Timed verification with μCRL

Stefan Blom[1], Natalia Ioustinova[1], and Natalia Sidorova[2]

[1] Department of Software Engineering, CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Stefan.Blom@cwi.nl, Natalia.Ioustinova@cwi.nl
[2] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513, 5612 MB Eindhoven, The Netherlands
n.sidorova@tue.nl

Abstract. μCRL is a process algebraic language for specification and verification of distributed systems. μCRL allows to describe temporal properties of distributed systems but it has no explicit reference to time. In this work we propose a manner of introducing discrete time without extending the language. The semantics of discrete time we use makes it possible to reduce the time progress problem to the diagnostics of "no action is enabled" situations. The synchronous nature of the language facilitates the task. We show some experimental verification results obtained on a timed communication protocol.
Keywords: modelling, verification, discrete time, μCRL, model checking.

1 Introduction

The specification language μCRL [9] (micro Common Representation Language) is a process algebraic language that was especially developed to take account of data in the study of communicating processes. The μCRL toolset [5] provides support for enumerative model checking. One of the most important application areas for μCRL is specification and verification of communication protocols. Communication protocols are mostly *timed* systems. A common way to use time is the timeout. In some cases it is possible to abstract from duration and simulate timeouts with non deterministic choice. However, in other cases the lengths of the timeouts are essential to the correctness of the protocol. To deal with these cases one needs an explicit notion of time.

In [8], a timed version of the μCRL language is proposed where time is incorporated in μCRL as an abstract data type satisfying a few conditions plus a construct to make an action happen at a specific time. The timed version of the language turned out to be useful as a formalism for the specification and analysis of hybrid systems [10]. However, it is not clear yet whether timed μCRL can be used to analyse larger systems than the examples considered in the paper. Moreover, some of the existing tools cannot be used for timed μCRL without modification. Most importantly, timed μCRL is incompatible with linearisation (translating a specification into the intermediate format) and with partial order reduction.

The goal of the work we present in this paper is to establish a framework in which timed verification may proceed using the existing untimed tools. The reason why we want to do this is that real life systems tend to consist of both timed and untimed components. Timed tools are very good at detailed analysis of the timed components of these systems, but cannot be used for analysing the whole system.

To achieve the goal of timed verification with untimed tools, we must restrict ourselves to discrete relative time: the state spaces of systems with dense or absolute time are almost always infinite. Techniques, such as regions and zones, which allow finite representations of such infinite state spaces, are not implemented in the untimed tools. Timestamping actions with "absolute" time, as is done in timed μCRL, leads to infinite state spaces in case of unbounded delays: Consider the process $X = \sum_{t=0}^{\infty} \mathtt{a}@t$ which uses time tags (the @ symbol must be read as "at time") and thus can do action `a` at any time. The transition system of X consists of 2 states and infinitely many transitions. For this reason, we have chosen a "relative" time solution. Namely, we introduce time through an action `tick`, which by convention expresses one unit of time elapsing. In this case we can specify the process that can do `a` at any time as $Y = \mathtt{tick}.Y + \mathtt{a}$. The transition system of process Y has two states and two transitions. The advantage of representing time progression as an action is that we stay within the syntax of μCRL. Moreover, the special use of tick is compatible with the semantics of μCRL and hence the existing toolset can be used for analysis and verification.

The proposed discrete time semantics is suitable to express time aspects and analyse time properties of a large class of systems. We argue the usefulness of our approach with the verification experiments on a μCRL specification of the positive acknowledgment retransmission protocol (PAR) [15], whose behaviour depends on the timers' settings.

To express *timed* properties of systems, we introduce an LTL-like timed temporal logic on actions and show how to encode its timed constraints with the use of `tick`, which results in *untimed* temporal formulas. These formulas can be then translated to μ-calculus and checked with the μCRL toolset.

The rest of the paper is organized as follows. In the following Section 2, we sketch the syntax and semantics of μCRL. In Section 3 we present the discrete time semantics that we work with and afterwards in Section 4 we explain how timed specifications can be developed within the untimed framework with following the proposed approach. In Section 5 we discuss some experimental results. In section 6 we introduce a timed temporal logic. We conclude in Section 7 with discussing the related works and the directions for the future work.

2 μCRL: basic notions

The specification language μCRL (micro Common Representation Language) is essentially an extension of the process algebra ACP with abstract data types

```
sort Bool
func T,F: ->Bool
map  eq,and:Bool#Bool->Bool
var  b:Bool
rew  eq(b,b)=T
     eq(T,F)=F
     eq(F,T)=F
     and(T,b)=b
     and(F,b)=F
```

Fig. 1. A μCRL specification of the sort Bool.

```
act  a
     b,c:Bool
proc X=a.X
     Y=sum(b':Bool,b(b').c(b').Y)
     Y'(b1:Bool,state:Bool)=
           sum(b':Bool,b(b').Y'(b',F)<|eq(state,T)|>delta+
           c(b1).Y'(b',T)<|eq(state,F)|>delta
```

Fig. 2. Components in μCRL

and recursive definitions. The μCRL toolset provides tool support for a subset of the μCRL language. In the remainder of this section, we will give an overview of both the language and its tool support. Details about the language can be found in [8]. Details about the tool support can be found in [5].

Data in μCRL is specified using equational abstract data types. Every μCRL specification must include a specification of the sort `Bool`, which represents the booleans. An example can be found in Fig. 1. Although the language specification does not require them, the tool support requires the function `eq : D#D-> Bool` for every sort `D` and the boolean function `and`. The equations following the keyword `rew` are oriented from left to right and used as rewrite rules by the tools, but may be used in both directions for reasoning.

The usual way of modeling a system in μCRL is to decompose the system into components and then specify the components and their interactions separately.

Components are usually recursively defined using atomic actions, alternative composition and conditionals. For example, in Fig. 2 we have specified processes X and Y. Process X simply repeats action `a` infinitely often. Process Y infinitely often chooses between T and F nondeterministically and performs `b` and `c` with the same choice as argument.

Component processes can be glued into a system using parallel composition, encapsulation, hiding and renaming operators. The parallel composition operator of μCRL combines two processes into a single process by allowing nondeterministic choice between interleaving the actions of the two processes and synchronous interaction (communication) of an action from each of the processes. Other types of parallel composition operators can be defined in terms of the basic operators. For example, the operator `|{tick}|` which lets the processes X and Y run interleaved except for the action tick, which must be performed synchronously by both X and Y may be encoded as follows:

```
act  tick tick'
comm tick|tick=tick'
     X |{tick}| Y = rename({tick'->tick},encap({tick},X||Y))
```

In this encoding we have a process `X||Y` where tick actions from X and Y may be performed interleaved or at the same time resulting in a `tick'`. In the process `encap({tick},X||Y)` the interleaved execution is disallowed by means of encapsulation. Finally the `tick'` is renamed to `tick` to get the desired result. Note that interaction is not limited to two parties. The result of an interaction may itself interact. For example, the interaction of actions `a`, `b` and `c`, resulting in action `d` can be expressed as

```
comm   a|b=ab b|c=bc a|c=ac    a|bc=d b|ac=d c|ab=d
```

Tool support for μCRL is centered around the linear process format. A linear specification consists of a single recursive process, which can choose between possibilities of the form action followed by a recursive call:

$$
\begin{array}{l}
\texttt{proc}\ X(d_1 : D_1 \cdots, d_n : D_n) = \\
\qquad\qquad \sum_{e_{11}:E_{11}} \cdots \sum_{e_{1n_1}:E_{1n_1}} a_1(\vec{s}_1).X(\vec{t}_1) \lhd c_1 \rhd \delta + \\
\qquad\qquad \vdots \\
\qquad\qquad \sum_{e_{k1}:E_{k1}} \cdots \sum_{e_{kn_k}:E_{kn_k}} a_k(\vec{s}_k).X(\vec{t}_k) \lhd c_k \rhd \delta + \\
\texttt{init}\ X(\vec{t}_0)
\end{array}
$$

The toolset allows the transformation of specifications into linear form (on Fig. 2, process Y' is a linear equivalent of process Y), the optimisation of specification in linear form, the simulation of a linear specification, and the generation of a Labelled Transition System (LTS) from a linear specification. The toolset allows the user to apply a partial order method based on τ-confluence. Partial order reduction guarantees that the reduced state space is branching bisimilar to the original state space. Branching bisimulation preserves CTL*-X properties [13], so all CTL*-X properties are preserved by partial order reduction.

3 Semantics of Time

In this section we discuss what time semantics is appropriate for our purpose.

The first choice to be made is a choice between dense and discrete time. It is normally assumed that real-time systems operate in "real", continuous time (though some physicists contest against the statement that the changes of a system state may occur at any real-numbered time point). Due to the development of regions and zones techniques [1] the verification of real-time systems became possible. However, a less expensive, discrete time solution is for many systems as good as dense time in the modelling sense, and better than the dense one when the verification is concerned: [11] showed that discrete time suffices for a large and important class of systems and properties including all systems that can be modelled as timed transition systems and such properties as time-bounded invariance and time-bounded response. Another work that compares the use of dense and discrete time is [7]; the authors state that discrete time automata can be analysed using any representation scheme used for dense time, and *in addition* can benefit from enumerative and symbolic techniques (such as BDDs) which are not naturally applicable to dense time. Having in mind that we should not step out the current non-timed framework of μCRL the choice for discrete time is obvious.

The nature of systems under consideration suggests the choice for timers, not clocks: expiration of a timer is a natural way to model an interruption for hardware or a trigger for a software event. Both interrupt and software event must be handled, and they must be handled exactly once, i.e. with taking an event guarded by a timer condition, we assume that the timer which triggered this event became deactivated (otherwise, the system could handle one event several times). Time progresses by decreasing the values of all active timers by one time unit. We will refer to the time progress action as `tick` and to the period of time between two `tick`'s as a time slice.

We consider a class of systems where delays are significantly larger than the duration of normal events within the system. Therefore, we assume system transitions to be instantaneous. This assumption leads us to the conclusion that time progress can never take place if there is still an untimed action enabled, or in other words, the time-progress transition has the least priority in the system and may take place only when the system is *blocked*: there is no any transition enabled except for time progress and communication with the environment.

4 Specifying timed systems in μCRL

In the μCRL framework, we can implement timers as data parameterising processes. Figure 3 shows the specification of the sort `Timer`. Terms `on(n)` stand for active timers (`n` is of sort natural numbers) while deactivated timers are represented by `off` terms. (Note that μCRL specifications containing sort `Timer` should also include sort `Nat` providing operation `pred` that decreases a non-zero natural number by one and operation `eq` checking for the equality of two numbers.) The operations we allow on timers are (1) setting a timer to a value given by a natural number that shows the time delay left until the timer expiration; (2) resetting a timer (setting it to `off`). Timer expiration condition given by predicate `expired` is the check whether the delay until the timer expiration is zero. We assume that the action guarded by the timer expiration is normally resetting the timer or setting it to a positive value, though we leave the designer the freedom to continue considering the timer as set to `on(0)`.

```
sort Timer                            var  t:Timer
func off:-> Timer                          n:Nat
     on:Nat->Timer                    rew  expired(off)=F
map  pred:Timer->Timer                     expired(on(n))=eq(0,n)
     expired:Timer->Bool                   pred(on(n))=on(pred(n))
     set: Timer # Nat -> Timer             pred(off)=off
     reset: Timer -> Timer                 set(t, n)=on(n)
                                           reset(t)=off
```

Fig. 3. A μCRL specification of the sort Timer.

Following the time semantics described in Section 3, we want to model the time progress by the `tick` action, which is a global action decreasing all active timers of the system by one and enabled only when the system is blocked. To achieve this, we enable the `tick` action in a component if that component is blocked and if every timer in that component is off or non-zero. By combining components with the `|{tick}|` operator as defined in Section 2 rather than the standard `||` operator we get precisely the desired behaviour.

A system is considered blocked if there are no *urgent* actions possible. As μCRL has no priority mechanism, there is a potential for technical problems, which we solve by following a specification discipline.

First, we classify a number of actions as *urgent*. Enabled *internal* operators are urgent — they take zero time and, hence, they may not be postponed until the next time slice, and `tick` may not be proposed as an alternative to an internal action. The situation with the *communication* is more complicated: When the two communicating parties are both ready to communicate, the communication should take place in the current time slice. Thus, no `tick` action can be given as an alternative to a communication action. However, when only one of the parties is willing to communicate, time progress should *not* be disabled, meaning that the process willing to communicate but not having this chance yet, should be able to take the `tick` action. We resolve the problem by introducing the asymmetry into communication: Though μCRL has no notions of "sender" and "receiver", it is rather usual for a large class of systems to distinguish between the sending and

$$
\begin{aligned}
&\texttt{proc}\ A(t_1 : Timer \cdots, t_m : Timer, d_1 : D_1 \cdots, d_n : D_n) = \\
&\quad a_1.X_1(\vec{t}_1, \vec{y}_1) \triangleleft expired(t_1) \triangleright \delta + \\
&\quad \vdots \\
&\quad a_m.X_m(\vec{t}_m, \vec{y}_m) \triangleleft expired(t_m) \triangleright \delta + \\
&\quad tick.A(pred(\vec{t}), \vec{d}) \triangleleft not(\textstyle\bigvee_{j=1}^{n} expired(t_j)) \triangleright \delta + \\
&\quad \textstyle\sum_{d_{11}:D_{11}} \cdots \sum_{d_{1n_1}:d_{1n_1}} in_1(\vec{s}_1).X'_1(\vec{t}_1, \vec{x}_1) \triangleleft c_1 \triangleright \delta + \\
&\quad \vdots \\
&\quad \textstyle\sum_{d_{k1}:D_{k1}} \cdots \sum_{d_{kn_k}:D_{kn_k}} in_k(\vec{s}_k).X'_k(\vec{t}_k, \vec{x}_k) \triangleleft c_k \triangleright \delta
\end{aligned}
$$

Fig. 4. Pattern of an input state.

$$
\begin{array}{l}
\texttt{proc}\ B(t_1 : Timer \cdots, t_m : Timer, d_1 : D_1 \cdots, d_n : D_n) = \\
\qquad \sum_{d_{11}:D_{11}} \cdots \sum_{d_{1n_1}:D_{1n_1}} a_1(\vec{s}_1).X_1(\vec{t}, \vec{x}_1) \triangleleft c_1 \triangleright \delta + \\
\qquad \vdots \\
\qquad \sum_{d_{k1}:D_{k1}} \cdots \sum_{d_{kn_k}:D_{kn_k}} a_k(\vec{s}_k).X_n(\vec{t}, \vec{x}_n) \triangleleft c_k \triangleright \delta
\end{array}
$$

Fig. 5. Pattern of a non-input state.

the receiving party in the communication action. Moreover, it is logical to expect for a correct specification that sendings take place in the time slice when they become enabled; otherwise the communication cannot be seen as synchronous and should go via a channel. Input, or reception, can be postponed until the next time slice. Consequently, we allow `tick` as an alternative to an input action and not to an output action.

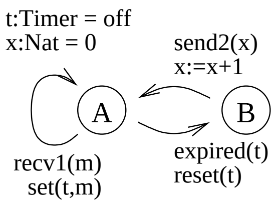
The classification of actions results in the classification of process states: We require every state to be either an *input state*, i.e. a state where only input actions can be enabled, or a *non-input state*, i.e. a state where outputs and internal actions can be taken. The check that the specification meets this requirement can be easily automated by introducing conventional names for input and output actions. To simplify matters further, we have used patterns for specifying states of components as μCRL processes.

The patterns of input and non-input states are given in Fig. 4 and 5, respectively. In these patterns, all μCRL processes which correspond to states in a component have the same list of parameters. For a component with m timers and n other variables, the first m parameters are timers and the next n are the other variables. Input and non-input states have different transitions: In an input state, we have timer expiration events for expired timers, `tick` if no timer is expired and receive actions. In non-input states, we have send actions and internal actions. After a `tick` action, all active timers are decreased and everything else remains the same. After a read or send action, timers may be set or reset, data parameters can be modified and the state may change.

When we build a system from components, we must not only make sure that time progression is handled correctly but also that all messages sent on the set of internal channels I are kept within the system. The first means using `|{tick}|`, the latter means encapsulation of the send and receive actions for the channels in I. That is, a system with N components and internal channels I is described by the following μCRL init statement:

$$\texttt{init encap}(\{s_i, r_i\}_{i \in I}, C_1 \mid \{tick\} \mid \cdots \mid \{tick\} \mid C_N)$$

In Fig. 6 we give a pictorial representation and μCRL code of a simple watchdog. On channel 1, the watchdog receives a delay. While within that delay a new delay is sent the watchdog keeps waiting. If the watchdog times out then a message will be sent on channel 2.




```
proc A(t:Timer,x:Nat)=
    expire.B(reset(t),x)<|expired(t)|>delta+
    tick.A(pred(t),x)<|not(expired(t))|>delta+
    sum(m:Nat,recv1(m).A(set(t,m),x)<|T|>delta)
proc B(t:Timer,x:Nat)=
    send2(x).A(t,S(x))<|T|>delta
init A(off,0)
```

Fig. 6. A simple component.

5 Experiments

We have tested our approach on a number of examples one of which was a positive acknowledgment retransmission protocol (PAR) [15]. This is a classical example of a communication protocol in which time issues are essential for correct functionality. The usual scenario includes a sender, a receiver, a message channel and an acknowledgment channel. The sender receives a frame from the upper layer, sends it to the receiver via the message channel and waits for a positive acknowledgment from the receiver via acknowledgment channel. When the receiver delivered the message to the upper layer it sends the acknowledgement to the sender. After the positive acknowledgment is received, the sender becomes ready to send next message. The receiver needs some time to deliver the received frame to the upper layer. The channels delay the delivery of messages as well. Moreover, they can lose or corrupt messages. Therefore, the sender handles lost frames by timing out. If the sender times out, it re-sends the message.

The possible erroneous scenario is following. The sender times out while the acknowledgement is still on the way. The sender sends a duplicate, then receives the acknowledgment and believes that this is the acknowledgment for the duplicate. The sender sends the next frame, which gets lost. The sender receives however the acknowledgment for the duplicate, which it believes to be the acknowledgement for the last frame. Thus the sender does not retransmit the lost message and the protocol fails. To avoid the erroneous behaviour, the timeout interval must be long enough to prevent a premature timeout, which means that the timeout interval should be larger than the sum of delays on the message channel, acknowledgment channel and receiver.

We have specified PAR in μCRL using timers to represent delays on the channels and the receiver and timeout for the sender. Since the system is open, i.e. both the sender and the receiver communicate with upper layers, we have closed the system by the environment process that provides frames for the sender and receives frames delivered by the receiver. If the sender is ready to send next frame before the environment gets the previous frame delivered by the receiver, the environment process issues an error action `err`. The `err` action also occurs if the environment gets a wrong (not sent to the sender) frame from the receiver.

Using the μCRL toolset we have generated the state space for the μCRL specification of the protocol. With the CADP toolset, we have verified then a number of properties expressed by formulas of regular alternation-free μ-

calculus [12]. One of the properties was absence of traces containing error action `err`: `P1: [T*."err"]F`, which held when the sender's timeout was large enough to avoid premature timeouts.

Another property, we have checked, was inevitable reachability of `__out` action after `__in` action meaning that the frame sent by the sender to receiver will always be delivered by the receiver to the environment:

```
P_2:[T*."__in"] "mu" X.(<T>T and [not("__out")]X).
```

This property held neither for the system with correct timeout intervals nor for the system with premature timeouts. This can be explained by the fact that we do not use fairness, and hence the message channel may continue lose or corrupt frames forever, so the frame will never be delivered to the environment.

Using the notion of weak fairness, we have specified the property stating fair reachability of `__out` action after `__in`:

```
P_3:[T*."__in".(not("__out"))*]<(not("__out"))*."__out">T.
```

The property P_3 held for the system with correct timeout intervals and not for the system with wrong ones.

6 Timed verification

In the previous sections we showed how to specify a timed system in μCRL and how to verify properties dependent on the settings of timers. In this section, we discuss how to verify *timed* properties. For this purpose, we introduce an LTL-like language that allows the direct use of timed constraints and then show how to encode those timed constraints with the use of `tick`.

6.1 Path restricted LTL

First, we will give an untimed version. This untimed version is in fact a restriction of the μ-calculus, which is the input language used by the CADP toolset. Hence, the untimed formulas can be verified with the CADP toolset.

Let S be a set of states, and Act be a set of labels (actions). A path π of length N is a pair of functions

$$(s_\pi : \{0, \ldots, N\} \to S,\ a_\pi : \{1, \ldots, N\} \to Act)$$

Thus, $s_\pi(i)$ stands for the i-state of the path and $a_\pi(i)$ for the i-action of the path. Note that N may be infinite. We write $\pi(i,k)$ to denote a subpath of π starting at state $s_\pi(i)$ and ending at state $s_\pi(k)$ for $i = 0 \ldots N$.

Let Φ be a set of *state* formulas defined as $\Phi = \{true, false\}$, where *true* holds in all the states and *false* holds in none of them.

Let $\mathcal{R}$ be a set of *action* formulas, where an action formula r is defined as follows:

$$r ::= action \mid any \mid none \mid r_1 \ and \ r_2 \mid r_1 \ or \ r_2 \mid \neg r$$

Here $action \in Act$ is a formula satisfied by the corresponding label only. Any label from Act satisfies *any* and none of them satisfies *none*. A label satisfies $\neg r$

iff it does not satisfy r, a label satisfies r_1 *and* r_2 iff it satisfies both r_1 and r_2, and a label satisfies r_1 *or* r_2 iff it satisfies r_1 or r_2.

Using action formulas one can build *path* formula p as follows:

$$p ::= nil \mid r \mid p_1.p_2 \mid p_1 + p_2 \mid p^* \mid p^+$$

Here nil is an empty operator, $p_1.p_2$ is the concatenation, $p_1 + p_2$ is a choice operator, p^* is the transitive reflexive closure and p^+ is the transitive closure.

Let $\mathcal{P}$ be a set of all path formulas. We write $\pi(i,k) \models_{\mathcal{P}} p$ if $a_\pi(i+1) \ldots a_\pi(k)$ string matches path expression p.

Further we define a path-restricted LTL, where LTL modalities are parameterized by path formulas.

Definition 1 (Syntax of path restricted LTL).

$$\phi ::= \varphi \mid \langle p \rangle \phi \mid [p]\phi \mid \phi U(p)\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi$$

where p stands for a path formula and φ for a state formula.

First we give an intuition for formulas of the path-restricted LTL and then we provide more formal semantics.

$\langle p \rangle \phi$ holds on a path π if there exists a prefix $\pi(0,i)$ of π that matches p and ϕ holds on the suffix of π starting at $s_\pi(i)$.

$[p]\phi$ holds on a path π if for every its prefix $\pi(0,i)$ that matches p, ϕ holds on the suffix of π starting at $s_\pi(i)$.

$\psi U(p)\phi$ holds on a path π if there exists a state $s_\pi(i)$ on the path such that the path up to this state matches p, the path starting at $s_\pi(i)$ satisfies ϕ and the path starting at any state before this state satisfies ψ.

Definition 2 (Semantics of path restricted LTL). *Let π, i be the suffix of π starting at $s_\pi(i)$, then:*

- $\pi, i \models \varphi$ *where* $\varphi \in \Phi$ *if* $s_\pi(i) \models \varphi$;
- $\pi, i \models \langle p \rangle \phi$ *if there exists some* $k \geq i$ *such that* $\pi(i,k) \models_{\mathcal{P}} p$ *and* $\pi, k \models \phi$;
- $\pi, i \models [p]\phi$ *if for any* $k \geq i$ *such that* $\pi(i,k) \models_{\mathcal{P}} p$ *we have* $\pi, k \models \phi$;
- $\pi, i \models \psi U(p)\phi$ *if there exists some* $k \geq i$ *such that* $\pi(i,k) \models_{\mathcal{P}} p$ *and* $\pi, k \models \phi$ *and for any* $j : i \leq j < k \quad \pi, j \models \psi$ *holds.*

We say that π *satisfies* ϕ, denoted as $\pi \models \phi$, if $\pi, 0 \models \phi$. Formula ϕ is *satisfied* by an LTS T if all paths of T starting at the initial state satisfy the formula.

6.2 Path restricted LTL with time

Now we extend the path restricted LTL with time constraints of the form:

$$tc ::= \leq c \mid = c \mid \geq c$$

Let $d(\pi(i,k))$ denote the number of `tick` steps in $\pi(i,k)$. Then:

– $\pi(i,k) \models \leq c$ if $d(\pi(i,k)) \leq c$;
– $\pi(i,k) \models \geq c$ if $d(\pi(i,k)) \geq c$;
– $\pi(i,k) \models = c$ if $d(\pi(i,k)) = c$.

Then a path restricted LTL formula with time is defined as follows:

$$\phi ::= \varphi \mid \langle p\rangle_{tc}\phi \mid [p]_{tc}\phi \mid \phi U(p)_{tc}\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \ ,$$

where none of the path formulas p refer to the action `tick`.

The intuitive semantics of the formulas is similar to those of path-restricted LTL. $\langle p\rangle_{tc}\phi$ holds on a path if there exists a state on that path such that the path up to that state satisfies both p and the time constraint and ϕ is satisfied by the path starting at that point.

$[p]_{tc}\phi$ holds on a path if for every prefix of the path that matches both p and the time constraint, ϕ holds the corresponding suffix of the path.

$\psi U(p)_{tc}\phi$ holds on a path if there exists a state on the path such that the path up to that state matches both p and tc, the path starting at that state satisfies ϕ and the path starting at any state before satisfies ψ.

The intuition about path formulas is that they hold on traces regardless of time progression. This means that a timed path satisfies a path formula if the path with the `tick` steps removed satisfies the path formula. Formally, we have

$$\pi(i,k) \models_{\mathcal{P}}^{tick} p \text{ if } a'_\pi(i+1)\ldots a'_\pi(k) \text{ matches } p, \text{ where } a' = \begin{cases} \epsilon \ , \text{if } a = \texttt{tick} \\ a \ , \text{otherwise} \end{cases}$$

Definition 3 (Semantics of path restricted LTL with time). *Let π, i is a suffix of π starting at $s_\pi(i)$, then:*

– $\pi, i \models^{tick} \varphi$ *where* $\varphi \in \Phi$ *if* $s_\pi(i) \models \varphi$;
– $\pi, i \models^{tick} \langle p\rangle_{tc}\phi$ *if there exists some* $k \geq i$ *such that* $\pi(i,k) \models_{\mathcal{P}}^{tick} p$ *and* $\pi(i,k) \models tc$ *and* $\pi, k \models^{tick} \phi$;
– $\pi, i \models^{tick} [p]_{tc}\phi$ *if for any* $k \geq i$ *such that* $\pi(i,k) \models_{\mathcal{P}}^{tick} p$ *and* $\pi(i,k) \models tc$ *we have* $\pi, k \models^{tick} \phi$;
– $\pi, i \models^{tick} \psi U(p)_{tc}\phi$ *if there exists some* $k \geq i$ *such that* $\pi(i,k) \models_{\mathcal{P}}^{tick} p$ *and* $\pi(i,k) \models tc$ *and* $\pi, k \models^{tick} \phi$ *and for any* $j : i \leq j < k \quad \pi, j \models^{tick} \psi$ *holds;*

We say that π *satisfies* ϕ denoted $\pi \models^{tick} \phi$ if $\pi, 0 \models^{tick} \phi$. Formula ϕ is *satisfied* by an LTS T if all paths of T starting at the initial state satisfy the formula.

Example 1: each *request* is followed by *answer* in at most 5 time units:

$$[any^*.request] < any^*.given >_{\leq 5} true$$

Example 2: "request" is never followed by "fail" within 2 time units.

$$[any^*.request][any^*.fail]_{\leq 2} false$$

6.3 `tick`-encoding of path restricted LTL with time

In this section we present a construction for translating a formula from path restricted LTL with time into path restricted LTL with `tick`. The key to this translation is the construction of a path formula over an action domain with `tick` from a path formula over a domain without `tick` but with a time constraint. This is done by translating both the path formula and the time constraint into a finite automaton, combining these automata into a single automaton and translating this automaton back into a path formula.

It is common knowledge that regular expression and finite automata have the same expressive power and can be translated into each other. Let $RE(\mathcal{A})$ be the translation from an automata $\mathcal{A}$ to an equivalent regular expression and let $\mathcal{A}(RE)$ be the transformation of a regular expression RE into a finite automaton.

Next, we will give the translation of time constraints into finite automata. But first, we give the formal definition of finite automata and languages recognized by finite automata.

Definition 4. *A finite automaton is a tuple* $\mathcal{A} \equiv (S, \Sigma, T, I, F)$*, where*

- S *is a set of states;*
- Σ *is a set of labels;*
- $T \subseteq S \times \Sigma \times S$ *is a set of transitions;*
- $I \subseteq S$ *is a set of initial states;*
- $F \subseteq S$ *is a set of final states.*

The set of strings recognized by $\mathcal{A}$ *is given by*

$$L(\mathcal{A}) = \{a_1 \dots a_n \mid s_0 \in I, s_n \in F, \forall j = 1..n : (s_{j-1}, a_j, s_j) \in T\}$$

Definition 5. *The automata recognizing time constraints are:*

$\mathcal{A}(\leq c) =$
$(\{0, 1, \dots, c+1\}, \{\texttt{tick}\}, \{(i, \texttt{tick}, i+1) \mid i = 0 \dots c\}, \{0\}, \{0, 1, \dots, c\})$
$\mathcal{A}(= c) =$
$(\{0, 1, \dots, c+1\}, \{\texttt{tick}\}, \{(i, \texttt{tick}, i+1) \mid i = 0 \dots c\}, \{0\}, \{c\})$
$\mathcal{A}(\geq c) =$
$(\{0, 1, \dots, c\}, \{\texttt{tick}\}, \{(i, \texttt{tick}, i+1), (c, \texttt{tick}, c) \mid i = 0 \dots c-1\}, \{0\}, \{c\})$

We now have a finite automaton corresponding to the path formula and a finite automaton corresponding to the time constraint. All we need to do is to build the product automata, which will recognize all interleavings of strings recognized by these two automata. The following definition gives such a construction:

Definition 6. *Given two finite automata* $\mathcal{A}_1 \equiv (S_1, \Sigma_1, T_1, I_1, F_1)$ *and* $\mathcal{A}_2 \equiv (S_2, \Sigma_2, T_2, I_2, F_2)$*, we define*

$$\mathcal{A}_1 \times \mathcal{A}_2 = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, T_1 \times S_2 \cup S_1 \times T_2, I_1 \times I_2, F_1 \times F_2), \text{ where}$$

$$T_1 \times S_2 = \{((s_1, s_2), a, (t_1, s_2)) \mid (s_1, a, t_1) \in T_1 \wedge s_2 \in S_2\} \ ;$$
$$S_1 \times T_2 = \{((s_1, s_2), a, (s_1, t_2)) \mid s_1 \in S_1 \wedge (s_2, a, t_2) \in T_2\} \ .$$

We can now define the translation of path restricted LTL with time to path restricted LTL with `tick`.

Definition 7. *The function* $[\![\cdot]\!]$ *translating path restricted LTL with time to path restricted LTL with* `tick` *is given by:*

$$\begin{array}{ll} [\![\varphi]\!] & = \varphi \\ [\![\langle p\rangle_{tc}\psi]\!] & = \langle p \times tc\rangle [\![\psi]\!] \\ [\![[p]_{tc}\psi]\!] & = [p \times tc][\![\psi]\!] \\ [\![\psi_1\, U(p)_{tc}\, \psi_2]\!] & = [\![\psi_1]\!]\, U(p \times tc)[\![\psi_2]\!] \\ [\![\psi_1 \wedge \psi_2]\!] & = [\![\psi_1]\!] \wedge [\![\psi_2]\!] \\ [\![\psi_1 \vee \psi_2]\!] & = [\![\psi_1]\!] \vee [\![\psi_2]\!] \\ [\![\neg\psi]\!] & = \neg [\![\psi]\!] \end{array}$$

where

$$p \times tc = RE(A(p) \times A(tc)) \ .$$

This translation preserves satisfaction:

Proposition 8. *For any LTS* L *and any path restricted LTL with time formula* ψ*, we have*

$$L \models^{tick} \psi \Longleftrightarrow L \models [\![\psi]\!] \ .$$

7 Conclusion

In this paper we proposed an approach to specification and verification of timed systems within the untimed μCRL framework. The experimental results confirmed the usefulness of the approach.

Related works Timed process algebras can be classified using three criteria. First, whether they use dense time or discrete time. Second, whether they use absolute or relative time. Third, whether they use time progression constructs or time stamping of actions. For example, timed μCRL [8] uses absolute time, time stamping of actions and leaves the choice between dense and discrete time open. Several versions of ACP with time have been studied. (E.g. [3, 2].) These algebras use an operator σ to express time progression rather than an action. For example, the process $\sigma(P)$ in ACP with discrete relative time ($\mathrm{ACP}_{\mathrm{drt}}$ [2]) is intuitively the same as the process `tick`.P in μCRL with the `tick`-convention. For theoretical work the σ operator is more convenient. For tool support the `tick` action is easier. Hence in ACP one uses σ and in μCRL we use `tick`.

The use of the `tick` action results in a time semantics which is similar to the semantics used in others tools, such as DT Spin [6] and ObjectGeode [14]. However, the input languages of those tools restrict to one particular message passing model and in μCRL we are free to use whatever model we want. Moreover, Spin restricts to LTL model checking while in μCRL we can use regular alternation free μ-calculus.

Future work It will be interesting to find out if the framework presented in this paper can be extended to provide tool support for timed μCRL. Namely, we are going to investigate what class of specifications in timed μCRL can be adequately translated to the class of specifications described in the paper. Another research topic is the development of time-specific optimisation techniques, such as a `tick`-confluence based partial order method.

References

1. R. Alur. Timed Automata. In *Proceedings of CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer-Verlag, 1999.
2. J. C. M. Baeten and J. A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.
3. J. C. M. Baeten and C. A. Middelburg. Process Algebra with Timing: Real Time and Discrete Time. In Bergstra et al. [4].
4. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.
5. S. C. C. Blom, W. J. Fokkink, J. F. Groote, I. A. van Langevelde, B. Lisser, and J. C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings 13th Conference on Computer Aided Verification (CAV'01), Paris, France*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.
6. D. Bošnački and D. Dams. Integrating real time into Spin: A prototype implementation. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Proceedings of Formal Description Techniques and Protocol Specification, Testing, and Verification*. Kluwer Academic Publishers, 1998.
7. M. Bozga, O. Maler, and S. Tripakis. Efficient verification of timed automata using dense and disrete time semantics. In T. Kropf and L. Pierre, editors, *Proc. CHARME'99*, volume 1703 of *Lecture Notes in Computer Science*, pages 125–141. Springer, September 1999.
8. J. F. Groote. The syntax and semantics of timed μCRL. SEN R9709, CWI, Amsterdam, 1997.
9. J. F. Groote and M. Reniers. Algebraic process verification. In Bergstra et al. [4], pages 1151–1208.
10. J. F. Groote and J. J. van Wamel. Analysis of three hybrid systems in timed μCRL. *Science of Computer Programming*, 39:215–247, 2001.
11. T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In W. Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer, 1992.
12. R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular aletrnation-free mu-calculus. In *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems, FMICS'2000*, 2000.
13. R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM(JACM)*, 42(2):458–487, 1996.
14. ObjectGeode 4. `http://www.csverilog.com/products/geode.htm`, 2000.
15. A. S. Tanenbaum. *Computer Networks*. Prentice Hall International, Inc., 1981.