (p:—) and *delay on* p, with the underlying TNA to which the port belongs either providing (p:–!–) or getting (p:–?–) a reason for the delay on its port p. Intuitively, p:–?– means that the TNA cannot actively delay dataflow through p, instead, delay requires a reason from the *outside*. On the other hand, p:–!– denotes that the TNA *itself* delays the communication, for example because no data is available to be transmitted.

A TNA consists of the externally visible ports, plus the internal behaviour. We specify this internal behaviour by means of finite automata, which are equipped with real-valued clocks. As for TA (cf. Section 2.2), we assume location changes to be instantaneous, time may only elapse while the automaton remains in one of

[8]We here adopt the term *colour* and related notions, as introduced in [CCA07], to avoid confusion with other uses of the word *state*.

its locations.[9] The delays may depend on environmental constraints. Therefore, we specify the admissible delays as explicit transitions of the automaton modelling the internal behaviour of the TNA.

Similar to TCA, TNA are equipped with a finite set of data variables $\mathcal{D}ata$. These can be used by locations, to store data values for use in subsequent steps. In addition, data constraints on transitions may reason about data variables which are not used by source or target location of the transition (this was not allowed for TCA, cf. Definition 2.3.1). This feature is primarily used for conciseness of the definition of TNA composition, but also to retain information when hiding ports from the environment (cf. Definitions 2.4.13 and 2.4.14).

Recalling Definitions 2.1.2, 2.1.4, 2.1.5, 2.1.7 and 2.1.8, we define TNA as follows.

Definition 2.4.2 (Timed Network Automaton). A *TNA* $\mathfrak{N}$ *over data domain* $\mathcal{D}ata$ is a tuple $\mathfrak{N}=(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, with S a finite set of locations, $s_0 \in S$ the initial location, $\mathcal{P}=\mathcal{P}^r \dot\cup \mathcal{P}^w$ a finite set of ports, with $\mathcal{P}^r$ and $\mathcal{P}^w$ disjoint sets of read respectively write ports, $\mathcal{X}$ a finite set of real-valued clocks, $I: S \to \mathcal{CC}(\mathcal{X})$ a function assigning a clock constraint (*location invariant*) to every location, $\mathcal{D}$ a finite set of data variables, $\#: S \to 2^{\mathcal{D}}$ a function assigning to each location the set of data variables it may use, and $E \subseteq (S \times \mathbb{C}(\mathcal{P}) \times \mathcal{DC}(\mathcal{P},\mathcal{D}) \times \mathcal{CC}(\mathcal{X}) \times \Lambda(\mathcal{X}) \times S)$ the finite transition relation. The set $\mathcal{P}$ of ports is also called the *external interface of* $\mathfrak{N}$.

An element $e=(s, \mathbb{c}, dc, cc, \lambda, s') \in E$ describes a transition from *source location* s to *target location* s', with dataflow/delay according to colouring $\mathbb{c}$, enabled under *data guard* dc and *clock guard* cc, and updating all clocks according to the update map λ. For every such transition, we require that dc only reasons about active ports, i.e., $dc \in \mathcal{DC}(\mathcal{Q}, \mathcal{D})$, with $\mathcal{Q}=\{p \in \mathcal{P} \mid \mathbb{c}(p)=\text{——}\}$. We require both dc and cc to be satisfiable. Transition e is called *delay* iff $s'=s$, $\lambda=id$ (identity mapping), and $\mathbb{c}(p) \neq \text{——}$ for all $p \in \mathcal{P}$, and *communication* otherwise. Two TNA are called *disjoint* if the respective constituents (i.e., locations, ports, clocks, data variables) are disjoint.

A communication $(s, \mathbb{c}, dc, cc, \lambda, s')$ describes the conditions for a location change from s to s', while a delay $(s, \mathbb{c}, dc, cc, id, s)$ describes the conditions under which $\mathfrak{N}$ may delay in location s—namely, as long as guard cc is satisfied, and a reason for delay exists which satisfies colouring $\mathbb{c}$. Note that the data constraint dc on delays may not involve ports (since no dataflow is allowed), but only data variables (the duration of the delay or whether the delay is possible at all may still depend on the contents of the memory cells).

Remark 2.4.3 (Use of Data Variables). As for TCA, we not impose any restrictions on the set of data variables used by locations, cf. Remark 2.3.2. This may lead to the same ambiguities as described in the Remark, if for a transition $(s, \mathbb{c}, dc, cc, \lambda, s')$, both locations use the same data variable d, and d occurs in dc. We use the same conventions as introduced for TCA (cf. Remark 2.3.2 again):

[9] Yet, it is straightforward to model duration of data flow in a TCA like style, for example by adding a fresh clock and appropriate clock guards >0 on transitions, which forces the automaton to delay in every location.

we prefix the occurrence of a data variable d in dc by "s." if it belongs to the source location, and by "t." if it belongs to the target location. That means, instead of $(d{=}\mathbb{d}_1){\wedge}(d{=}\mathbb{d}_2)$ (which would obviously evaluate to `false`), we write $(s.d{=}\mathbb{d}_1){\wedge}(t.d{=}\mathbb{d}_2)$, and any data assignment δ will consider $s.d$ and $t.d$ to be different elements. A data variable without prefix is used in the data constraint only and does not correspond to a memory cell of source or target location.

As for TCA before, unconstrained memory cells of the target location can take random values, and we may explicitly require data variables to be empty.

Example 2.4.4. An example for a TNA can be found in Figure 2.8. We model again the 1-bounded FIFO buffer with expiration from Example 2.3.3, but now in addition take into account environmental constraints. To model dataflow and environmental constraints, the TNA has a read port r, through which it receives the data item, a write port w, through which it releases the data item to the environment again, and uses a data variable m, to model the memory cell in location *full*. We denote communications by solid lines, and delays by dashed lines.

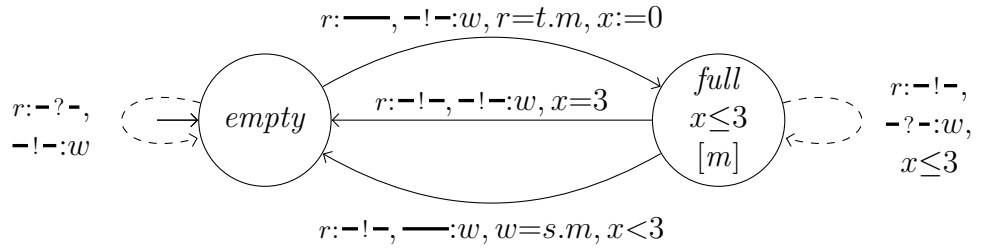


Figure 2.8: 1-bounded FIFO Buffer with Expiration and Environmental Constraints

The general idea of the TNA in Figure 2.8 is identical to the TCA presented in Figure 2.5. The three communications correspond almost directly to the three transitions in the TCA. On each communication, the inactive port always provides (and never requires) a reason to delay. On the upper communication (from *empty* to *full*), for example, this is due to the fact that the buffer is empty, so no data can flow out of it (through w), so the TNA itself provides a reason to delay on write port w. The reason can be read as "no data available". The explanation for the lower transition is similar. On the middle transition, both ports provide a reason to delay: port r cannot be active, since the buffer is full and no (more) data can be accepted through r. The fact that port w is inactive models the loss of the data item: once the deadline of 3 time units is reached, the data item is lost and cannot be sent through w anymore.

To handle environmental constraints, we add two delays, one for each location. The delay in location *empty* models the fact that no data item can be written to the environment (since there is no data, i.e., the buffer is empty), therefore, w provides a reason to delay. Since the TNA itself is ready to accept data through r, the read port requires a reason for delay. Stated differently: if the buffer is empty, it is always ready to accept data. The explanation for the delay in *full* is symmetrical, the additional clock constraint $x{\leq}3$ is used to enforce the expiration threshold of 3 time units.

Notation 2.4.5 (TNA). If not state otherwise, we shall assume the constituents of a TNA $\mathfrak{N}$ to be denoted as $\mathfrak{N}=(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, with port set $\mathcal{P}=\mathcal{P}^r \cup \mathcal{P}^w$, and of a TNA $\mathfrak{N}_i$ to be denoted as $\mathfrak{N}_i=(S_i, s_{0,i}, \mathcal{P}_i, \mathcal{X}_i, I_i, \mathcal{D}, \#_i, E_i)$, with port set $\mathcal{P}_i=\mathcal{P}_i^r \cup \mathcal{P}_i^w$, for $i \in \mathbb{N}$. We lift $\#$ to reason about sets of locations, and we may omit curly braces: $\#(s, s')=\#(s) \cup \#(s')$.

By $CC(\mathcal{X})|_{\mathfrak{N}}$, we denote the set of clock constraints (over clock set $\mathcal{X}$) that occur in a TNA $\mathfrak{N}$ (invariants or guards). Equivalently, by $DC(\mathcal{P},\mathcal{D})|_{\mathfrak{N}}$, we denote the set of data constraints over port set $\mathcal{P}$ and data variables $\mathcal{D}$ that occur in a TNA $\mathfrak{N}$.

In the graphical representation of TNA, we use (as before) assignment rather than functional notation for updates, and we omit guards equal to `true` as well as identity updates. We denote a set of memory cells $\{m_1, \ldots, m_n\}$ associated to a location as a bracketed list $[m_1, \ldots, m_n]$.

2.4.2 Semantics of Timed Network Automata

The semantics of a TNA is given by the set of runs of the associated LTS $\mathfrak{S}_{\mathfrak{N}}$.

Definition 2.4.6 (Associated LTS). Let $\mathfrak{N}$ be a TNA. The *associated LTS* $\mathfrak{S}_{\mathfrak{N}}$ is a tuple $\mathfrak{S}_{\mathfrak{N}}=(\mathcal{Q}, q_0, \rightarrow)$, with $\mathcal{Q} \subseteq (S \times DA(\mathcal{D}) \times \mathcal{V}(\mathcal{X}))$ the set of configurations, such that for every $(s, \delta, \nu) \in \mathcal{Q}$, $\delta \in DA(\mathcal{D})$, with $\delta(m)=\bot$ if $m \notin \#(s)$, and $\nu \models I(s)$. The initial configuration is $q_0=\langle s_0, \mathbf{0}, \mathbf{0} \rangle$, with $\mathbf{0}(x)=0$ for all $x \in \mathcal{X}$ and $\mathbf{0}(m)=\bot$ for all $m \in \#(s_0)$, and the transition relation $\rightarrow \subseteq (\mathcal{Q} \times \mathbb{C} \times (DA(\mathcal{P},\mathcal{D}) \cup \mathtt{Time}) \times \mathcal{Q})$ is given by

$$\frac{\begin{array}{l}(s, \mathbb{c}, dc, cc, \lambda, s') \in E,\\ \nu \models cc, \nu[\lambda] \models I(s')\\ \bar{\delta} \in DA(\mathcal{P},\mathcal{D}) : \bar{\delta} \models dc\\ \bar{\delta}(s.m)=\begin{cases}\delta(m), & \text{if } m \in \#(s)\\ \bot, & \text{otherwise}\end{cases}\\ \bar{\delta}(t.m)=\delta(m) \text{ if } m \in \#(s), m \in \#(s'), t.m \notin \mathcal{D}|_{dc}\\ \bar{\delta}(d)=\bot \text{ if } d \in \mathcal{D} \setminus (\mathcal{D}|_{dc} \cup \#(s, s'))\\ \bar{\delta}(p)=\bot \text{ iff } \mathbb{c}(p) \neq \text{—}\!\!\!\text{—}\\ \delta'(m)=\bar{\delta}(t.m) \text{ for all } m \in \#(s')\end{array}}{\langle s, \delta, \nu \rangle \xrightarrow{\mathbb{c}, \bar{\delta}} \langle s', \delta', \nu[\lambda] \rangle} \tag{2.15}$$

$$\frac{\begin{array}{l}(s, \mathbb{c}, dc, cc, id, s) \in E,\\ t>0, t \geq t' \geq 0 : \nu+t' \models cc, \nu+t' \models I(s),\\ \delta \models dc\end{array}}{\langle s, \delta, \nu \rangle \xrightarrow{\mathbb{c}, t} \langle s, \delta, \nu+t \rangle} \tag{2.16}$$

A *run of* $\mathfrak{S}_{\mathfrak{N}}$ *(starting in configuration* q_0*)* is a sequence of transitions $q_0 \xrightarrow{\gamma_0} q_1 \xrightarrow{\gamma_1} \ldots$, with $\gamma_i \in (\mathbb{C} \times DA(\mathcal{P},\mathcal{D})) \cup (\mathbb{C} \times \mathtt{Time})$. A run is called *initial* if it starts in the initial configuration q_0, it is called *loop-free* if all configurations are different.

Transitions of $\mathfrak{S}_{\mathfrak{N}}$ directly correspond to the two types of transitions of TNA, cf. Definition 2.4.2: an *action transition* (2.15) describes the firing of an instantaneous communication in $\mathfrak{N}$, with dataflow according to colouring $\mathbb{c}$. The clock guard cc is satisfied before the execution of the transition, and the invariant of the target location is satisfied after the execution, i.e., after updating the clocks (second row). The data assignment satisfies the data guard dc (third row). As for TCA (cf. (2.7)), the values of memory cells before the execution of the transition remain unchanged if used by the source location s, otherwise, they are empty (fourth row). The values of memory cells which are used in both source and target location carry over to s' if they are not modified in the data constraint, if a memory cell is not used in the target location, its value after the execution of the transition is unspecified (fifth row). Data variables which are not used in source or target location, nor in the data constraint, are empty (sixth row), and data may only be pending at active ports (sixth row). Data variables used by the target location s' obtain their values according to the data assignment on the transition. A *delayed action transition* (2.16) describes the firing of a delay of $\mathfrak{N}$: if a reason for delay exists that satisfies colouring $\mathbb{c}$, the TNA can delay for a positive amount of time t, during which the invariant $I(s)$ and the clock guard cc need to be satisfied at all points (second row),[10] and the data guard dc has to be satisfied. Since by definition, all ports are inactive (i.e., have a colouring $\neq$——) during the execution of a delay, the data values stored in data variables do not change (new data values can only be received through active ports), i.e., the data assignment δ is identical in both configurations, it can only reason about memory cells that are used by s.

Definition 2.4.7 (Semantics of Timed Network Automata). Let $\mathfrak{N}$ be a TNA, $\mathfrak{S}_{\mathfrak{N}}$ the associated LTS. The *trace semantics of* $\mathfrak{N}$ is given by the set $Run_{\mathfrak{N}}$ of initial runs (also called *executions*) of $\mathfrak{S}_{\mathfrak{N}}$. With $Run_{\mathfrak{N}k}$, we denote the set of finite prefixes of elements of $Run_{\mathfrak{N}}$ of (at most) length k.

Example 2.4.8 (Execution of a Timed Network Automaton). In (2.17), we show an execution of the TNA from Example 2.4.4 of length 8. Again (cf. Example 2.3.7), we omit data assignments on transitions which evaluate to $\perp$, and we abuse the single symbol $\perp$ to denote the empty data assignment, which assigns $\perp$ to all elements. In order not to clutter up the illustration, we further omit port names; the colourings correspond to port r on top, and port w below.

2.4.3 Systems of Timed Network Automata

In this section, we present a compositional product construction for TNA, which allows to build complex TNA out or simpler ones. The basic idea of the composition operator is to join sets of (read and write) ports, which conceptually yields invisible *internal ports*. Note that internal ports are theoretical constructs, in that they do not actually appear in the composed TNA. The notion of internal ports is used for explanatory purposes, and to reason about the validity of the composition.

[10] Due to convexity, it is actually enough to check the clock constraints at the beginning and at the end of the time delay only. We will use this fact in the representation, cf. Section 3.1.4.

$$\langle \mathit{empty}, \bot, x{=}0\rangle \xrightarrow{\mathbf{-?-},4} \xrightarrow{\mathbf{-!-}} \langle \mathit{empty}, \bot, x{=}4\rangle \xrightarrow{\overline{\mathbf{-!-}}, r=2,\ t.m=2} \langle \mathit{full}, m{=}2, x{=}0\rangle \xrightarrow{\mathbf{-!-},2.5} \xrightarrow{\mathbf{-?-}}$$

$$\langle \mathit{full}, m{=}2, x{=}2.5\rangle \xrightarrow{\mathbf{-!-}} \xrightarrow{\overline{\mathbf{-!-}}, w=2,\ s.m=2} \langle \mathit{empty}, \bot, x{=}2.5\rangle \xrightarrow{\mathbf{-?-},10} \xrightarrow{\mathbf{-!-}}$$

$$\langle \mathit{empty}, \bot, x{=}10\rangle \xrightarrow{\overline{\mathbf{-!-}}, r=5,\ t.m=5} \langle \mathit{full}, m{=}5, x{=}0\rangle \xrightarrow{\mathbf{-!-},3} \xrightarrow{\mathbf{-?-}}$$

$$\langle \mathit{full}, m{=}5, x{=}3\rangle \xrightarrow{\mathbf{-!-}} \xrightarrow{\mathbf{-!-}, w=\bot,\ s.m=5} \langle \mathit{empty}, \bot, x{=}3\rangle \tag{2.17}$$

The intended behaviour of internal ports is to act as self-contained, stateless "pumping stations" [BSAR06], *merging* data from write ports, and *replicating* data to read ports. If data flows through an internal port, then it flows through *exactly one* underlying write port and through *all* underlying read ports. Absence of dataflow is subject to environmental constraints on the involved ports: if there is a reason for delay (**-!-**) on at least one read port (i.e., the TNA contributing the port provides a reason to delay on that port) or on all write ports, data cannot flow. Stated differently, a valid colouring of an internal port must not involve the colour **-?-** only. We do not restrict composition to one-to-one relations (as is done in [Arb04, CCA07, CPLA09], for example). On the contrary, we do not impose any restrictions on the number, type (read/write) or origin (same or different TNA) of ports to be merged; the only condition is that a port cannot be merged more than once. Though the composition of colourings would be slightly simpler in a one-to-one approach, our many-to-many composition provides a direct and more intuitive way of specifying compositions, for example for mergers, replicators or multi-synchronisations.

We now formalise these ideas.

Definition 2.4.9 (Merge Set, Validity of Colouring). Let $\mathcal{P}$ be a set of ports, $\mathcal{Q}{\subseteq}\mathcal{P}$ a subset, $\mathcal{Q}^r{\subseteq}\mathcal{P}^r$ and $\mathcal{Q}^w{\subseteq}\mathcal{P}^w$ the sets of read respectively write ports in $\mathcal{Q}$, and $\mathbb{c}{\in}\mathbb{C}(\mathcal{P})$ a colouring. If ports in $\mathcal{Q}$ are intended to be joined (merged), we call $\mathcal{Q}$ a *merge set (over $\mathcal{P}$)*, the resulting internal port is denoted as $p_{\prec\mathcal{Q}}$.

Colouring $\mathbb{c}$ is *valid over merge set* $\mathcal{Q}$ (or *valid over* $p_{\prec\mathcal{Q}}$), if it satisfies the following conditions for all ports $w, w', r{\in}\mathcal{Q}$:

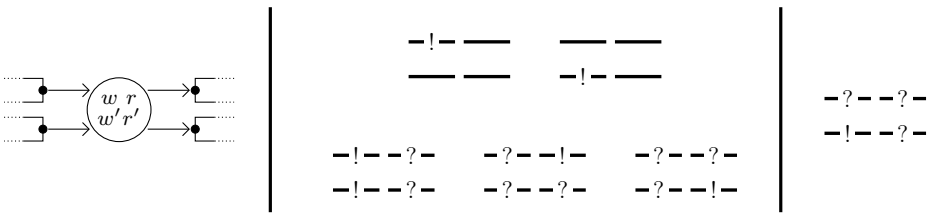
1. If $\exists w{\in}\mathcal{Q}^w{:}\mathbb{c}(w){=}\overline{\mathbf{-!-}}$, then $\forall r{\in}\mathcal{Q}^r{:}\mathbb{c}(r){=}\overline{\mathbf{-!-}}$, and $\forall w'{\in}\mathcal{Q}^w, w'{\neq}w{:}\mathbb{c}(w'){\neq}\overline{\mathbf{-!-}}$
2. If $\exists r{\in}\mathcal{Q}^r{:}\mathbb{c}(r){=}\overline{\mathbf{-!-}}$, then $\exists w{\in}\mathcal{Q}^w{:}\mathbb{c}(w){=}\overline{\mathbf{-!-}}$
3. If $\nexists w{\in}\mathcal{Q}{:}\mathbb{c}(w){=}\overline{\mathbf{-!-}}$, then $((\forall w'{\in}\mathcal{Q}^w{:}\mathbb{c}(w'){=}\mathbf{-!-})$ or $(\exists r{\in}\mathcal{Q}^r{:}\mathbb{c}(r){=}\mathbf{-!-}))$

Colouring $\mathbb{c}$ is valid over a set $\mathcal{Q}'$ of disjoint merge sets $\mathcal{Q}'{=}\{\mathcal{Q}_1, \ldots, \mathcal{Q}_n\}$, $n{\geq}1$, if it is valid over each $\mathcal{Q}_i$.

Only valid colourings correctly reflect/model the aforementioned behaviour of internal ports: conditions 1 and 2 in Definition 2.4.9 describe simultaneous dataflow

through exactly one write port and all read ports of $p_{\prec\mathcal{Q}}$. Condition 3 describes the propagation of environmental constraints (delays): no dataflow is possible only if either all write ports or at least one read port in $\mathcal{Q}$ provide a reason to delay.

Example 2.4.10 (Validity of Colourings). To illustrate validity of colourings over internal ports, consider a merge set which contains two write ports w, w', and two read ports r, r'. The internal port resulting from this merge set is conceptually depicted on the left side of the illustration below (note that the ports do not need to come from different TNA, as suggested in the picture). Some of the valid colourings of the internal port are given in the middle (the layout of colourings reflects the layout of the ports on the left), there are 17 valid colourings in total.



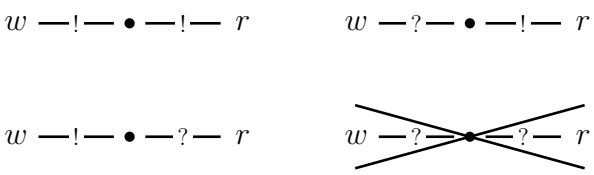
The colouring in the lower right, for example, can be read as follows: if read port r' provides a reason to delay (−!−), while the other ports can only delay if they get a reason (−?−), then the reason from r' is enough to delay dataflow through the internal port.

The colouring on the right side of the illustration is an example for an invalid colouring. It corresponds to a situation where there is no actual reason for delay: neither of the read ports can delay the communication, both read ports require a reason for delay. Write port w' provides a reason for delay, but write port w requires a reason, that means, w is actually ready to communicate. Thus, data could flow through ports w, r and r', since they are all ready to communicate, so this "no flow" colouring is not valid.

The *flip rule*, introduced in [CCA07], is used to reduce the size of composed TNA, by identifying redundant (with respect to compositionality) colourings.

Remark 2.4.11 (Flip Rule). Let $\mathcal{P}$ be a set of ports, $p{\in}\mathcal{P}$, and $\mathbb{c}_1, \mathbb{c}_2{\in}\mathbb{C}(\mathcal{P})$. If $\mathbb{c}_1$ and $\mathbb{c}_2$ are identical except for $\mathbb{c}_1(p){=}$−!− and $\mathbb{c}_2(p){=}$−?−, then $\mathbb{c}_2$ is redundant and can be removed: the set of colourings with which $\mathbb{c}_2$ can compose over p is a strict subset of the set of colourings with which $\mathbb{c}_1$ can compose over p.

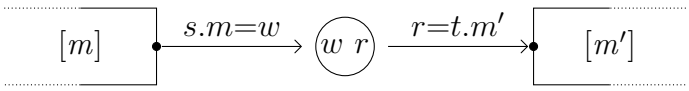
Example 2.4.12 (Flip Rule). To illustrate the flip rule, consider the following simple example.



A write port w with colouring w:−!− can compose with a read port r under both possible "no flow" colourings of r (left side). But the colouring w:−?− can compose with the colouring −!−:r only (right side), the colouring in the lower right is not valid, since it corresponds to a situation where both ports delay, without actually having a reason to delay (cf. also Example 2.4.10). Therefore, the colouring w:−?− is redundant and can be removed.

As explained above, internal ports are theoretical constructs, and do not actually appear in the composed TNA. In particular, the ports in the merge set are removed from the composed TNA. Consequently, we would have to remove the data constraints on these ports as well, since data constraints may only reason about (active) ports. If we want to ensure that data values are transmitted correctly over internal ports, we need to preserve the information from data constraints on ports in the merge set; simply removing such data constraints would not correctly reflect the intended behaviour.

To illustrate this, consider the following example: a TNA $\mathfrak{N}_1$ has a data value stored in a memory cell m, then writes it to the environment through a write port w. The corresponding transition contains a data constraint of the form $s.m{=}w$, to ensure the value that is written through w is indeed that value that was contained in m. A second TNA $\mathfrak{N}_2$ reads a data value from the environment into a memory cell m', the corresponding transition contains a data constraint of the form $r{=}t.m'$. Ports w are r are to be merged, i.e., there exists a merge set $\mathcal{Q}{=}\{w, r\}$. This scenario is conceptually depicted as



The expected result of merging ports w and r is to create a permanent link, with the expected behaviour that the data value that arrives in m' is always identical to the one that was contained in m. Yet, if we simply remove the two data constraints, we cannot guarantee this behaviour, since the correlation between m and m' is completely lost. So instead, we would like to obtain a data constraint (that is equivalent to) $s.m{=}t.m'$. For this, we define reduced (with respect to a merge set) data constraints.

Definition 2.4.13 (Reduced Data Constraint). Let $\mathcal{P}$ be a set of ports, $\mathcal{Q}{\subseteq}\mathcal{P}$ a merge set over $\mathcal{P}$, $\mathcal{Q}'{=}\{\mathcal{Q}_1, \ldots, \mathcal{Q}_n\}$, $n{\geq}1$, a set of disjoint merge sets over $\mathcal{P}$, $dc{\in}\mathcal{DC}(\mathcal{P})$ a data constraint, $d, d_1, \ldots, d_n{\in}\mathcal{D}$ distinct data variables not occurring in dc, i.e., $d, d_1, \ldots, d_n{\notin}\mathcal{D}|_{dc}$. The *reduced data constraint* $dc|_{\mathcal{Q}}$ *of* dc *(with respect to* $\mathcal{Q}$*)* is obtained by replacing every occurrence of a port $q{\in}\mathcal{Q}$ in dc by data variable d. The *reduced data constraint* $dc|_{\mathcal{Q}'}$ *of* dc *(with respect to* $\mathcal{Q}'$*)* is obtained by replacing every occurrence of port $q_i{\in}\mathcal{Q}_i$ by data variable d_i.

The reduced data constraint $dc|_{\mathcal{Q}}$ removes the occurrence of all ports in the merge set $\mathcal{Q}$, while preserving the information on admissible data values. By replacing *all* occurrences of ports $q{\in}\mathcal{Q}$ by the same data variable d, we ensure that the transmitted

data value is the same on all ports contained in the merge set. Note that the constraints about whether dataflow is possible at all are already covered by valid colourings (see Definition 2.4.9).

We now have all the necessary concepts for defining the composition of TNA. The basic idea of TNA composition is along the same lines as the standard cross product in other automata models: the sets of read and write ports are joined, the ports in the merge sets are removed from the TNA, and the colourings of the involved transitions are composed. The composition of colourings (over disjoint port sets) is defined by standard function composition: for two colourings $\mathbb{c}_1 \in \mathcal{P}_1$ and $\mathbb{c}_2 \in \mathcal{P}_2$, with $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, the *composition* $\mathbb{c}_1 \cup \mathbb{c}_2$ is a new colouring $\mathbb{c} = \mathbb{c}_1 \cup \mathbb{c}_2 \in \mathbb{C}(\mathcal{P}_1 \cup \mathcal{P}_2)$, with $\mathbb{c}(p) = \mathbb{c}_1(p)$ iff $p \in \mathcal{P}_1$, and $\mathbb{c}(p) = \mathbb{c}_2(p)$ iff $p \in \mathcal{P}_2$, for all ports $p \in \mathcal{P}_1 \cup \mathcal{P}_2$.

Definition 2.4.14 (TNA Composition). Let $\mathcal{N} = \{\mathfrak{N}_1, \ldots, \mathfrak{N}_k\}$, $k \geq 1$, be a set of disjoint TNA, $\mathcal{Q} = \{\mathcal{Q}_1, \ldots, \mathcal{Q}_n\}$, $n \geq 1$, be a set of disjoint merge sets over $\bigcup \mathcal{P}_i$, $i = 1, \ldots, k$. The *composition of the* $\mathfrak{N}_i$ *over* $\mathcal{Q}$, denoted $\mathfrak{N}_1 \bowtie_{\mathcal{Q}} \ldots \bowtie_{\mathcal{Q}} \mathfrak{N}_k$ (or simply $\mathcal{N} \bowtie_{\mathcal{Q}}$), is a new TNA $\mathcal{N} \bowtie_{\mathcal{Q}} = (S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, with $S = \prod S_i$ (Cartesian product), $s_0 = (s_{0,1}, \ldots, s_{0,k})$, $\mathcal{P} = \bigcup \mathcal{P}_i \setminus \bigcup \mathcal{Q}_i$, $\mathcal{X} = \bigcup \mathcal{X}_i$, $I((s_1, \ldots, s_k)) = \bigwedge I_i(s_i)$, $\mathcal{D} = \bigcup \mathcal{D}_i$, $\#: S \to 2^{\mathcal{D}}$, with $\#((s_1, \ldots, s_k)) = \bigcup \#(s_i)$, and E is defined in (2.18).

$$\frac{\begin{array}{c}(s_1, \mathbb{c}_1, dc_1, cc_1, \lambda_1, s_1') \in E_1, \ldots, (s_k, \mathbb{c}_k, dc_k, cc_k, \lambda_k, s_k') \in E_k, \\ \mathbb{c} = (\mathbb{c}_1 \cup \ldots \cup \mathbb{c}_k)|_{\mathcal{P}} \text{ valid over } \mathcal{Q} \\ dc = (dc_1 \wedge \ldots \wedge dc_k)|_{\mathcal{Q}} \\ cc = cc_1 \wedge \ldots \wedge cc_k, \lambda = \lambda_1 \circ \ldots \circ \lambda_k\end{array}}{((s_1, \ldots, s_k), \mathbb{c}, dc, cc, \lambda, (s_1', \ldots, s_k')) \in E} \tag{2.18}$$

We call the $\mathfrak{N}_i$ the *underlying TNA* of $\mathcal{N} \bowtie_{\mathcal{Q}}$.

A transition in the composed TNA results from composing k transitions (called *underlying transitions*) from the underlying TNA. If all underlying transitions are delays, the resulting transition in $\mathcal{N} \bowtie_{\mathcal{Q}}$ is a delay as well. If at least one of the underlying transitions is a communication, the transition in $\mathcal{N} \bowtie_{\mathcal{Q}}$ is a communication as well. In the latter case, the associated TNA of underlying delays (i.e., TNA which contribute a delay to the composed transition) perform zero-delay steps. As for TCA (cf. Definition 2.3.9 and explanations thereafter), we have that the semantics of TNA (in particular the fourth and fifth row of (2.15)) ensures that data values stored in data variables used by locations correctly carry over to the next location.

Example 2.4.15 (TNA Composition). Consider two instances of the TNA from Example 2.4.4. The second instance is identical to the TNA in Figure 2.8, except that we add a prime to all names (locations, ports, data variables, clocks). We compose the two TNA over the merge set $\mathcal{Q} = \{w, r'\}$, the result is shown in Figure 2.9. As has been done in Figure 2.7, we abbreviate location names for readability. The resulting TNA models an expiring FIFO buffer with capacity 2, where each buffer cell has its own expiration timer.

The explanation of the behaviour of the TNA is essentially equivalent to the explanation in Example 2.3.10. A significant difference is the fact that it is not possible

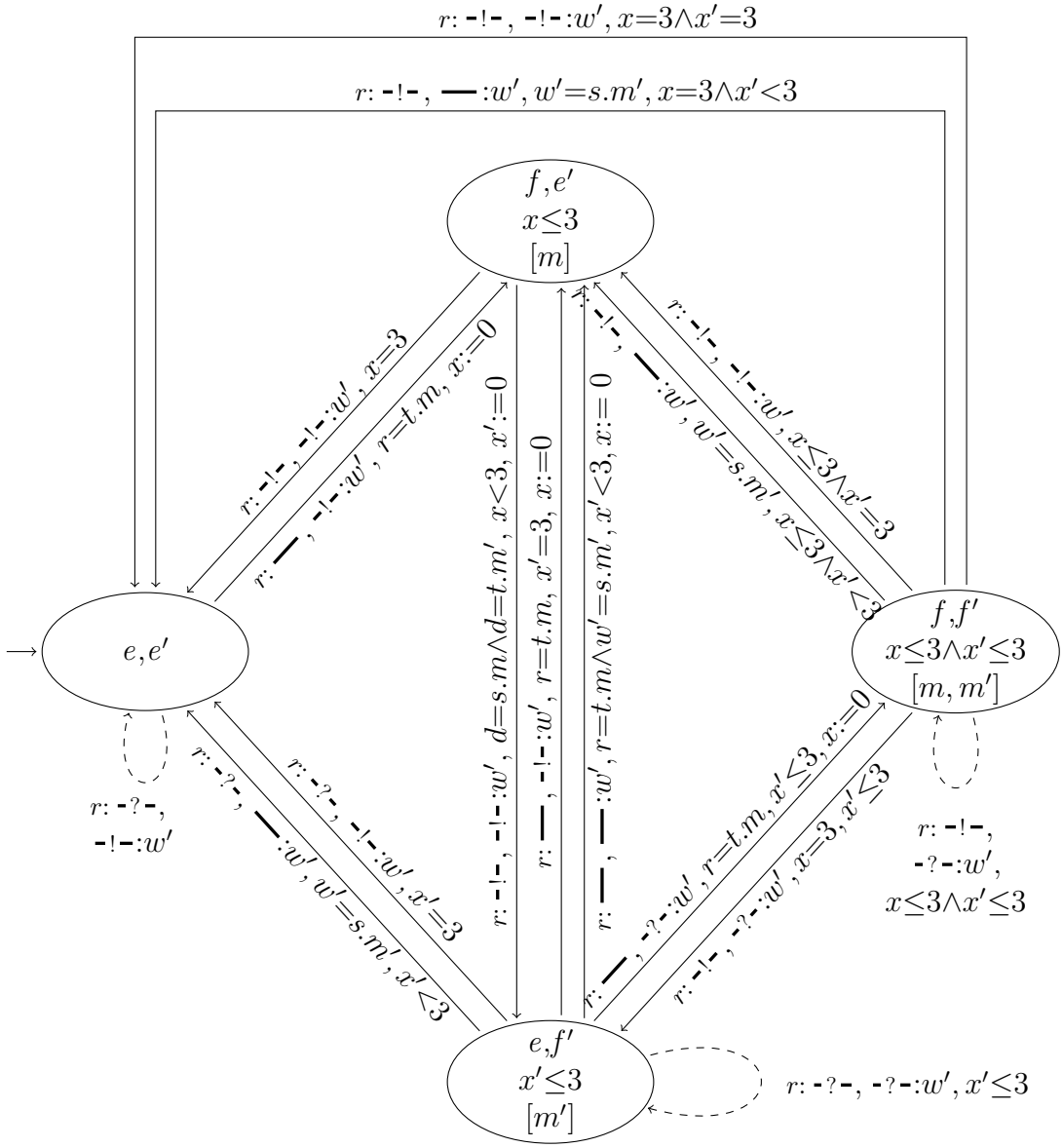


Figure 2.9: TNA Composition: Example

to delay in location (f,e') (while this was possible in the TCA in Example 2.3.10). After entering this location, the only possible transitions are the communications to locations (e,e') and (e,f'). The first transition is taken if $x{=}3$, i.e., the timer of the first buffer cell has expired. The second transition corresponds to an "internal" transition, where the data item is transmitted from the first to the second buffer cell. This transition shows that the composed TNA is indeed a FIFO buffer: data items may not remain in the first buffer cell if the second buffer cell is empty, but are pushed towards the "end" as far as possible.

Also note that on the transition from (e,f') to (f,f'), the value of m' is unconstrained. Since the underlying transition in the second (primed) TNA is a delay, the semantics ensures that the value carries over to (f,f') unchanged.

Proposition 2.4.16 (TNA Composition). The composition of TNA over *disjoint* merge sets is commutative and—after applying the flip rule to remove redundant colourings—associative, up to isomorphy of location names.

Proof.

1. Commutativity follows from the commutativity of $\cup$ on port sets, clock sets and data variable sets, the commutativity of $\wedge$ on clock and data constraints, the commutativity of $\cup$ on colourings over disjoint ports sets, and the commutativity of $\circ$ on update maps over disjoint clock sets.

2. Associativity follows from the associativity of $\cup$ on port sets, clock sets and data variable sets, the associativity of $\wedge$ on clock and data constraints, the associativity of $\cup$ on colourings over disjoint port sets, the associativity of $\circ$ on update maps over disjoint clock sets, and the fact that for disjoint $\mathcal{P}_i$, $i{=}1,2,3,$, and disjoint $\mathcal{Q}_j$, $j{=}1,2,3$, with $\mathcal{Q}_j{\subseteq}\bigcup\mathcal{P}_i$, we have

$$(((\mathcal{P}_1{\cup}\mathcal{P}_2)\setminus(\mathcal{Q}_1{\cup}\mathcal{Q}_2)){\cup}\mathcal{P}_3){\setminus}\mathcal{Q}_3{=}(((\mathcal{P}_2{\cup}\mathcal{P}_3)\setminus(\mathcal{Q}_2{\cup}\mathcal{Q}_3)){\cup}\mathcal{P}_1){\setminus}\mathcal{Q}_1$$

□

The semantics of a composed TNA $\mathcal{N}{\bowtie_{\mathcal{Q}}}$ (Definition 2.4.14) is defined in the same way as for simple TNA (Definition 2.4.7), i.e., by executions of the associated LTS (Definition 2.4.6).

In (2.19), we show a run of the composed TNA from Figure 2.9 of length 7. Again, we omit data assignments on transitions which evaluate to $\perp$, and we abuse the single symbol $\perp$ to denote the empty data assignment. Further, we omit port names, the colourings correspond to port r on top and port w' below.

$$\begin{aligned}
&\langle (e,e'), \perp, \tfrac{x=0}{x'=0}\rangle \xrightarrow{-?-,4\ \ -!-} \langle (e,e'), \perp, \tfrac{x=4}{x'=4}\rangle \xrightarrow{\text{——},r=3\ \ -!-,t.m=3} \langle (f,e'), m{=}3, \tfrac{x=0}{x'=4}\rangle \\
&\xrightarrow{-!-,t.m=3\ \ -!-,s.m'=3\ \ -!-\ d=3} \langle (e,f'), m'{=}3, \tfrac{x=0}{x'=0}\rangle \xrightarrow{-?-,0.5\ \ -?-} \langle (e,f'), m'{=}3, \tfrac{x=0.5}{x'=0.5}\rangle \\
&\xrightarrow{\text{——},r=5\ \ -?-,t.m=5\ \ -?-\ t.m'=3} \langle (f,f'), \tfrac{m=5}{m'=3}, \tfrac{x=0}{x'=0.5}\rangle \xrightarrow{-!-,2.5\ \ -?-} \langle (f,f'), \tfrac{m=5}{m'=3}, \tfrac{x=2.5}{x'=3}\rangle \\
&\xrightarrow{-!-,s.m=5\ \ -!-,t.m=5} \langle (f,e'), m{=}5, \tfrac{x=2.5}{x'=3}\rangle \xrightarrow{-!-,t.m=5\ \ -!-,s.m'=5\ \ -!-\ d=5} \langle (e,f), m'{=}5, \tfrac{x=2.5}{x'=0}\rangle \\
&\xrightarrow{-?-,3\ \ -?-} \langle (e,f), m'{=}5, \tfrac{x=5.5}{x'=3}\rangle \xrightarrow{-?-,\perp\ \ -!-} \langle (e,e'), \perp, \tfrac{x=5.5}{x'=3}\rangle
\end{aligned} \tag{2.19}$$

The run describes almost the same behaviour as the TCA run presented in (2.14). In particular, the time values on transitions, that means the lengths of delays, are identical wherever possible. Yet, as mentioned in Example 2.4.15, the significant difference is that it is not possible to delay in location (f,e'): whenever the TNA enters that location, it (in this example) immediately transfers the data item from the first to the second buffer cell, thereby moving to location (e,f').

Remark 2.4.17 (Size of the Composition). The size of a TNA, i.e., the number of locations and transitions, can be exponential (in the number of locations and transitions of the underlying TNA) in the worst case. For model checking/verification, we present a linear technique to avoid the explicit construction of the composed TNA in Section 3.1.4.

2.4.4 Discussion

In this Section, we have described a powerful framework for *compositional construction of* and *coordination in* real-time dataflow networks, which takes into account environmental constraints from outside the network. The approach is suitable to model both the "algorithmic" behaviour of components and connectors (for example the internal implementation of the coordination pattern), and the inter-component coordination behaviour. In this way, whole networks can easily be described with our formalism.

The development of TNA in [Kem10] was—amongst others—motivated by the fact that in TCA, it is not possible to enforce a transition to be taken as soon as dataflow through some port is possible. For example, consider again the FIFO buffer in Figure 2.5: the TCA is not required to take the transition from location *empty* to *full* as soon as data is available through p, instead, it can delay for an arbitrary amount of time. In contrast, in the corresponding TNA in Figure 2.8, when data is available through r, the TNA can no longer delay in location *empty*. Yet, we could still permit to delay for an arbitrary amount of time in Figure 2.8, by changing the colouring $r{:}{-}?{-}$ to $r{:}{-}!{-}$ on the delay in location *empty*.

In this thesis, we have further extended the basic definition of TNA from [Kem10] in two ways, by introducing data guards and data variables (together with a domain $\mathcal{D}ata$ of admissible data values). Data guards on transitions allow to model coordination patterns which depend on concrete data values, rather than only on presence and absence of dataflow (which was the case in [Kem10]). The benefits of introducing data variables are twofold: as for TCA, using data variables in locations ("location memory") allows the coordination pattern to reason about data values which were exchanged *prior* to the current step. Secondly, in the composition of TNA, using data variables and data guards allows us to preserve the information about the data values which are exchanged through internal ports (cf. Definitions 2.4.13 and 2.4.14, and explanations before that).

With TNA, we have defined a powerful yet simple framework for defining and describing the behaviour of connectors and whole networks. Our liberal notion allows to encode many common (coordination) models, like for example TA and TCA, in our framework.[11] On the other hand, the state-based approach makes our framework easy to understand (and thus, use), and facilitates the introduction of new, user-defined coordination patterns.

[11] The basic idea for TCA is to restrict the use of data variables to locations, and to use the no-flow colour $-!-$ only. For TA, the basic idea is to not allow data variables or data guards at all, and permit only one port per transition. We skip the technical details.

2.5 Conclusion

In this Chapter, we have presented general concepts for handling time and data in real-time systems, and we have presented three formal models for specification of real-time systems and real-time coordination patterns.

The formal model of Timed Automata, presented in Section 2.2, has first been introduced in [AD94] (and later in [Alu99]), and—as has been outlined in Section 2.2.4—has been studied, modified and extended intensively since that time. The Definitions and results in Section 2.2 have not been developed by us, but are entirely based on previous work on this field, which we have made clear by providing pointers to corresponding fundamental literature all throughout the Section. The nonetheless in-depth character of Section 2.2 is owed to the facts that there exist so many variants of TA (with sometimes only minor differences) that it is impossible to use the notion of *the* Timed Automaton without further explanations, and that the time-related notions (guards, invariants, update maps) directly carry over to TCA and TNA. Section 2.2 thus serves the purpose to make clear which notions and concepts of TA we use in this thesis, and to establish the concepts for handling of real-time. Moreover, it serves as the formal basis for the formula representation and SAT-based verification of TA introduced in the next Chapter.

We have presented the second formal model, Timed Constraint Automata, in Section 2.3. The first formal definition of syntax and semantics of TCA can be found in [ABdBR04] (and its extended version [ABdBR07]), though the authors do not handle memory cells. In [PSHA09], memory cells are added to constraint automata, resulting in the formal model of CASM (Constraint Automata with Memory Cells), but as the name suggests, CASM do not include time. While the underlying idea of handling data and memory cells in CASM is similar to the approach presented here, the semantics of CASM is not defined formally, which leaves a number of open questions. For example, it is unclear whether a memory cell used by the target location of a CASM transition has to be initialised in the data guard of ingoing transitions, or what happens in case it is not initialised (is the value undefined, empty, random). From [PSHA09], we have adopted the notion of prefixing memory cells with "s." and "t." on transitions (cf. Remark 2.3.2), but to the best of our knowledge, this is the first work on combining TCA with memory cells, and defining a formal semantics.

Finally, in Section 2.4, we have presented our third and most powerful (with respect to expressiveness of the three models) formal model: Timed Network Automata. As explained in Section 2.4.4, we have originally developed TNA as an enhancement of TCA, to be able to specify constraints under which it is admissible to delay (further). In this work, we have improved the TNA model from [Kem10], by adding data guards and data values, with the advantages discussed in Section 2.4.4.