Verification of Networks of Timed Automata using mCRL2

Jan Friso Groote Michel A. Reniers Yaroslav S. Usenko

Laboratory for Quality Software, Department of Mathematics and Computer Science,
Technical University of Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

1 Introduction

It has been our long time wish to combine the best parts of the real-time verification methods based on timed automata (TA) (the use of regions and zones), and of the process-algebraic approach of languages like LOTOS and timed μCRL. This could provide us with additional verification possibilities for real-time systems, not available in existing timed-automata-based tools like UPPAAL [8].

In the previous work [7] we transformed a timed μCRL process equation representing a timed automaton into a closely related timed μCRL process equation with finite discrete parameters and bound variables only. This could enable simulation and verification via enumeration of reachable states. As a result, existing untimed analysis tools become applicable to the analysis of real-time systems.

In this paper we extend the applicability of such discretization to extensions of TA available in UPPAAL [8] as *networks of timed automata* and *shared variables*. To this end, we make use of mCRL2, the newer version of μCRL that includes time and *multi-actions*. The multi-actions are used to model the simultaneous access to shared variables and action synchronization[1].

Timed Automata A timed automaton [2] consists of locations and transitions between locations. The amount of time that can be spent in a certain location is described by means of invariants on a number of clock variables.

In UPPAAL, networks of timed automata are used to specify real-time systems. In such specifications a number of timed automata execute in an interleaved manner. Synchronous communication between the timed automata is performed by hand-shake synchronization using input and output actions. Output and input actions are denoted with an exclamation mark and a question mark respectively, e.g., $a!$ and $a?$. Asynchronous communication is achieved by means of shared variables. We do not consider urgent channels and committed locations in the present paper.

mCRL2 The language mCRL2 [5] offers a uniform framework for the specification of data and processes. Data are specified by equational specifications: one can declare sorts and functions working upon these sorts, and describe the meaning of these functions by equational axioms. Processes are described in process algebraic style, where the particular process syntax stems from ACP [4], extended with data-parametric ingredients: there are constructs for conditional composition, and for data-parametric choice and communication. As is common in process algebra, infinite processes are specified by means of (finite systems of) recursive equations. In mCRL2 such equations can also be data-parametric.

Several timed extensions have been proposed for different kinds of process algebras. For an overview of ACP extensions with time we refer to [3]. According to [3], timed process algebras can be categorized by three criteria: *discrete* vs. *continuous* time; *relative* vs. *absolute* time; *two-phase* vs. *timed-stamped* model. mCRL2 makes use of absolute time, a timed stamped-model, and the time domain is defined by the user (discrete and continuous domains are possible).

Translating Networks of TA to mCRL2 In [7], we use the existing results of [14] to translate a timed automaton to a timed μCRL processes in the form of a *Timed Linear Process Specification* (TLPS). This translation uses a very simple sort **T** to represent the real-time clock values. As a result we obtain a semantically equivalent specification in timed μCRL. The resulting timed μCRL process only uses a very restricted subset of the full syntax of the language: it has the form of a TLPS.

In the present paper, we extend the approach for translating one timed automaton to timed μCRL from [7] to translating a network of communicating timed automata with shared variables to a process in mCRL2[2].

Linearization In [11], we outlined a method to describe and analyze real-time systems using timed μCRL. Most descriptions of such systems contain operators such as parallel composition that complicate analysis. As a first step towards the analysis of such systems, we *linearize* the given description using the algorithm from [13]. The result is a TLPS which is equivalent to the original description and has a very simple structure (see Section 4).

Discretization In [7], we present an approach to replace parameters of sort **T** in TLPS's of a certain form by parameters of discrete sorts. We introduce process-algebraic transformations and abstractions for this purpose. The result of the application of those transformations to a TLPS that re-

[1] It is possible to achieve the same effect without multiactions, but in a much more complex way.

[2] mCRL2 is the successor of μCRL ([6]) and timed μCRL ([10]). With respect to the use of timed μCRL in [7] there are no relevant differences.

sults from the step described before, is a process specification with only parameters of discrete sorts that is closely related to the original TLPS in the following sense. After abstraction from the fractional parts of the time stamps in the actions, the two specifications are timed bisimilar.

Example In this paper, we apply the steps described before to the example of Fischer's protocol. After translating the timed automaton describing Fischer's protocol to an mCRL2 process, this process is linearized, i.e., transformed into an equivalent timed linear process specification. The resulting TLPS is in the input format of the discretization and time abstraction transformations as presented in [7]. Application of those results in a finite representation of the original system which can be used for establishing correctness more easily.

2 mCRL2

In this section, we present those parts of mCRL2 that are needed for a good understanding of the material that follows in subsequent sections. For a more complete treatment we refer to [5].

The mCRL2 data language is a functional language based on *higher-order abstract data types*. As mentioned before, mCRL2 also has concrete data types: *standard data types* and sorts constructed from a number of *type constructors*.

Data types Basically, mCRL2 contains a simple and straightforward data type definition mechanism. Sorts (types), constructor functions, maps (functions) and their definitions can be declared. Sorts declared in such a way are called user-defined sorts. mCRL2 has also the following built-in data types:

- Booleans (*Bool*) with constructor functions **t** (representing *true*) and **f** (for *false*) and operators $\neg$, $\wedge$, $\vee$, and $\rightarrow$. It is assumed that **t** and **f** are different.
- Unbounded positive numbers ($\mathbf{N}^+$), natural numbers ($\mathbf{N}$), integers ($\mathbf{Z}$), and real numbers ($\mathbf{R}$) with relational operators $<$, $\leq$, $>$, $\geq$, unary negation $-$, binary arithmetic operators $+$, $-$, $*$, *div*, *mod* and arithmetic operations *max*, *min*, *abs*, *succ*, *pred*, *exp*. These functions are only available for appropriate sorts, e.g. *div* and *mod* are only defined for a denominator of sort $\mathbf{N}^+$. Also conversion functions *A2B* are provided for all sorts $A, B \in \{\mathbf{N}^+, \mathbf{N}, \mathbf{Z}, \mathbf{R}\}$.
- Real numbers are used to represent time values in mCRL2. In this paper we refer to the time domain as sort $\mathbf{T}$ and make use of constant $\mathbf{0}$ representing time zero. We assume that the values of sort $\mathbf{T}$ are non-negative real numbers (to avoid a lot of conditions of the form $t \geq \mathbf{0}$), and make use of functions of real numbers to construct terms of sort $\mathbf{T}$.

There are a number of type constructors for structured types, function types, and for lists, sets and bags. In this paper we use the *list* type constructor. The sort of (finite) lists containing elements of sort A is declared by $\mathbf{L}(A)$ and has constructor functions $[\,] : \mathbf{L}(A)$ and $\triangleright : A \times \mathbf{L}(A) \rightarrow \mathbf{L}(A)$. Other operators include $\triangleleft$, $+\!\!+$ (concatenation), . (element at), *head*, *tail*, *rhead* and *rtail* together with list enumeration $[\,e_0, \ldots, e_n\,]$. The following expressions of type $\mathbf{L}(A)$ are all equivalent: $[\,c, d, d\,]$, $c \triangleright [\,d, d\,]$, $[\,c, d\,] \triangleleft d$ and $[\,] +\!\!+ [\,c, d\,] +\!\!+ [\,d\,]$.

Processes The most basic notion in mCRL2 is an action. Actions can be *parametrized* with data. In general, we write $\mathsf{a}, \mathsf{b}, \ldots$ to denote action names and $\vec{d}, \vec{e}, \ldots$ to denote vectors of data parameters. In the notation $\mathsf{a}(\vec{d})$, we assume that the vector of data parameters $\vec{d}$ is of the type that is specified for the action name a. An action without data parameters can be seen as an action with an empty vector of data parameters. Actions are allowed to occur simultaneously in mCRL2. In this case we speak about multiactions. A multiaction is a bag of actions that are constructed according to the following BNF:

$$\alpha ::= \tau \mid \mathsf{a}(\vec{d}) \mid \alpha \mid \alpha\,,$$

where $\mathsf{a} \in \mathit{ActLab}$ denotes an action name and $\vec{d}$ a vector of data parameters.

The constructor τ represents the multiaction containing no (observable) actions. This so-called hidden or internal action cannot be observed. It is not very useful when specifying the behavior of processes, but it is essential when it comes to analyzing them. The constructor $\mathsf{a}(\vec{d})$ represents a multiaction that contains only (one occurrence of) the action $\mathsf{a}(\vec{d})$ and the constructor $\alpha \mid \beta$ represents a multiaction containing the actions from the multiactions α and β.

The process operations used in this paper are the ones listed below:

- multiactions as discussed before.
- deadlock $\delta :\rightarrow \mathit{Proc}$. The constant δ models inaction, or the inability to perform (multi)actions. The process δ can delay for an arbitrary long time, but cannot perform any multiaction.
- alternative composition $+ : \mathit{Proc} \times \mathit{Proc} \rightarrow \mathit{Proc}$. The process $p + q$ behaves like p or like q, depending on which of the two performs the first multiaction.
- sequential composition $\cdot : \mathit{Proc} \times \mathit{Proc} \rightarrow \mathit{Proc}$. The process $p \cdot q$ first performs the multiactions of p, until p terminates, and then continues as q.
- conditional operator $\_ \rightarrow \_ : \mathit{Bool} \times \mathit{Proc} \rightarrow \mathit{Proc}$. The process term $b \rightarrow p$ behaves like p if b is equal to **t**, and if b is equal to **f** it behaves like $\delta \vartriangleleft \mathbf{0}$.
- alternative quantification $\sum_{d:D} : \mathit{Proc} \rightarrow \mathit{Proc}$, for each data variable d of sort D. The process $\sum_{d:D} p$ behaves like $p[d_1/d] + p[d_2/d] + \cdots$[3], i.e., as the possibly infinite alternative composition of processes $p[d_i/d]$ for any data term d_i of sort D.
- at-operator $\vartriangleleft : \mathit{Proc} \times \mathbf{T} \rightarrow \mathit{Proc}$. The process $p \vartriangleleft t$ behaves as p, with the restriction that the first multiaction of p must start at time t. The process $p \vartriangleleft t$ can delay until at most time t. If p consists of several alternatives, then only those with the first multiactions starting at time t will remain in $p \vartriangleleft t$. The alternatives that start earlier than t will express that $p \vartriangleleft t$ can delay until that earlier time. The alternatives that start later than t will

[3] The notation $p[e/d]$ denotes the substitution of e for all free occurrences of d in p.

express that $p \vartriangleleft t$ can wait until time t (but not until that later time).

- parallel composition $\| : Proc \times Proc \to Proc$. The process $p \| q$ can first perform a multiaction of p, first perform a multiaction of q, or first perform a multiaction of p and a multiaction of q simultaneously. This means that the multiactions of p and q are *interleaved* and *synchronized*.
- encapsulation $\partial_H : Proc \to Proc$, for $H \subseteq ActLab$. The process $\partial_H(p)$ behaves as the process p where the execution of (multiactions containing) actions from the set H is prohibited.
- restriction operator $\nabla_V : Proc \to Proc$. $\nabla_V(p)$ (also known as *allow*), where V is a set consisting of (non-empty) multisets of action names specifying exactly which multiactions from p are allowed to occur. Restriction $\nabla_V(p)$ disregards the data parameters of the multiactions in p when determining if a multiaction should be allowed.
- communication operator $\Gamma_C : Proc \to Proc$, where C is a set of allowed communications of the form $\mathsf{a}_0 \mid \cdots \mid \mathsf{a}_n \to \mathsf{c}$, with $n \geq 1$ and a_i and c action names. For each communication $\mathsf{a}_0 \mid \cdots \mid \mathsf{a}_n \to \mathsf{c}$, multiactions containing $\mathsf{a}_0(\vec{d}) \mid \cdots \mid \mathsf{a}_n(\vec{d})$ (for some $\vec{d}$) in p are replaced by $\mathsf{c}(\vec{d})$. Note that the data parameters are retained in action c.
- hiding operator $\tau_I : Proc \to Proc$. $\tau_I(p)$ hides (or renames to τ) all actions with an action name in I in all multiactions in p. Hiding $\tau_I(p)$ disregards the data parameters of the actions in p when determining if an action should be hidden.

The precedence of the operators introduced so far, in decreasing order, is as follows: $|, \vartriangleleft, \cdot, \to, \|, \sum, +$. Furthermore, $\cdot$ and $+$ are associative.

A key feature of mCRL2 is that it has the expressive power to describe that a process can delay until a certain time. The process $p + \delta \vartriangleleft t$ can certainly delay until time t, but can possibly delay longer, depending on p. Consequently, the process $\delta \vartriangleleft \mathbf{0}$ can neither delay nor perform multiactions. We follow the intuition that a process that can delay until time t can also delay until an earlier moment, and a process that can perform a *first* multiaction at time t can also delay until time t.

To prove identities in mCRL2 we use a combined many-sorted calculus, which for the sort of processes has the rules of binding-equational calculus [12]. The axioms of mCRL2 are the ones presented in [5].

We consider *systems* of *process definitions* with the right hand sides being mCRL2 process terms extended with parametrized recursive calls of the form $\mathsf{Y}(\vec{t})$ for process name Y with parameters $\vec{t}$. It is possible to transform mCRL2 specifications to a TLPS. A TLPS can be seen as a (symbolic) representation of the transition system of a model. It uses the basic mCRL2 operators only, and in a very specific way. A *timed linear process specification* (TLPS) is a process specification that contains a single process definition of the *linear form*

$$\begin{aligned}\textbf{proc } \mathsf{P}(\overrightarrow{d{:}D}) = &\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} c_i(\overrightarrow{d,e_i}) \to \alpha_i(\overrightarrow{d,e_i}) \vartriangleleft t_i(\overrightarrow{d,e_i}) \cdot \mathsf{P}(\overrightarrow{g_i}(\overrightarrow{d,e_i}))\\ &+ \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} c_j(\overrightarrow{d,e_j}) \to \alpha_j^{\delta}(\overrightarrow{d,e_j}) \vartriangleleft t_j(\overrightarrow{d,e_j});\end{aligned}$$

where data expressions of the form $x(\vec{d})$ contain at most free variables from $\vec{d}$, I and J are finite index sets, and for $i \in I$ and $j \in J$:

- $c_i(\overrightarrow{d,e_i})$ and $c_j(\overrightarrow{d,e_j})$ are boolean expressions representing the conditions,
- $\alpha_i(\overrightarrow{d,e_i})$ is a multiaction $\mathsf{a}_i^1(\overrightarrow{f_i^1}(\overrightarrow{d,e_i}))|\ldots|\mathsf{a}_i^{n_i}(\overrightarrow{f_i^{n_i}}(\overrightarrow{d,e_i}))$, where $\overrightarrow{f_i^k}(\overrightarrow{d,e_i})$ (for $1 \leq k \leq n_i$) are the parameters of action name a_i^k,
- $\alpha_j^{\delta}(\overrightarrow{d,e_j})$ is either δ or a multiaction $\mathsf{a}_j^1(\overrightarrow{f_j^1}(\overrightarrow{d,e_j})) \mid \ldots \mid \mathsf{a}_j^{n_j}(\overrightarrow{f_j^{n_j}}(\overrightarrow{d,e_j}))$, where $\overrightarrow{f_j^k}(\overrightarrow{d,e_j})$ (for $1 \leq k \leq n_j$) are the parameters of action name a_j^k, respectively,
- $t_i(\overrightarrow{d,e_i})$ and $t_j(\overrightarrow{x,y_j})$ are expressions of sort $\mathbf{T}$ representing the time stamps of multiactions $\alpha_i(\overrightarrow{d,e_i})$ and $\alpha_j^{\delta}(\overrightarrow{d,e_j})$, respectively,
- $\overrightarrow{g_i}(\overrightarrow{d,e_i})$ is an expression of sort D representing the next state of the process definition P;

and contains an initialization of the following form, where $\overrightarrow{d_0}$ is a closed data expression.

$$\textbf{init } P(\overrightarrow{d_0});$$

Note that the summands $\sum_{i\in I}$ and $\sum_{j\in J}$ are *meta-level* operations: $\sum_{i\in I} p_i$ is a shorthand for $p_1 + \cdots + p_n$, where $I = \{1, \ldots, n\}$.

The form of the first summand as described above is sometimes presented as the *condition-action-effect* rule. In a particular state $\vec{d}$ and for some data value $\vec{e_i}$ the multiaction $\alpha_i(\overrightarrow{d,e_i})$ can be done at time $t_i(\overrightarrow{d,e_i})$ if condition $c_i(\overrightarrow{d,e_i})$ holds. The effect of the action on the state is given by the fact that the next state is $\overrightarrow{g_i}(\overrightarrow{d,e_i})$.

3 Timed Automata in mCRL2

In this section, we first present the form of an mCRL2 specification by which a single timed automaton can be represented. Then, we consider networks of timed automata with synchronous communication and shared variables.

The process specification in which a single timed automaton can be represented is of the following form[4]:

$$\begin{aligned}&\textbf{proc } \mathsf{X}(\overrightarrow{d{:}D}, t_a{:}\mathbf{T}, v{:}\mathbf{L}(\mathbf{T})) =\\ &\quad\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}}\sum_{t_r:\mathbf{T}}\Big(c_i(\overrightarrow{d,e_i}) \wedge is\_enabled_i(v,t_r)\Big) \to\\ &\qquad\alpha_i(\overrightarrow{d,e_i}) \vartriangleleft (t_a + t_r) \cdot \mathsf{X}(\overrightarrow{g_i}(\overrightarrow{d,e_i}), t_a + t_r, (v + t_r)_{\lambda_i})\\ &+ \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}}\sum_{t_r:\mathbf{T}}\Big(c_j(\overrightarrow{d,e_j}) \wedge can\_wait_j(v,t_r)\Big) \to \delta \vartriangleleft (t_a + t_r);\\ &\textbf{init } \mathsf{X}(\overrightarrow{d_0}, \mathbf{0}, \overrightarrow{\mathbf{0}});\end{aligned}$$

[4]This representation is adapted from [7] where timed μCRL is used.

where

- the parameters in vector $\overrightarrow{d}$ are discrete parameters that have nothing to do with time. In TA translations they contain location names, local variables of the TA, etc. In the same way, the conditions c_i and c_j and the "next state" functions $\overrightarrow{g_i}$ do not depend on time (i.e., the parameter t_a).
- in TA translations the initial parameter $\overrightarrow{d_0}$ contains information of the initial location of the TA and the initial values of local variables.
- parameter t_a represents the absolute (local) time of the process X, which is needed due to the absolute time model of mCRL2. Parameter v is used to store the values of all clock variables. Here we assume that all clocks are numbered from 1 to some number n.
- $(v + t_r)_{\lambda_i}$ represents an update of clock values in v by adding t_r to each element, and resetting (setting to $\mathbf{0}$) of all the clocks in the reset list λ_i.
- the syntax of the conditions $is\_enabled_i$ and can_wait_j is $v.k \equiv n \mid v.k - v.l \equiv n \mid v.k + t_r \equiv n \mid \phi_1 \wedge \phi_2$, where k and l are clock numbers, $\equiv \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbf{N}$ [5].

Networks of Timed Automata In UPPAAL, timed automata can communicate synchronously by means of synchronization of an output action and an input action with the same label, e.g., $a!$ and $a?$ (necessarily) synchronize (if they occur in different timed automata).

Assuming that two timed automata are represented in mCRL2 by means of the processes $\mathsf{X}(\overrightarrow{d_0}, \mathbf{0}, \overrightarrow{\mathbf{0}})$ and $\mathsf{Y}(\overrightarrow{d'_0}, \mathbf{0}, \overrightarrow{\mathbf{0}})$, the network of these two automata is then defined as the parallel composition of the two with the output and input action being forced to communicate. Assuming that the joint alphabet of the two automata $\Sigma = \alpha! \cup \alpha? \cup \beta$ consists of the output actions $\alpha!$, the corresponding input actions $\alpha?$, and the interleaved actions β (including the result a of the communication of $a!$ and $a?$), the corresponding mCRL2 process will be represented as

$$\nabla_{\alpha\cup\beta}(\Gamma_C(\mathsf{X}(\overrightarrow{d_0}, \mathbf{0}, \overrightarrow{\mathbf{0}}) \parallel \mathsf{Y}(\overrightarrow{d'_0}, \mathbf{0}, \overrightarrow{\mathbf{0}})))$$

where $\alpha = \{a \mid \exists_a(a! \in \alpha! \wedge a? \in \alpha?)\}$ and $C = \{(a! \mid a? \rightarrow a) \mid \exists_a(a! \in \alpha! \wedge a? \in \alpha?)\}$.

Note that the result of a communication in UPPAAL is represented by means of the internal action τ. Here we chose to allow the result of a communication to be an observable action. Also, in addition to the UPPAAL automata, we also allow non-synchronizing labels other than the internal action. This allows to make state information observable. In the example discussed in Section 6 we introduce internal actions for entering and leaving the critical section (which is a location of a timed automaton).

In case the visibility of certain actions is undesirable, they can be hidden using the hiding operation $\tau_I(p)$ that renames all actions in I to τ.[6]

[5] Or a condition can be equal to $\mathbf{t}$ (in this case it disappears).

[6] This operation should be used outside communication operations to avoid synchronization of τ with other multiactions.

Shared Variables In UPPAAL it is possible for the processes to communicate asynchronously by means of (globally) shared integer variables and arrays (denoted *SVars*). These shared variables are allowed to occur in the guards and assignments of the transitions and not in the invariants. In case two processes synchronize the execution of their transitions, the changes to the shared variables are first executed for the transition with the output label and then for the transition with the input label. For simplicity we assume that the variables that are assigned in the involved transitions are disjoint.

In mCRL2 for each shared integer variable v:$\mathbf{Z}$ (or v:$\mathbf{L}(\mathbf{Z})$) we introduce a process SV_v with one parameter representing the current value of the variable. The following actions are introduced for modeling the different forms of communication between the processes and the shared variable:

- $\mathsf{s\_get}_v, \mathsf{r\_get}_v, \mathsf{c\_get}_v$:$\mathbf{Z}$ representing the sending, receiving and communicating of a request for the current value of the variable respectively;
- $\mathsf{s\_set}_v, \mathsf{r\_set}_v, \mathsf{c\_get}_v$:$\mathbf{Z}$ representing the sending, receiving and communicating of a reset of the value of the variable respectively.

The following process specification defines the behavior of the shared variable v with value n. It should allow for at most two processes to access the shared variable simultaneously. Due to the assumption of disjointness of the variables that are assigned by the transitions involved in a synchronization, at most one process writes the variable simultaneously.

$$\begin{aligned}
&\mathsf{SV}(n{:}\mathbf{Z}) = \\
&\quad \mathsf{r_get}_v(n) \cdot \mathsf{SV}(n) \;+\; \mathsf{r_get}_v(n) \mid \mathsf{r_get}_v(n) \cdot \mathsf{SV}(n) \\
&\quad + \sum_{m:\mathbf{Z}} \mathsf{r_set}_v(m) \cdot \mathsf{SV}(m) \\
&\quad + \sum_{m:\mathbf{Z}} \mathsf{r_get}_v(n) \mid \mathsf{r_set}_v(m) \cdot \mathsf{SV}(m) \\
&\quad + \sum_{m:\mathbf{Z}} \mathsf{r_get}_v(n) \mid \mathsf{r_get}_v(n) \mid \mathsf{r_set}_v(m) \cdot \mathsf{SV}(m);
\end{aligned}$$

Although the above process specification is not a TLPS, it can easily be transformed into one.

Now, the mCRL2 processes representing the timed automata have to be adapted to additionally generate the relevant $\mathsf{s\_get}$ and $\mathsf{s\_set}$ actions. Let a summand i of X represent a transition of a timed automaton that accesses shared variables $va_1, \ldots, va_n$, and modifies shared variables $vm_1, \ldots, vm_m$ by means of expressions $t_1(\overrightarrow{d, e_i, va}), \ldots, t_m(\overrightarrow{d, e_i, va})$. The checks such a transition performs on the values of the shared variables can be incorporated into the condition of summand i so that it becomes $c_i(\overrightarrow{d, e_i, va})$. The corresponding summand in the process specification for X should look as:

$$\begin{aligned}
&\sum_{i\in I} \sum_{e_i:E_i} \sum_{z_1:\mathbf{Z}} \cdots \sum_{z_n:\mathbf{Z}} \sum_{t_r:\mathbf{T}} \Big(c_i(\overrightarrow{d, e_i, z}) \wedge is_enabled_i(v, t_r)\Big) \rightarrow \\
&\quad \alpha_i(\overrightarrow{d, e_i}) \mid \mathsf{s_get}_{va_1}(z_1) \mid \ldots \mid \mathsf{s_get}_{va_n}(z_n) \mid \\
&\quad \mathsf{s_set}_{vm_1}(t_1(\overrightarrow{d, e_i, z})) \mid \ldots \mid \mathsf{s_set}_{vm_m}(t_m(\overrightarrow{d, e_i, z})) \vartriangleleft (t_a + t_r) \cdot \\
&\qquad\qquad \mathsf{X}(\overrightarrow{g_i(d, e_i)}, t_a + t_r, (v + t_r)_{\lambda_i})
\end{aligned}$$

Since the shared variables are not allowed to occur in the invariants it is not the case that a state change described by

one process is disallowed due to another process becoming inconsistent (since the invariant may be invalidated). The process representing the network of two timed automata represented by X and Y and a shared variable represented by process SV has the following form:

$$\nabla_{\alpha\cup\beta}(\tau_I(\partial_H(\Gamma_{C'}(\mathsf{SV}(0) \parallel \mathsf{X}(\overrightarrow{d_0}, \mathbf{0}, \overrightarrow{\mathbf{0}}) \parallel \mathsf{Y}(\overrightarrow{d'_0}, \mathbf{0}, \overrightarrow{\mathbf{0}})))));$$

where

- $C' = C \cup \{\mathsf{s\_get}_v \mid \mathsf{r\_get}_v \rightarrow \mathsf{c\_get}_v, \mathsf{s\_set}_v \mid \mathsf{r\_set}_v \rightarrow \mathsf{c\_set}_v \mid v \in \mathit{SVars}\}$ describes the synchronous communications of the outputs and inputs of the timed automata and the interactions with the process representing the shared variable.
- $H = \alpha! \cup \alpha? \cup \{\mathsf{s\_get}_v, \mathsf{r\_get}_v\, \mathsf{s\_set}_v, \mathsf{r\_set}_v \mid v \in \mathit{SVars}\}$ denotes the separate synchronous communication actions that need to be blocked.
- $I = \{\mathsf{c\_get}_v, \mathsf{c\_set}_v \mid v \in \mathit{SVars}\}$ denotes the interactions with the process representing the shared variable which need to be made unobservable.

4 Elimination of Parallel Composition

As a result of the transformations in this section the mCRL2 representation of a network of timed automata with shared variables is transformed to a single timed automaton representation in mCRL2.

We focus on the elimination of the parallel composition of two specifications since the other operators can easily be eliminated without affecting the format of the specification. It can be seen from the axioms of mCRL2 that the resulting process specification for the parallel composition of X and Y will have the form given below. Process XY is parametrized by the discrete parameters $\overrightarrow{d{:}D}$ and $\overrightarrow{d'{:}D'}$ of both X and Y; by clock values $v{:}\mathbf{L}(\mathbf{T})$ and $v'{:}\mathbf{L}(\mathbf{T})$ of both X and Y; and by one absolute time parameter $t_a{:}\mathbf{T}$.

$$\begin{aligned}
&\textbf{proc } \mathsf{XY}(\overrightarrow{d{:}D}, \overrightarrow{d'{:}D'}, t_a{:}\mathbf{T}, v{:}\mathbf{L}(\mathbf{T}), v'{:}\mathbf{L}(\mathbf{T})) = \\
&\quad \sum_{i\in I} \sum_{\overrightarrow{e_i:E_i}} \sum_{t_r:\mathbf{T}} \Big(c_i(\overrightarrow{d, e_i}) \wedge \mathit{is_enabled}_i(v, t_r)\Big) \wedge \\
&\qquad \Big(\Big(\bigvee_{i'\in I'} \exists_{\overrightarrow{e'_{i'}:E_{i'}}} c'_{i'}(\overrightarrow{d', e'_{i'}}) \wedge \\
&\qquad\qquad \exists_{t'_r:\mathbf{T}}(\mathit{is_enabled}'_{i'}(v', t'_r) \wedge t_r \le t'_r)\Big) \vee \\
&\qquad \Big(\bigvee_{j'\in J'} \exists_{\overrightarrow{e'_{j'}:E_{j'}}} c'_{j'}(d', e'_{j'}) \wedge \\
&\qquad \exists_{t'_r:\mathbf{T}}(\mathit{can_wait}'_{j'}(v', t'_r) \wedge t_r \le t'_r)\Big)\Big) \rightarrow \alpha_i(\overrightarrow{d, e_i}) \vartriangleleft (t_a + t_r) \cdot \\
&\qquad\qquad \mathsf{XY}(\overrightarrow{g_i}(\overrightarrow{d, e_i}), \overrightarrow{d'}, t_a + t_r, (v + t_r)_{\lambda_i}, v' + t_r) \\
&+ \sum_{i\in I'} \sum_{\overrightarrow{e'_i:E'_i}} \sum_{t_r:\mathbf{T}} \Big(c'_i(\overrightarrow{d', e'_i}) \wedge \mathit{is_enabled}'_i(v', t_r)\Big) \wedge \\
&\qquad \Big(\Big(\bigvee_{i\in I} \exists_{\overrightarrow{e_i:E_i}} c_i(\overrightarrow{d, e_i}) \wedge \exists_{t'_r:\mathbf{T}}(\mathit{is_enabled}_i(v, t'_r) \wedge t_r \le t'_r)\Big) \vee \\
&\qquad \Big(\bigvee_{j\in J} \exists_{\overrightarrow{e_j:E_j}} c_j(d, e_j) \wedge \\
&\qquad \exists_{t'_r:\mathbf{T}}(\mathit{can_wait}_j(v, t'_r) \wedge t_r \le t'_r)\Big)\Big) \rightarrow \alpha'_i(\overrightarrow{d', e'_i}) \vartriangleleft (t_a + t_r) \cdot \\
&\qquad\qquad \mathsf{XY}(\overrightarrow{d}, \overrightarrow{g'_i}(\overrightarrow{d', e'_i}), t_a + t_r, v + t_r, (v' + t_r)_{\lambda'_i}) \\
&+ \sum_{i\in I} \sum_{i'\in I'} \sum_{\overrightarrow{e_i:E_i}} \sum_{\overrightarrow{e'_i:E'_i}} \sum_{t_r:\mathbf{T}} \\
&\qquad \Big(c_i(\overrightarrow{d, e_i}) \wedge c'_{i'}(\overrightarrow{d', e'_{i'}}) \wedge \mathit{is_enabled}_i(v, t_r) \wedge \\
&\qquad \mathit{is_enabled}'_{i'}(v', t_r)\Big) \rightarrow \alpha_i(\overrightarrow{d, e_i}) \mid \alpha'_{i'}(\overrightarrow{d', e'_i}) \vartriangleleft (t_a + t_r) \cdot \\
&\qquad\qquad \mathsf{XY}(\overrightarrow{g_i}(\overrightarrow{d, e_i}), \overrightarrow{g'_{i'}}(\overrightarrow{d', e'_{i'}}), t_a + t_r, (v + t_r)_{\lambda_i}, (v' + t_r)_{\lambda'_{i'}}) \\
&+ \sum_{j\in J} \sum_{j'\in J'} \sum_{\overrightarrow{e_j:E_j}} \sum_{\overrightarrow{e'_j:E'_j}} \sum_{t_r:\mathbf{T}} \\
&\qquad \Big(c_j(\overrightarrow{d, e_j}) \wedge c'_{j'}(\overrightarrow{d', e'_{j'}}) \wedge \mathit{can_wait}_j(v, t_r) \wedge \\
&\qquad \mathit{can_wait}'_{j'}(v', t_r)\Big) \rightarrow \delta \vartriangleleft (t_a + t_r); \\
&\textbf{init } \mathsf{XY}(\overrightarrow{d_0}, \overrightarrow{d'_0}, \mathbf{0}, \overrightarrow{\mathbf{0}}, \overrightarrow{\mathbf{0}});
\end{aligned}$$

The first summand represents the interleaving behavior of the parallel composition where the first action comes from X. The condition of this summand consists of the condition of the corresponding summand from X and the condition stating that the process Y can wait at least until the time of this summand. In case the process Y cannot wait that long, this interleaving is not enabled. As a simple example to illustrate this, $\mathsf{a} \vartriangleleft 5 \parallel \mathsf{b} \vartriangleleft 3$ is equal to $\mathsf{b} \vartriangleleft 3 \cdot \mathsf{a} \vartriangleleft 5$, so only one of the two interleavings of actions a and b is possible here.

The second summand is the symmetric interleaving with the first action from Y. The third summand represents the synchronization of the two processes, for which the conditions of both of the two processes should be satisfied. The last summand represents the ability of the parallel composition to wait to a certain time only if both processes can wait until this time.

As the next step we deal with the quantifiers. We replace the quantifiers over discrete data domains by corresponding sums. These do not pose a problem to our format. To the expressions with quantifiers over the time domain, e.g., $\exists_{t'_r:\mathbf{T}}(\mathit{is\_enabled}(v, t'_r) \wedge t_r \le t'_r))$, we apply a quantifier elimination procedure. As a result we get new formulas that have the same syntax as *is_enabled*. The same procedure is applied to *can_wait* formulas. We define functions $\mathit{can\_postpone}_i(v, t_r)$ for all $i \in I \cup J \cup I' \cup J'$ to be equal to the results of such quantifier eliminations in the following way:

$$\mathit{can\_postpone}_i(v, t_r) = \begin{cases} \exists_{t'_r:\mathbf{T}}(\mathit{is\_enabled}_i(v, t'_r) \wedge t_r \le t'_r)) \\ \qquad\qquad \text{if } i \in I \cup I' \\ \exists_{t'_r:\mathbf{T}}(\mathit{can\_wait}_i(v, t'_r) \wedge t_r \le t'_r)) \\ \qquad\qquad \text{if } i \in J \cup J' \end{cases}$$

As a result, we obtain the following specification which is in the desired form:

$$\begin{aligned}
&\textbf{proc } \mathsf{XY}(\overrightarrow{d{:}D}, \overrightarrow{d'{:}D'}, t_a{:}\mathbf{T}, v{:}\mathbf{L}(\mathbf{T}), v'{:}\mathbf{L}(\mathbf{T})) = \\
&\quad \sum_{i\in I} \sum_{j\in I'\cup J'} \sum_{\overrightarrow{e_i:E_i}} \sum_{\overrightarrow{e_j:E_j}} \sum_{t_r:\mathbf{T}} \\
&\qquad \Big(c_i(\overrightarrow{d, e_i}) \wedge c'_j(\overrightarrow{d', e'_j}) \wedge \mathit{is_enabled}_i(v, t_r) \wedge \\
&\qquad \mathit{can_postpone}_j(v', t_r)\Big) \rightarrow \alpha_i(\overrightarrow{d, e_i}) \vartriangleleft (t_a + t_r) \cdot \\
&\qquad\qquad \mathsf{XY}(\overrightarrow{g_i}(\overrightarrow{d, e_i}), \overrightarrow{d'}, t_a + t_r, (v + t_r)_{\lambda_i}, v' + t_r) \\
&+ \sum_{i\in I'} \sum_{j\in I\cup J} \sum_{\overrightarrow{e'_i:E'_i}} \sum_{\overrightarrow{e_j:E_j}} \sum_{t_r:\mathbf{T}} \\
&\qquad \Big(c'_i(\overrightarrow{d', e'_i}) \wedge c_j(\overrightarrow{d, e_j}) \wedge \mathit{is_enabled}'_i(v', t_r) \wedge \\
&\qquad \mathit{can_postpone}_j(v, t_r)\Big) \rightarrow \alpha'_i(\overrightarrow{d', e'_i}) \vartriangleleft (t_a + t_r) \cdot
\end{aligned}$$

$$\mathsf{XY}(\overrightarrow{d}, \overrightarrow{g_i'}(\overrightarrow{d'}, \overrightarrow{e_i'}), t_a + t_r, v + t_r, (v' + t_r)_{\lambda_i'})$$
$$+ \sum_{i\in I} \sum_{i'\in I'} \sum_{\overrightarrow{e_i:E_i}} \sum_{\overrightarrow{e_i':E_i'}} \sum_{t_r:\mathbf{T}}$$
$$\Big(c_i(\overrightarrow{d, e_i}) \wedge c'_{i'}(\overrightarrow{d', e_{i'}'}) \wedge \mathit{is\_enabled}_i(v, t_r) \wedge$$
$$\mathit{is\_enabled}'_{i'}(v', t_r)\Big) \rightarrow \alpha_i(\overrightarrow{d, e_i}) \mid \alpha'_{i'}(\overrightarrow{d', e_i'}) \vartriangleleft (t_a + t_r) \cdot$$
$$\mathsf{XY}(\overrightarrow{g_i}(\overrightarrow{d, e_i}), \overrightarrow{g_{i'}'}(\overrightarrow{d', e_{i'}'}), t_a + t_r, (v + t_r)_{\lambda_i}, (v' + t_r)_{\lambda_{i'}'})$$
$$+ \sum_{j\in J} \sum_{j'\in J'} \sum_{\overrightarrow{e_j:E_j}} \sum_{\overrightarrow{e_j':E_j'}} \sum_{t_r:\mathbf{T}}$$
$$\Big(c_j(\overrightarrow{d, e_j}) \wedge c'_{j'}(\overrightarrow{d', e_{j'}'}) \wedge \mathit{can\_wait}_j(v, t_r) \wedge$$
$$\mathit{can\_wait}'_{j'}(v', t_r)\Big) \rightarrow \delta \vartriangleleft (t_a + t_r);$$

init $\mathsf{XY}(\overrightarrow{d_0}, \overrightarrow{d_0'}, \mathbf{0}, \overrightarrow{\mathbf{0}}, \overrightarrow{\mathbf{0}})$;

The resulting equation has the form of the mCRL2 representation of a single TA in Section 3.

5 Discretization and Finalization

In [7] we presented a number of transformation steps to transform μCRL representations of timed automata into a form that uses discrete and finite data types only. Here we combine these discretization steps in just one step.

The idea behind discretization is based on the notion of clock regions. Instead of the time-valued parameters t_a and $\overrightarrow{v}$ we use the natural valued parameters t'_a and $\overrightarrow{v'}$, and one additional parameter *ord* that keeps track of the order of the fractional parts of t_a and $\overrightarrow{v}$. In fact, instead of the fractional parts of clock values $\overrightarrow{v}$ we keep the fractional values of the times these clocks were reset for the last time.

Parameter *ord* can be implemented as a list of natural numbers, representing the position numbers of the fractional values. We choose the first element of the list (*ord*.0) to represent the position number of the fractional part of the current time (t_a), and for any clock with the number i, *ord*.i represents the position of the fractional part of the last time the clock i was reset. For example, the ordering $[\,1, 0, 2\,]$ says that the fractional part of current time is bigger than the last time clock 1 was reset, and smaller than the fractional part of the last time when clock 2 was reset.

Positions in such an ordering *ord* can be represented by a pair consisting of a natural number p, representing the position number in *ord*, and a boolean x, saying if the position is exact or "just before" the position in the ordering. For example, if $p = 2$ and $x = \mathbf{f}$, position (p, x) represents a position between 1 and 2.

Given these assumptions it is not difficult to define the predicates $\mathit{is\_lt}(\mathit{ord}, i, j)$, $\mathit{is\_eq}(\mathit{ord}, i, j)$, $\mathit{is\_pos\_lt}(\mathit{ord}, i, p, x)$, $\mathit{is\_pos\_eq}(\mathit{ord}, i, p, x)$, and $\mathit{is\_pos\_gt}(\mathit{ord}, i, p, x)$. The first one represent the fact that the fractional part of the last reset time of clock i is less than the fractional part of the last reset time of clock j. The second one represents the fact that these parts are equal. The third predicate represents the fact that the position (p, x) is smaller than the position of the fractional part of the last reset time of clock i.

It is also possible to define the update function $\mathit{upd\_ord}(\mathit{ord}, p, x, \lambda)$ that sets the element 0 and the elements indexed by λ to have the position represented by (p, x). For example, $\mathit{upd\_ord}([\,1, 0, 2\,], 1, \mathbf{f}, [\,\mathbf{f}, \mathbf{t}\,])$ is equal to $[\,1, 0, 1\,]$ (the position $(1, \mathbf{f})$ can be seen as position 1.5 in $[\,1, 0, 2\,]$; setting the elements 0 and 2 of the list to 1.5 gives $[\,1.5, 0, 1.5\,]$; normalization of this list gives $[\,1, 0, 1\,]$).

The predicate $\mathit{compat}(\mathit{ord}, p, x)$ gives true if position (p, x) is compatible with *ord* (p should be in range of the elements in *ord*). For the reference implementation of these functions we refer to Appendix A.

The representation of the TA X is transformed to the following specification:

$$\textbf{proc}\ \mathsf{X}'(\overrightarrow{d{:}D}, t_{a'}{:}\mathbf{N}, v'{:}\mathbf{L}(\mathbf{N}), \mathit{ord}{:}\mathbf{L}(\mathbf{N})) =$$
$$\sum_{i\in I} \sum_{\overrightarrow{e_i:E_i}} \sum_{t_{r'}:\mathbf{N}} \sum_{p:\mathbf{N}} \sum_{x:\mathit{Bool}}$$
$$((t_{r'} > 0 \vee \mathit{is\_pos\_geq}(\mathit{ord}, 0, p, x)) \wedge \mathit{compat}(\mathit{ord}, p, x) \wedge$$
$$c_i(\overrightarrow{d, e_i}) \wedge \mathit{is\_enabled}'_i(v', t_{r'})) \rightarrow \alpha_i(\overrightarrow{d, e_i}) \vartriangleleft (t_a + t_{r'}) \cdot$$
$$\mathsf{X}'(\overrightarrow{g_i}(\overrightarrow{d, e_i}), t_{a'} + t_{r'}, (v' + t_{r'})_{\lambda_i}, \mathit{upd\_ord}(\mathit{ord}, p, x, \lambda_i))$$
$$+ \sum_{j\in J} \sum_{\overrightarrow{e_j:E_j}} \sum_{t_{r'}:\mathbf{N}} \sum_{p:\mathbf{N}} \sum_{x:\mathit{Bool}}$$
$$((t_{r'} > 0 \vee \mathit{is\_pos\_geq}(\mathit{ord}, 0, p, x)) \wedge \mathit{compat}(\mathit{ord}, p, x) \wedge$$
$$(c_j(\overrightarrow{d, e_j}) \wedge \mathit{can\_wait}'_j(v', t_{r'})) \rightarrow \delta \vartriangleleft (t_{a'} + t_{r'});$$

init $\mathsf{X}'(\overrightarrow{d_0}, \overrightarrow{\mathbf{0}}, \overrightarrow{\mathbf{0}})$;

In Table 1 we present the rules by which the discrete versions of conditions from *is_enabled*′ and *can_wait*′ are being generated from the conditions in *is_enabled* and *can_wait*, respectively.

Constraint	Discretized constraint
$v(c_i) < n$	$\mathit{if}(\mathit{is\_lt}(\mathit{ord}, 0, i), v'(c_i) \le n, v'(c_i) < n)$
$v(c_i) \le n$	$\mathit{if}(\mathit{is\_lt}(\mathit{ord}, i, 0), v'(c_i) < n, v'(c_i) \le n)$
$v(c_i) = n$	$\mathit{is\_eq}(\mathit{ord}, 0, i) \wedge v'(c) = n$
$v(c_i) \ge n$	$\mathit{if}(\mathit{is\_lt}(\mathit{ord}, 0, i), v'(c_i) > n, v'(c_i) \ge n)$
$v(c_i) > n$	$\mathit{if}(\mathit{is\_lt}(\mathit{ord}, i, 0), v'(c_i) \ge n, v'(c_i) > n)$
$v(c_i) + u < n$	$\mathit{if}(\mathit{is\_pos\_lt}(\mathit{ord}, i, p, x), v'(c_i) + u' \le n, v'(c_i) + u' < n)$
$v(c_i) + u \le n$	$\mathit{if}(\mathit{is\_pos\_gt}(\mathit{ord}, i, p, x), v'(c_i) + u' < n, v'(c_i) + u' \le n)$
$v(c_i) + u = n$	$\mathit{is\_pos\_eq}(\mathit{ord}, i, p, ex) \wedge v'(c) + u' = n$
$v(c_i) + u \ge n$	$\mathit{if}(\mathit{is\_pos\_lt}(\mathit{ord}, i, p, x), v'(c_i) + u' > n, v'(c_i) + u' \ge n)$
$v(c_i) + u > n$	$\mathit{if}(\mathit{is\_pos\_gt}(\mathit{ord}, i, p, x), v'(c_i) + u' \ge n, v'(c_i) + u' > n)$
$v(c_i) - v(c_j) < n$	$\mathit{if}(\mathit{is\_lt}(\mathit{ord}, j, i), v'(c_i) - v'(c_j) \le n, v'(c_i) - v'(c_j) < n)$
$v(c_i) - v(c_j) \le n$	$\mathit{if}(\mathit{is\_lt}(\mathit{ord}, i, j), v'(c_i) - v'(c_j)' < n, v'(c_i) - v'(c_j) \le n)$
$v(c_i) - v(c_j) = n$	$\mathit{is\_eq}(\mathit{ord}, i, j) \wedge v'(c) - v'(c_j) = n$
$v(c_i) - v(c_j) \ge n$	$\mathit{if}(\mathit{is\_lt}(\mathit{ord}, j, i), v'(c_i) - v'(c_j) > n, v'(c_i) - v'(c_j) \ge n)$
$v(c_i) - v(c_j) > n$	$\mathit{if}(\mathit{is\_lt}(\mathit{ord}, i, j), v'(c_i) - v'(c_j) \ge n, v'(c_i) - v'(c_j) > n)$

Table 1. Rules for Constraint Discretization

Limiting the values of parameters and sum variables can be done in a similar way to Section 6 of [7]. As a result we obtain a system with only a finite number of states.

6 Verification of Fischer's protocol

In this section, we discuss the approach for verification of timed automata as set out in the previous sections on the well-known Fischer's protocol (cf. [9]) for establishing mutual exclusion. We quote the problem statement from [9]: "We consider a system of n sequential threads of control called processes, which communicate through shared multi-writer, multi-reader atomic registers. Communication consists of read and write operations, each of which is assumed in this paper to be executed instantaneously."

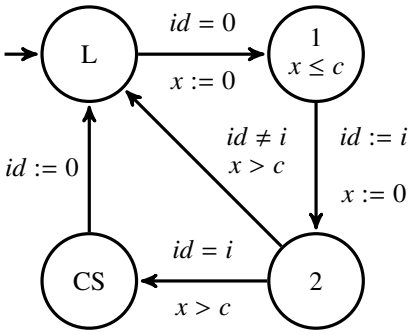
In [9], among others a timing-based solution to this problem is discussed that is attributed to Mike Fischer. In this solution, bounds in $[c_1, c_2]$ are assumed for the time between successive steps of a process; when it is trying to enter its critical section or when it is leaving its critical section. We assume that $0 < c_1 \leq c_2 < \infty$.

Below we present pseudo-code for the algorithm:

```
while true do
begin
    'non-critical section';
    L: if id ≠ 0 then goto L;
    1 : id := i;
    2: pause(c);
    3: if id ≠ i then goto L;
    'critical section';
    id := 0;
end
```

The key idea behind the algorithm is to delay each process i for a period greater than c after it has written id (in line 1) and before testing it (in line 3). Thus, by the time process i has reached line 3, any process j that has passed the test in line L and might overwrite id with value j, has already done so. If process i reads id to be i in line 3, it can safely enter the critical region, since all other processes are either before line L and will not pass it, or after line 1 with their index overwritten by i, so they will fail the test in line 3.

Next, we present the UPPAAL model of Fischer's protocol (from [1]) that closely reflects the above pseudo-code. Note that the register is modeled by means of the shared variable id.



Following the simplifications made in [1], we assume that waiting longer than c is possible. The algorithm has the property that if the timing constraints are met, and the value of the delay parameter is chosen to be strictly greater than c_2/c_1, then mutual exclusion and deadlock-freedom are guaranteed.

Below we present an mCRL2 model with two processes that is obtained by translating the UPPAAL model. With each location of the timed automaton a recursive definition is associated. For the shared variable we introduce a separate process ID. Note that writing the shared variable and reading it are thus modeled as interactions between the P-processes and the process ID. In this example no action synchronization and simultaneous writing and reading of the shared variable takes place, so the process ID is simplified.

$$\textbf{act}\ \ \mathsf{s\_get}_{id}, \mathsf{r\_get}_{id}, \mathsf{c\_get}_{id}, \mathsf{s\_set}_{id}, \mathsf{r\_set}_{id}, \mathsf{c\_set}_{id}{:}\mathbf{N};\ \mathsf{cs}_{\mathsf{in}}, \mathsf{cs}_{\mathsf{out}}{:}\mathbf{N}^{+};$$

$$\textbf{proc}\ \mathsf{P}(pid{:}\mathbf{N}^{+}) = \mathsf{P}_{\mathsf{L}}(pid, 0, 0);$$

$$\mathsf{P}_{\mathsf{L}}(pid{:}\mathbf{N}^{+}, t{:}\mathbf{T}, x{:}\mathbf{T}) = \sum_{u:\mathbf{T}} (0 \leq 2) \rightarrow \mathsf{s\_get}_{id}(0) \triangleleft (t+u) \cdot \mathsf{P}_{1}(pid, t+u, 0);$$

$$\mathsf{P}_{1}(pid{:}\mathbf{N}^{+}, t{:}\mathbf{T}, x{:}\mathbf{T}) = \sum_{u:\mathbf{T}} (x+u \leq 2) \rightarrow \mathsf{s\_set}_{id}(pid) \triangleleft (t+u) \cdot \mathsf{P}_{2}(pid, t+u, 0);$$

$$\mathsf{P}_{2}(pid{:}\mathbf{N}^{+}, t{:}\mathbf{T}, x{:}\mathbf{T}) = \sum_{u:\mathbf{T}} (x+u > 2) \rightarrow (\mathsf{s\_get}_{id}(pid) \mid \mathsf{cs}_{\mathsf{in}}(pid)) \triangleleft (t+u) \cdot \mathsf{P}_{\mathsf{CS}}(pid, t+u, x+u) + \sum_{id:\mathbf{N}} \sum_{u:\mathbf{T}} (id \neq pid \wedge x+u > 2) \rightarrow \mathsf{s\_get}_{id}(id) \triangleleft (t+u) \cdot \mathsf{P}_{\mathsf{L}}(pid, t+u, x+u);$$

$$\mathsf{P}_{\mathsf{CS}}(pid{:}\mathbf{N}^{+}, t{:}\mathbf{T}, x{:}\mathbf{T}) = \sum_{u:\mathbf{T}} (\mathsf{cs}_{\mathsf{out}}(pid) \mid \mathsf{s\_set}_{id}(0)) \triangleleft (t+u) \cdot \mathsf{P}_{\mathsf{L}}(pid, t+u, x+u);$$

$$\mathsf{ID}(id{:}\mathbf{N}) = \sum_{n:\mathbf{N}} \mathsf{r\_set}_{id}(n) \cdot \mathsf{ID}(n) + \mathsf{r\_get}_{id}(id) \cdot \mathsf{ID}(id);$$

$$\textbf{init}\ \ \tau_{\{\mathsf{c\_get}_{id}, \mathsf{c\_set}_{id}\}}(\nabla_{\{\mathsf{cs}_{\mathsf{in}}|\mathsf{c\_get}_{id}, \mathsf{cs}_{\mathsf{out}}|\mathsf{c\_set}_{id}, \mathsf{c\_get}_{id}, \mathsf{c\_set}_{id}\}}(\Gamma_{\{\mathsf{s\_get}_{id}|\mathsf{r\_get}_{id} \rightarrow \mathsf{c\_get}_{id}, \mathsf{s\_set}_{id}|\mathsf{r\_set}_{id} \rightarrow \mathsf{c\_set}_{id}\}}(\mathsf{ID}(0) \parallel \mathsf{P}(1) \parallel \mathsf{P}(2)))));$$

This model is in the input format of the transformation steps for obtaining a discretized representation as in [7].

As a result of transformation steps presented in the previous sections, state-space generation and reduction of the resulting untimed state-space modulo branching bisimulation equivalence we get the LTS from Figure 1.

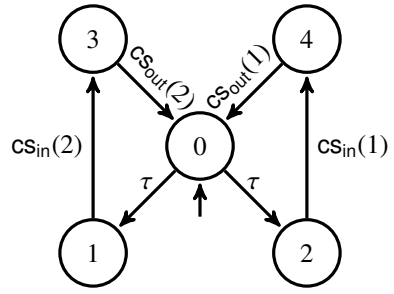


Figure 1. LTS modulo branching bisimulation

Minimization modulo weak-trace equivalence, which identifies more processes, gives the LTS from Figure 2. The correctness property of the Fischer's protocol (exclusivity of the critical regions) is a safety property, which is preserved by the weak-trace equivalence. This example illustrates that after applying the verification method set out in this paper, the further untimed verification is possible by using existing untimed tools.

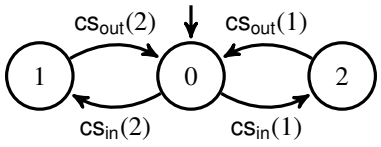


Figure 2. LTS modulo weak trace equivalence

7 Conclusions and Future Work

The key transformation presented here is the flattening of the parallel composition of two TA representations in mCRL2 into a single one. Such an operation is well-known in the TA world, but lifting it to the level of timed process algebra can lead to generalizations beyond TA.

We illustrate the applicability of the method on the well-known example of the Fischer's protocol, and use the UPPAAL specification of it as the input. As a result, we obtain an explicit LTS corresponding to the specification.

As the next step we would like to factorize the time-related parameters to be able to deal with them like with zones. Zones, as well as the operations on them could be specified as an abstract data type in mCRL2, either as a set of clock constraints or using difference-bound matrices.

Going further, one could analyze where exactly the fact that we are dealing with TA has been used and try to generalize some of the results, perhaps lifting some restrictions on the clock constraints. A good first step in this direction is analyzing whether disjunction and negation of clock constraints break the transformations steps.

A full implementation of the presented transformations in a tool and its application to other examples of timed systems is of a particular interest to us.

References

[1] L. Aceto, A. Ingolfsdottir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[2] R. Alur and D. L. Dill. A theory of timed automata. *TCS*, 126:183–235, 1994.

[3] J. C. M. Baeten and C. A. Middelburg. *Process Algebra with Timing*. Monographs in TCS. Springer, 2002.

[4] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[5] J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007.

[6] J. F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes 1994*, Workshop in Computing, pages 26–62. Springer, 1995.

[7] J. F. Groote, M. A. Reniers, and Y. S. Usenko. Time abstraction in timed μCRL a la regions. In *IPDPS*. IEEE, 2006.

[8] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[9] N. A. Lynch and N. Shavit. Timing-based mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1992.

[10] M. A. Reniers, J. F. Groote, J. J. van Wamel, and M. B. van der Zwaag. Completeness of Timed μCRL. *Fund. Inf.*, 50(3-4):361–402, 2002.

[11] M. A. Reniers and Y. S. Usenko. Analysis of timed processes with data using algebraic transformat ions. In *Proc. TIME'05*, pages 192–194. IEEE Computer Society, 2005.

[12] Y. Sun. An algebraic generalization of Frege structures — binding algebras. *TCS*, 211:189–232, 1999.

[13] Y. S. Usenko. *Linearization in μCRL*. PhD thesis, Eindhoven University of Technology, December 2002.

[14] T. A. C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. PhD thesis, Eindhoven University of Technology, 2003.

A *Ord* Operations in mCRL2

```
sort Ord=List(Nat);
map
  is_le,is_eq:Ord#Nat#Nat->Bool;
  is_pos_le,is_pos_eq,
  is_pos_gt:Ord#Nat#Nat#Bool->Bool;
  upd_ord:Ord#Nat#Bool#List(Bool)->Ord;
  compat:Ord#Nat#Bool->Bool;

  upd_list:
    Ord#Nat#List(Bool)->Ord; % upd_list(ord,n,updates)
                             % replace elements marked
                             % in updates by n
  max_list:Ord->Nat;    % maximal element of the list
  x2p1:Ord->Ord;        % apply 2n+1 to all elements
  compress:Ord->Ord;    % keeps only the order of
                        % the elements
                        % (compress([5,7,3])=[1,2,0])
  compress_index:       % the second list is
    Ord#Ord->Ord;       % used as the index
  index:Ord#Nat->Nat;   % returns the index
                        % of the element
  sort_unique:Ord->Ord; % sorts the list removing
                        % repeating elements
  insert_unique:        % insert an element into a
   Ord#Nat->Ord;        % sorted list if not yet there

var n,n1,clock,pos:Nat; exact:Bool;
    resets,lb:List(Bool); l,ln1,ord:Ord;
eqn
  is_le(ord,n,n1)=ord.n<ord.n1;
  is_eq(ord,n,n1)=ord.n==ord.n1;
  is_pos_le(ord,clock,pos,false)=pos<=ord.clock;
  is_pos_le(ord,clock,pos,true)=pos<ord.clock;
  is_pos_eq(ord,clock,pos,false)=false;
  is_pos_eq(ord,clock,pos,true)=pos==ord.clock;
  is_pos_gt(ord,clock,pos,exact)=pos>ord.clock;

  compat(ord,n,true)=max_list(ord)>=n;
  compat(ord,n,false)=succ(max_list(ord))>=n;
  upd_ord(ord,pos,false,resets)=
    compress(upd_list(x2p1(ord),2*pos,true|>resets));
  upd_ord(ord,pos,true,resets)=
    compress(upd_list(x2p1(ord),succ(2*pos),
                                    true|>resets));
  upd_list([],n,[])=[];
  upd_list(n1|>l,n,false|>lb)=n1|>upd_list(l,n,lb);
  upd_list(n1|>l,n,true|>lb)=n|>upd_list(l,n,lb);
  compress(l)=compress_index(l,sort_unique(l));
  compress_index([],l)=[];
  compress_index(n|>ln1,l)=
    index(l,n)|>compress_index(ln1,l);
  sort_unique([])=[];
  sort_unique(n|>l)=insert_unique(sort_unique(l),n);
  insert_unique([],n)=[n];
  (n<n1)->insert_unique(n1|>l,n)=n|>n1|>l;
  (n==n1)->insert_unique(n1|>l,n)=n|>l;
  (n>n1)->insert_unique(n1|>l,n)=
                              n1|>insert_unique(l,n);
  index(n|>l,n)=0;
  (n!=n1)->index(n1|>l,n)=Pos2Nat(succ(index(l,n)));
  max_list([])=0;
  max_list(n|>l)=max(n,max_list(l));
  x2p1([])=[];
  x2p1(n|>l)=succ(2*n)|>x2p1(l);
```