Time-critical reactive systems (II)

Luís S. Barbosa

DI-CCTC
Universidade do Minho
Braga, Portugal

May, 2010

Traces

Definition

A timed trace over a temporal LTS is a (finite or infinite) sequence $\langle t_1, a_1\rangle, \langle t_2, a_2\rangle, \cdots$ in $\mathbb{R}^+ \times Act$ such that there exists a path

$$\cdots \xleftarrow{a_2} \langle l_1, \eta_3\rangle \xleftarrow{d_2} \langle l_1, \eta_2\rangle \xleftarrow{a_1} \langle l_0, \eta_1\rangle \xleftarrow{d_1} \langle l_0, \eta_0\rangle$$

such that

$$t_i = t_{i-1} + d_i$$

with $t_0 = 0$ and, for all clock x, $\eta_0\, x = 0$.

Intuitively, each t_i is an absolute time value acting as a time-stamp.

Warning

All results from now on are given over an arbitrary temporal LTS; they naturally apply to $\mathcal{T}(ta)$ for any timed automata ta.

Traces

Definition

A timed trace over a temporal LTS is a (finite or infinite) sequence $\langle t_1, a_1\rangle, \langle t_2, a_2\rangle, \cdots$ in $\mathbb{R}^+ \times Act$ such that there exists a path

$$\cdots \xleftarrow{a_2} \langle l_1, \eta_3\rangle \xleftarrow{d_2} \langle l_1, \eta_2\rangle \xleftarrow{a_1} \langle l_0, \eta_1\rangle \xleftarrow{d_1} \langle l_0, \eta_0\rangle$$

such that

$$t_i = t_{i-1} + d_i$$

with $t_0 = 0$ and, for all clock x, $\eta_0\, x = 0$.

Intuitively, each t_i is an absolute time value acting as a time-stamp.

Warning

All results from now on are given over an arbitrary temporal LTS; they naturally apply to $\mathcal{T}(ta)$ for any timed automata ta.

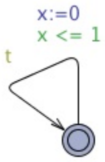
Traces

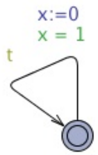
Given a timed trace tc, the corresponding untimed trace is $(\pi_2)^\omega\ tc$.

Definition

- two states s_1 and s_2 of a timed LTS are timed-language equivalent if the set of finite timed traces of s_1 and s_2 coincide;
- ... similar definition for untimed-language equivalent ...

Example





are not timed-language equivalent: $\langle(0, t)\rangle$ is not a trace of the TLTS generated by the second system.

Bisimulation

Timed bisimulation

A relation R is a timed simulation iff whenever $s_1 R s_2$, for any action a and delay d,

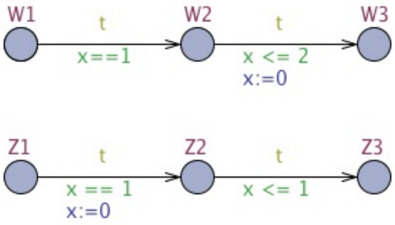
$$s_1' \xleftarrow{a} s_1 \;\Rightarrow\; \text{there is a transition} \quad s_2' \xleftarrow{a} s_2 \wedge s_1' R s_2'$$

$$s_1' \xleftarrow{d} s_1 \;\Rightarrow\; \text{there is a transition} \quad s_2' \xleftarrow{d} s_2 \wedge s_1' R s_2'$$

And a timed bisimulation if its converse is also a bisimulation.

Bisimulation

Example



$$\langle\langle W1, [x = 0]\rangle, \langle Z1, [x = 0]\rangle\rangle \in R$$

where

$$\begin{aligned} R \; = \; & \{\langle\langle W1, [x = d]\rangle, \langle Z1, [x = d]\rangle\rangle \mid d \in \mathbb{R}_0^+\} \; \cup \\ & \{\langle\langle W2, [x = d + 1]\rangle, \langle Z2, [x = d]\rangle\rangle \mid d \in \mathbb{R}_0^+\} \; \cup \\ & \{\langle\langle W3, [x = d]\rangle, \langle Z3, [x = e]\rangle\rangle \mid d, e \in \mathbb{R}_0^+\} \end{aligned}$$

Bisimulation

Untimed bisimulation

A relation R is a untimed simulation iff whenever $s_1 R s_2$, for any action a and delay t,

$$s_1' \xleftarrow{a} s_1 \Rightarrow \text{there is a transition} \quad s_2' \xleftarrow{a} s_2 \wedge s_1' R s_2'$$

$$s_1' \xleftarrow{d} s_1 \Rightarrow \text{there is a transition} \quad s_2' \xleftarrow{d'} s_2 \wedge s_1' R s_2'$$

And a untimed bisimulation if its converse is also a untimed bisimulation.

Alternatively, it can be defined over a modified LTS in which all delays are abstracted on a unique, special transition labelled by ϵ.

Properties: expression and satisfaction

The satisfaction problem

Given a timed automata, *ta*, and a property, ϕ, show that

$$\mathcal{T}(ta) \models \phi$$

- in which logic language shall ϕ be specified?
- how is $\models$ defined?

Properties: expression and satisfaction

The satisfaction problem

Given a timed automata, *ta*, and a property, ϕ, show that

$$\mathcal{T}(ta) \models \phi$$

- in which logic language shall ϕ be specified?
- how is $\models$ defined?

Expressing properties: UPPAAL

UPPAAL variant of CTL

- state formulae: describes individual states in $\mathcal{T}(ta)$
- path formulae: describes properties of paths in $\mathcal{T}(ta)$

Expressing properties: UPPAAL

State formulae

Any expression which can be evaluated to a boolean value for a state (typically involving the clock constraints used for guards and invariants and similar constraints over integer variables):

`x >= 8`, `i == 8 and x < 2`, ...

Additionally,

- *ta*.`l` which tests current location: $(l, \eta) \models ta.\mathtt{l}$ provided (l, η) is a state in $\mathcal{T}(ta)$
- `deadlock`: $(l, \eta) \models \forall_{d \in \mathbb{R}_0^+}$. there is no transition from $\langle l, \eta + d \rangle$

Expressing properties: UPPAAL

Path formulae

$$\Pi \ ::= A\Box\,\Psi \mid A\Diamond\,\Psi \mid E\Box\,\Psi \mid E\Diamond\,\Psi \mid \Phi \leadsto \Psi$$

where

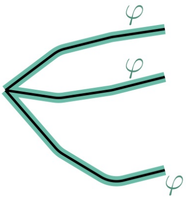
- A, E quantify (universally and existentially, resp.) over paths
- $\Box$, $\Diamond$ quantify (universally and existentially, resp.) over states in a path

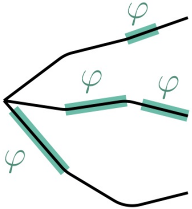
also notice that

$$\Phi \leadsto \Psi \ \stackrel{\text{abv}}{=}\ A\Box\,(\Phi \Rightarrow E\Diamond\,\Psi)$$

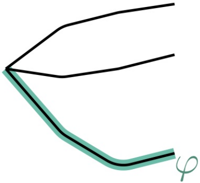
Expressing properties: Uppaal

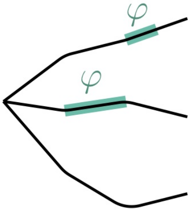
$A\Box\,\varphi$ and $A\Diamond\,\varphi$





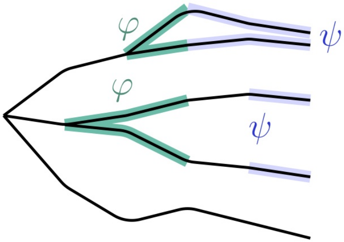
$E\Box\,\varphi$ and $E\Diamond\,\varphi$





Expressing properties: UPPAAL

$\varphi \rightsquigarrow \psi$



Reachability properties

$E \Diamond \phi$

Is there a path starting at the initial state, such that a state formula ϕ is eventually satisfied?

- Often used to perform sanity checks on a model:
  - is it possible for a sender to send a message?
  - can a message possibly be received?
  - ...
- Do not by themselves guarantee the correctness of the protocol (i.e. that any message is eventually delivered), but they validate the basic behavior of the model.

Safety properties

$A\Box\,\phi$ and $E\Box\,\phi$

Something bad will never happen
or something bad will possibly never happen

Examples

- In a nuclear power plant the temperature of the core is always (invariantly) under a certain threshold.
- In a game a safe state is one in which we can still win, ie, will possibly not loose.

In Uppaal these properties are formulated positively: something good is invariantly true.

Liveness properties

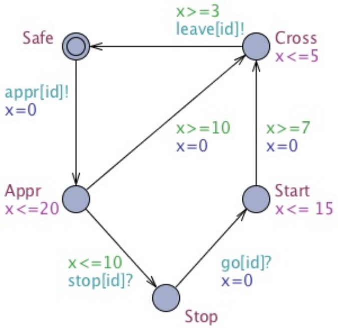
$A\Diamond\,\phi$ and $\phi \rightsquigarrow \psi$

Something good will eventually happen
or if something good happen, then something else will eventually happen!

Examples

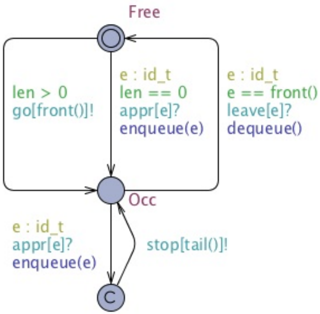
- When pressing the on button, then eventually the television should turn on.
- n a communication protocol, any message that has been sent should eventually be received.

The train gate example



- Events model approach/leave, order to stop/go
- A train can not be stopped or restart instantly
- After approaching it has 10m to receive a stop.
- After that it takes further 10 time units to reach the bridge
- After restarting takes 7 to 15m to reach the cross and 3-5 to cross

The train gate example



- Note the use of parameters and the select clause on transitions
- Programming ...

Demo

- The train gate case study (included in the UPPAAL distribution).

Mutual exclusion

Properties

- mutual exclusion: no two processes are in their critical sections at the same time
- deadlock freedom: if some process is trying to access its critical section, then eventually some process (not necessarily the same) will be in its critical section; similarly for exiting the critical section

Mutual exclusion

The Problem

- Dijkstra's original asynchronous algorithm (1965) requires, for n processes to be controlled, $\mathcal{O}(n)$ read-write registers and $\mathcal{O}(n)$ operations.
- This result is a theoretical limit (proved by Lynch and Shavit in 1992) which compromises scalability.

but it can be overcome by introducing specific timing constraints

Two *timed* algorithms:

- Fisher's protocol (included in the UPPAAL distribution)
- Lamport's protocol

Mutual exclusion

The Problem

- Dijkstra's original asynchronous algorithm (1965) requires, for n processes to be controlled, $\mathcal{O}(n)$ read-write registers and $\mathcal{O}(n)$ operations.
- This result is a theoretical limit (proved by Lynch and Shavit in 1992) which compromises scalability.

but it can be overcome by introducing specific timing constraints

Two *timed* algorithms:

- Fisher's protocol (included in the UPPAAL distribution)
- Lamport's protocol

Fisher's algorithm

The algorithm

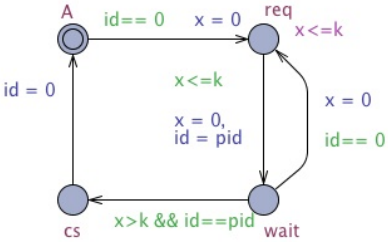
repeat
    repeat
        await $id = 0$
        $id := i$
        delay(k)
    until $id = i$
    (critical section)
    $id := 0$
forever

Fisher's algorithm

Comments

- One shared read/write register (the variable *id*)
- Behaviour depends crucially on the value for k — the time delay
- Constant k should be larger than the longest time that a process may take to perform a step while trying to get access to its critical section
- This choice guarantees that whenever process i finds $id = i$ on testing the loop guard it can enter safely ist critical section: all other processes are out of the loop or with their index in *id* overwritten by i.

Fisher's algorithm in UPPAAL



- Each process uses a local clock *x* to guarantee that the upper bound between between its successive steps, while trying to access the critical section, is *k* (cf. invariant in state *req*).
- Invariant in state *req* establishes *k* as such an upper bound
- Guard in transition from *wait* to *cs* ensures the correct delay before entering the critical section

Fisher's algorithm in UPPAAL

Properties

```
A[] forall (i:id_t) forall (j:id_t) P(i).cs && P(j).cs imply i == j
A[] not deadlock
P(1).req --> P(1).wait
```

- The algorithm is deadlock-free
- It ensures mutual exclusion if the correct timing constraints.
- ... but it is critically sensible to small violations of such constraints: for example, replacing $x > k$ by $x \geq k$ in the transition leading to *cs* compromises both mutual exclusion and liveness.

Lamport's algorithm

The algorithm

start : $a := i$

if $b \neq 0$ then goto start

$b := i$

if $a \neq i$ then delay(k)

else if $b \neq i$ then goto start

(critical section)

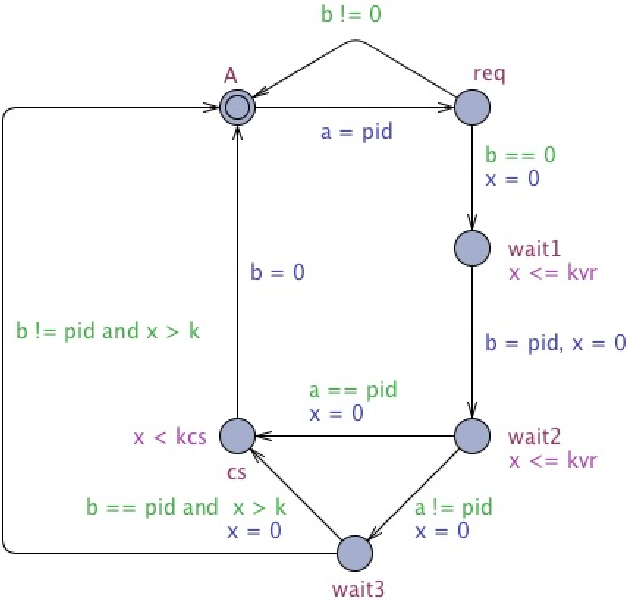
$b := 0$

Lamport's algorithm

Comments

- Two shared read/write registers (variables a and b)
- Avoids forced waiting when no other processes are requiring access to their critical sections

Lamport's algorithm in UPPAAL



Lamport's algorithm

Model time constants:

k — time delay

kvr — max bound for register access

kcs — max bound for permanence in critical section

Typically

$$k \geq kvr + kcs$$

Experiments

	k	kvr	kcs	verified?
Mutual Exclusion	4	1	1	Yes
Mutual Exclusion	2	1	1	Yes
Mutual Exclusion	1	1	1	No
No deadlock	4	1	1	Yes
No deadlock	2	1	1	Yes
No deadlock	1	1	1	Yes

Reading suggestions

7

A Fast Mutual Exclusion Algorithm

Leslie Lamport

November 14, 1985, Revised October 31, 1986

Reading suggestions

Distrib Comput (1996) 10: 1–10

DISTRIBUTED COMPUTING

Fast timing-based algorithms

Rajeev Alur[1], **Gadi Taubenfeld**[2]

[1] Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA. e-mail: alur@bell-labs.com
[2] The Open University, 16 Klausner Street, P.O. Box 39328, Tel-Aviv 61392, Israel, and AT&T Bell Laboratories. e-mail:gadi@research.att.com

Received: July 1993/Accepted: February 1996

Summary. Concurrent systems in which there is a known upper bound Δ on memory access time are considered. Two prototypical synchronization problems, mutual exclusion and consensus, are studied, and solutions that have constant (i.e. independent of Δ and the total number of processes) time complexity in the absence of contention are presented. For mutual exclusion, in the absence of contention, a process needs only five accesses to the shared memory to enter its critical section, and in the presence of contention, the winning process may need to delay itself for $4 \cdot \Delta$ time units. For consensus, in absence of contention, a process decides after four accesses to the shared memory, and in the presence of contention, it may need to delay itself for Δ time units.

Key words: Shared–memory algorithms – Mutual exclusion – Consensus – Timing–based model – Contention–free complexity

tive that enables us to design efficient algorithms. We refer to our model as the *known-delay* model.

To measure the time complexity of an algorithm in the known-delay model, we account for the *step complexity* that measures the number of times a process accesses shared registers, along with the *explicit-delay complexity* that is the sum of the explicit delays executed using the *delay* statement. Apart from the usual worst case complexity that indicates the maximum time it takes a process to attain its goal, we will also be interested in the *contention-free* complexity, which gives an upper bound on the time required for a process to attain its goal, when the process runs by itself without any interference from other processes. Since contention should be rare in well-designed systems, it is important to design algorithms that perform well also in the absence of contention. This was first pointed out in [11] where a mutual exclusion algorithm is presented, in which a process accesses shared registers only a constant number of times to enter its critical section in the absence of contention. A *fast* algo-