Timed I/O Automata

Pedro Ferreira do Souto

Departamento de Engenharia Informática
Faculdade de Engenharia
Universidade do Porto

Outline

Outline

Outline

Outline

Models of Computation

Synchronous model

Processes repeatedly execute rounds in lock-step. In each round, they:

1. Use their current state to generate messages to send to neighbors, and put them in the appropriate channels.
2. Compute the new state from the current state and the incoming messages, and remove all the messsages from the channels.

Asynchronous model

Makes no assumptions regarding the timing behavior of system components

1. Processes may take an arbitrary time to execute the actions prescribed by the algorithms.
2. Channels may take an arbitrary time to deliver messages that are sent through them.

Time and Models of Computation

- Both models abstract away the time.
- Time is sometimes used for the analysis of the (time) complexity.
  - But it is not part of the model itself.
- In practice, most systems use time, at least in the form of timeouts.
- Increasingly, systems interact with the real world, which sometimes imposes timing requirements. Correctness depends:
  - Not only on the outputs generated by the system, or even their order;
  - But also on the time at which these outputs are generated.
    - ★ For some systems, being late is at least as bad as an omission fault.

Outline

Timed I/O Automata

- There are several variants of Timed I/O Automata
  - They are all based on standard I/O automata
  - They all include extensions to reason about timing properties
- We consider the variant described in the first reference
  - It is a simplification of Hybrid I/O Automata
  - It relies only on a few concepts that do not appear in the standard I/O automata

    Dynamic Type of a Variable
    Trajectory

Timed Input/Output Automata (TIOA)

- A timed (I/O) automaton is a state machine whose states are the valuations of its **variables**
  - Variables are internal to an automaton
- The state of a timed automaton may change
  - instantatenously, by the occurrence of **discrete transitions**
  - over an interval of time via **trajectories**, which are functions that describe the evolution of the state variables with time
- The discrete transitions are labeled with **actions**, which may be one of **input**, **output** or **internal** actions
  - Input and output actions are used for communication with the automaton's enviroment
    - ★ Internal actions are not visible externally
  - Output and internal actions are under the automaton control
    - ★ But input actions are not
- Communication of a TIOA with its enviroment is limited to discrete transitions

Outline

Dynamic Type of a Variable

- Each variable in a IO automaton has a (static) type, which specifies the values it may take
- In the case of a TIOA, every variable has also a **dynamic type**, which specifies how its value may evolve over time
- We consider essentially two types:

  Discrete The value changes only at discrete points in time, remaining constant between those points
  - The values of discrete variables change only upon occurrence of transitions
  - All variables in standard I/O automata are discrete

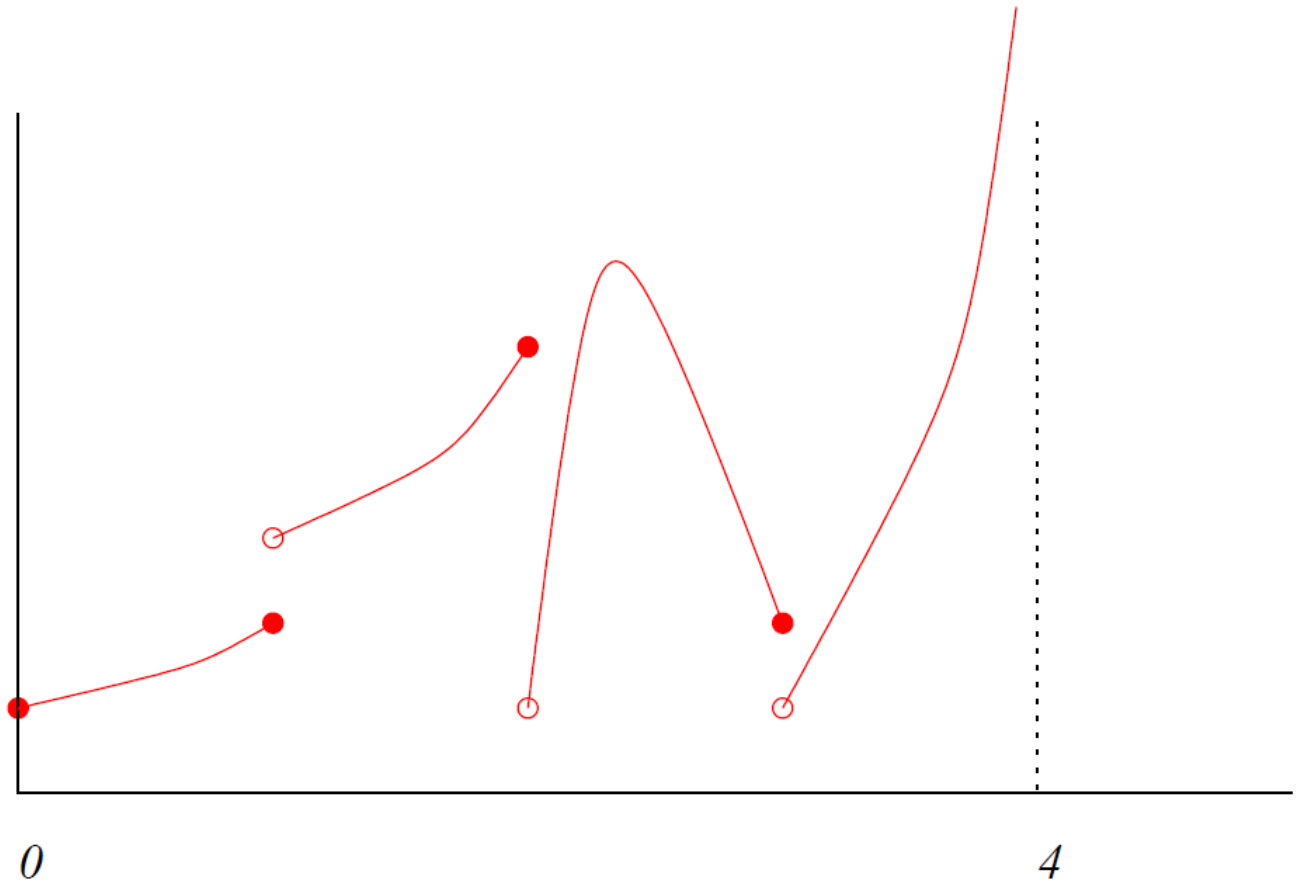
  Analog The value may change continuously over a time interval
  - This type is particularly useful to model timers/clocks, i.e. variables that measure the passage of time

Trajectory

- A **trajectory**, τ, describes the evolution of a set of variables over an interval of time, τ's domain, which:
  - Always starts at 0
  - May not be right closed
- Trajectories can be **concatenated**, using a concatenation operation $\frown$. The result is a trajectory:
  - Over a time interval whose duration is the sum of the time intervals of each of the trajectories
  - Obtained by **time-shifting** by the necessary amount of time each of the operand trajectories
- Given a trajectory, ν, we can define a **prefix** trajectory τ, by **restricting** ν to a time interval starting at 0 that is a subset of ν's domain

$$\tau \leq \nu$$

Trajectory Concatenation



src: Kaynar et. al. 2005

- The last valuation of a trajectory, which may not agree with the first valuation of the following operand trajectory, is the one that appears in the concatenation

Outline

Timed I/O Automata: Formal Definition

A **timed I/O automaton** (TIOA) $\mathcal{A} = (X, Q, \Theta, I, 0, H, \mathcal{D}, \mathcal{T})$ consists of:

- A set X of **internal variables**
- A set $Q \subseteq val(X)$ of **states**
- A nonempty set $\Theta \subseteq Q$ of **start states**
- A set I of **input actions**, a set O of **output actions** and a set H of **internal actions**, disjoint from each other. We write:

$$E \triangleq I \cup O \qquad \text{the set of external actions}$$

$$A \triangleq E \cup H \qquad \text{the set of all actions}$$

$$L \triangleq O \cup H \qquad \text{the set of locally controlled actions}$$

- A set $\mathcal{D} \subseteq Q \times A \times Q$ of **discrete transitions**
  - We write $x \xrightarrow{a} x'$ as a shorthand of $(x, a, x') \in \mathcal{D}$
  - We say that a is **enabled** in x if $x \xrightarrow{a} x'$ for some x'
  - We say that a set C of actions is enabled in a state x if some action in C is enabled in x
- A set $\mathcal{T} \subseteq trajs(Q)$ of trajectories, which must satisfy a set of axioms.

Timed I/O Automata: Trajectories (1/2)

- Given a trajectory $\tau \in \mathcal{T}$, we denote
  - $\tau.fstate$ the value of the state variables at time 0
  - $\tau.lstate$ the last value of the state variables, if τ is closed
  - When $\tau.fstate = x$ and $\tau.lstate = x'$, we write $x \xrightarrow{\tau} x'$
- The set of trajectories $\mathcal{T}$ of **timed automaton** (TA) must satisfy the following axioms:

  T0 Existence of point trajectories

  If $x \in Q$ then $\wp(x) \in \mathcal{T}$

  T1 Prefix closure

  For every $\tau \in \mathcal{T}$ and every $\tau' \leq \tau, \tau' \in \mathcal{T}$

  T2 Suffix closure

  For every $\tau \in \mathcal{T}$ and every $t \in dom(\tau), \tau \trianglerighteq t \in \mathcal{T}$

  T3 Concatenation closure

  Let $\tau_0\tau_1\tau_2 \ldots$ be a sequence of trajectories in $\mathcal{T}$ such that for each nonfinal index i, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$.

  Then $\tau_0 \frown \tau_1 \frown \tau_2 \ldots \in \mathcal{T}$

Timed I/O Automata: Trajectories (2/2)

- Axioms **T1, T2** are needed for the parallel composition operation for TA
  - In a composed system, any trajectory of any component automaton may be interrupted at any time by a discrete transition of another (possibly independent) automaton
    - ★ Axiom **T1** ensures that the part of the trajectory up to the discrete transition is a trajectory
    - ★ Axiom **T2** ensures that the remainder is a trajectory
- Axiom **T3** is required because the environment of a timed automaton may change its dynamics repeatedly, and the automaton must be able to follow this behavior.

Timed I/O Automata: Axioms

- The following axioms are satisfied:

  E1: Input action enabling

  For every $\mathbf{x} \in Q$ and every $a \in I$, there exists $\mathbf{x}' \in Q$ such that $\mathbf{x} \xrightarrow{\mathbf{a}} \mathbf{x}'$

  - I.e., a TIOA is able to perform every input action at any time
  - Standard IOA must also satisfy this axiom

  E2: Time-passage enabling

  For every $\mathbf{x} \in Q$, there exists $\tau \in \mathcal{T}$ such that $\tau.fstate = \mathbf{x}$ and either:

  1. $\tau.ltime = \infty$ or
  2. τ is closed and some $l \in L$ is enabled in $\tau.lstate$

  - I.e., a TIOA **either** allows time to advance forever **or** only up to a point where it is able to perform some locally controlled action

Outline

Timed I/O Automata Specification

Based on a TIOA language, in which a specification consists of four main parts:

Signature lists the actions along with their kinds (input, output or internal) and parameter types

State variables list declares the names and types of state variables. The dynamic type is defined implicitly

- Variables of type **Real** are analog, and all other variables are discrete

Collection of transition definitions defined in **precondition-effect** style

Trajectories definition

It differs from the IO Automata specification in that instead of a tasks definition we have trajectories definition

TIOA Example 4.1: A Time-Bounded Channel (1/6)

- The channel is reliable, i.e. does not drop messages
- The channel is FIFO, i.e. delivers the messages in the order they are sent
- Furthermore, is is **time-bounded**, i.e. it delivers the messages within a certain time bound (b) from being sent

TIOA Example 4.1: A Time-Bounded Channel (2/6)

```
automaton TimedChannel(b: Real, M: Type)
type Packet = tuple of message: M, deadline: Real
    signature
        input send(m: M)
        output receive(m: M)
    states
        queue: Seq[Packet] := {},
        now: Real := 0
        initially b ≥ 0
    transitions
        input send(m)
            eff
                queue := append([m,now+b], queue)
        output receive(m)
            pre
                head(queue).message = m
            eff
                queue := tail(queue)
    trajectories
        stop when
            ∃p: Packet p ∈ queue ∧ (now = p.deadline)
        evolve
            d(now) = 1
```

TIOA Example 4.1: A Time-Bounded Channel (3/6)

```
automaton TimedChannel(b: Real, M: Type)
type Packet = tuple of message: M, deadline: Real
```

- The `TimedChannel` automaton has two parameters:
  - `b` the bound on the time to deliver a message
  - `M` is the type of messages communicated by the channel
- The line:

  ```
  type Packet = tuple of message: M, deadline: Real
  ```

  defines the type `packet`, which:
  - Associates a message with its delivery deadline
  - Is used in the definition of variable `queue`

Signature specifies actions (this TIOA has no internal actions)

```
signature
    input send(m: M)
    output receive(m: M)
```

both of which take as a parameter the message being sent/received

TIOA Example 4.1: A Time-Bounded Channel (4/6)

State Comprises two variables:

```
states
  queue: Seq[Packet] := {},
  now: Real := 0
  initially b ≥ 0
```

`queue` is a queue with the packets in transit, it uses the built-in type **Seq[]** for sequences/queues

`now` is used to measure the time

The **initially** clause specifies a predicate that must be true of the automaton parameters and its initial state

TIOA Example 4.1: A Time-Bounded Channel (5/6)

Transitions Defines 2 actions:

`send(m)`

```
input send(m)
    eff
        queue := append([m,now+b], queue)
```

- Transitions on input actions have no preconditions, i.e. it is as if the precondition was **true**, which is omitted

`receive(m)`

```
output receive(m)
    pre
        head(queue).message = m
    eff
        queue := tail(queue)
```

- A `receive(m)` transition can occur only when `m` is the first message in the queue and it results in the removal of the first message from the queue

TIOA Example 4.1: A Time-Bounded Channel (6/6)

```
trajectories
    stop when
        ∃p: Packet p ∈ queue ∧ (now = p.deadline)
    evolve
        d(now) = 1
```

`stop when` specifies a stopping condition, which must hold only in the last state of the trajectory

- It ensures that time does not advance beyond the point where the stopping condition is true

`evolve` specifies the algebraic and differential equations that must be satisfied by the trajectories

- It is assumed that each variable follows a continuous function throughout a trajectory
- This implies that the value of a discrete variable is constant throughout a trajectory

TIOA Example 4.2: Periodic Sending Process

Process that sends messages every u time units

```
automaton PeriodicSend(u: Real, M: Type)
    signature
        output send(m: M)
    states
        clock: Real := 0
        initially u ≥ 0
    transitions
        output send(m)
            pre
                clock = u
            eff
                clock := 0
    trajectories
        stop when
            clock = u
        evolve
            d(clock) = 1
```

TIOA Example 4.3: Periodic Sending Process with Crashes

Process that send messages every u time units, unless it crashes

```
automaton PeriodicSend2(u: Real, M: Type)
    signature
        input crash
        output send(m: M)
    states
        crashed: Bool := false,
        clock: Real := 0
        initially u ≥ 0
    transitions
        output send(m)
            pre
                ¬crashed ∧ clock = u
            eff
                clock := 0
        input crash
            eff
                crashed := true
    trajectories
        stop when
            ¬crashed ∧ clock = u
        evolve
            d(clock) = 1
```

TIOA Example 4.4: Timeout Process

Process that awaits the receipt of a message from another process, performing a `timeout` action if u time units elapse without receiving it

```
automaton Timeout(u: Real, M: Type)
  signature
    input receive(m: M)
    output timeout
  states
    suspected: Bool := false,
    clock: Real := 0,
    initially u ≥ 0
  transitions
    input receive(m)
      eff
        clock := 0
        suspected := false
    output timeout
      pre
        ¬suspected ∧ clock = u
      eff
        suspected := true
  trajectories
    stop when
      ¬suspected ∧ clock = u
    evolve
      d(clock) = 1
```

Alternatively:

```
  trajectories
    trajdef suspected
      invariant suspected
      evolve d(clock) = 1
    trajdef notsupected
      invariant ¬suspected
      stop when clock = u
      evolve d(clock) = 1
```

TIOA Example 4.5: Clock Synchronization (1/2)

- Process in a clock synchronization algorithm. Each process:
  - Has a physical clock, which may drift from the real time with a drift rate bounded by `r`
  - Generates a logical clock
- The goal of the algorithm is to achieve:

  Agreement i.e. that the logical clocks are close to one another

  Validity i.e. that the logical clocks are within the range of the physical clocks
- **Idea** is to periodically exchange the physical clock values between the different processes and set the logical clock to the maximum value of all the physical clock values
  - The logical clock, `logclock` is a **derived variable**, which is a function whose value is defined in terms of the state variables

TIOA Example 4.5: Clock Synchronization (2/2)

```
automaton ClockSync(u,r : Real, i: Index)
  signature
    input  receive(m: Real, j: Index, const i: Index) where j ≠ i
    output send(m: Real, const i: Index),
  states
    nextsend: discrete Real := 0,
    maxother: discrete Real := 0,
    physclock: Real := 0,
    initially u ≥ 0 ∧ (0 ≤ r < 1)
  derived variables
    logclock = max(maxother, physclock)
  transitions
    output send(m, i)
      pre
        m = physclock ∧ physclock = nextsend
      eff
        nextsend := nextsend + u
    input receive(m,j,i)
      eff
        maxother := max(maxother, m)
  trajectories
    stop when physclock = nextsend
    evolve (1−r) ≤ d(physclock) ≤ (1 + r)
```

Outline

TIOA Behavior: Executions

- Like with (non-timed) I/O automata, executions record what happens during a particular run of a system. In the case of a timed system this means:
  - all the instantaneous, discrete state changes
  - all the changes to the state that occur while time advances

Execution of a timed automata $\mathcal{A}$ is an alternating sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$ where:

1. each τ_i is a trajectory in $\mathcal{T}$
2. $\tau_0.fstate$ is a start state
3. if τ_i is not the last trajectory in α then $\tau_i.lstate \xrightarrow{a_{i+1}} \tau_{i+1}.fstate$

Note To allow for simultaneous actions, i.e. actions occurring at the same time instant, a special **point trajectory**, $\wp(v)$, whose domain is the interval $[0, 0]$ is defined

Reachable state is a last state of a closed execution, i.e. of an execution whose last trajectory is closed

Invariant (assertion) is a predicate that is true for all the reachable states of a TIOA

TIOA Behavior: Traces

The trace of an execution of a TIOA captures its external behavior. It consists of a sequence of alternating

External actions

- By definition, internal actions are not externally observable

Trajectories over the empty set of variables, $\emptyset$ – they capture the amount of time that elapses between external actions

- Trajectories describe the evolution in time of state variables
- State variables are internal, i.e. they are not externally visible

TIOA Behavior Ex. 4.9: Periodic Sending Process (1/2)

Consider the TA of Example 4.2 where,

- u is instantiated to the real number 3
- `M` is instantatiated to the set $\{$`m1,m2,...`$\}$

Then the following sequence is an execution of the automaton:

$$\alpha = \tau \text{ send(m1) } \tau \text{ send(m2) } \tau \text{ send(m3) } \tau \ldots$$

where: $\tau : [0, 3] \rightarrow val(\{\text{clock}\})$ is defined such that $\tau(t)(\text{clock}) = t$ for all $t \in [0, 3]$

- The function τ is defined for closed intervals of length 3, starting at time 0
- It describes the evolution of the variable `clock`, which is 0 at the start of τ and increases with rate 1 for 3 time units
- The discrete `send` events occur periodically, every 3 time units and reset the `clock` variable to 0

TIOA Behavior Ex. 4.9: Periodic Sending Process (2/2)

The trace of the above execution is the sequence:

$$trace(\alpha) = \alpha' = \tau' \text{ send(m1) } \tau' \text{ send(m2) } \tau' \text{ send(m3) } \tau' \ldots$$

where $\tau' : [0, 3] \rightarrow val(\emptyset)$

- $trace(\alpha)$ does not contain any information about what happens to the value of `clock` as time progresses
  - The range of function τ' contains only the function with the empty domain
- α and α' express the same information about the amount of time that elapses between discrete steps.
  - The domains of τ and τ' are identical,

TIOA Behavior Example 4.10: Timeout Process

Consider the TIOA of Example 4.4 where

- the maximum waiting time for a message u is 5
- the message alphabet M is the set $\{\texttt{m1},\texttt{m2}\}$

Then the following sequence is an execution of the automaton:

$$\alpha = \tau_0 \text{ receive(m1) } \tau_1 \text{ timeout } \tau_2 \text{ send(m2) } \tau_3 \text{ timeout } \tau_4$$

where: $Val = val(\{\text{suspected,clock}\})$ and the trajectories $\tau_0, \tau_1, \tau_2, \tau_3, \tau_4$ are defined as follows:

$\tau_0 : [0,2] \to Val$ where $\tau_0(t)(\text{suspected}) = \text{false}$ and $\tau_0(t)(\text{clock}) = t$ for all $t \in [0,2]$

$\tau_1 : [0,5] \to Val$ where $\tau_1(t)(\text{suspected}) = \text{false}$ and $\tau_1(t)(\text{clock}) = t$ for all $t \in [0,5]$

$\tau_2 : [0,1] \to Val$ where $\tau_2(t)(\text{suspected}) = \text{true}$ and $\tau_2(t)(\text{clock}) = 5 + t$ for all $t \in [0,1]$

$\tau_3 : [0,5] \to Val$ where $\tau_3(t)(\text{suspected}) = \text{false}$ and $\tau_3(t)(\text{clock}) = t$ for all $t \in [0,5]$

$\tau_4 : [0,\infty) \to Val$ where $\tau_4(t)(\text{suspected}) = \text{true}$ and $\tau_4(t)(\text{clock}) = 5 + t$ for all $t \in [0,\infty)$

- The automaton Timeout can perform multiple timeout transitions
- This execution is a finite alternating sequence ending with a trajectory

TIOA Behavior Ex. 4.11: Time-Bounded Channel (1/6)

Consider the time-bounded channel from Example 4.1. Clearly,

- Time cannot pass beyond any delivery deadline recorded in the message queue
- Each deadline in the queue is less than or equal to the sum of the current time and bound b

We can state this property as an invariant (assertion) as follows:

Invariant: In any reachable state x of automaton `timedChannel`, for all $p \in x(queue)$, $x(now) \leq p.deadline \leq x(now) + b$

Alternatively, we could write $0 \leq x(now) - p.deadline \leq b$

TIOA Behavior Ex. 4.11: Time-Bounded Channel (2/6)

Note that:

- Reachable states are the final states of closed executions
- Any closed execution is the concatenation of closed execution fragments, $\alpha_0 \frown \alpha_1 \frown \ldots \alpha_k$, where every α_i is
  - either a closed trajectory
  - or a discrete action surrounded by point trajectories

  and where $\alpha_i.\mathit{lstate} = \alpha_{1+1}.\mathit{fstate}$ for $0 \leq i \leq k-1$.

Thus the invariant can be proved using induction on the length k of the sequence of execution fragments α_i

TIOA Behavior Ex. 4.11: Time-Bounded Channel (3/6)

Invariant: In any reachable state x of automaton `timedChannel`, for all $p \in x(queue)$, $x(now) \leq p.deadline \leq x(now) + b$

Proof By induction on the length k of the sequence of execution fragments α_i

Base case: k=1 In the initial state, $x = \alpha_0.fstate$

$$x(queue) = \{\}$$

Consider 2 cases:

1. α_0 is an action surrounded by point trajectories Clearly the only action that may occur is a **send(m)**. Thus, for $x = \alpha_0.lstate$ we have $x(queue) = \{[m,b]\}$ and $x(now) = 0$, hence the invariant is satisfied
2. α_0 is a trajectory Thus $queue = \{\}$, for all states in the trajectory and the invariant is trivially satisfied

TIOA Behavior Ex. 4.11: Time-Bounded Channel (4/6)

Invariant: In any reachable state x of automaton `timedChannel`, for all $p \in x(queue)$, $x(now) \leq p.deadline \leq x(now) + b$

Induction step Let's assume that the invariant is true for an execution with k trajectories $\alpha_0 \frown \alpha_1 \cdots \frown \alpha_{k-1}$. Consider 2 cases:

1. α_k is an action surrounded by point trajectories First note that now does not change (time does not advance) in α_k, i.e. $\alpha_{k-1}.lstate(now) = \alpha_k.fstate(now) = \alpha_k.lstate(now)$.
   Now, the action can be either:

   receive(m) removes the packet at the head of queue, and leaves the remaining packets in the queue. By the induction hypothesis, these packets, if any, satisfy the invariant

   send(m) appends a packet [m,$\alpha_k.fstate(now)$+b] to the queue. Clearly, at $\alpha_k.lstate$ this packet satisfies the invariant. The remaining packets, if any, were already in the queue, and by the induction hypothesis satisfy the invariant

TIOA Behavior Ex. 4.11: Time-Bounded Channel (5/6)

Invariant: In any reachable state x of automaton `timedChannel`, for all $p \in x(queue)$, $x(now) \leq p.deadline \leq x(now) + b$

Induction step

2. α_k is a trajectory In this case the state of queue does not change, and now increases at the same rate as time. We consider two cases:
   i. queue is empty then the invariant is trivially true
   ii. otherwise we consider the two inequalities separately:
      $p.deadline \leq x(now) + b$ this results directly from the induction hypothesis and that now increases monotonically
      $x(now) \leq p.deadline$ by the stopping clause, the predicate

$$\exists p: \text{Packet } p \in queue \wedge (now = p.deadline)$$

cannot be true except in $\alpha_k.lstate$.
Therefore for all states x but the last in α_k, for all packets p in $x(queue)$ we have: $x(now) < p.deadline$
For $\alpha_k.lstate$ there may be some packet p such that $x(now) = p.deadline$

TIOA Behavior Ex. 4.11: Time-Bounded Channel (6/6)

- In the `timedChannel` automaton, if instead of specifying the trajectory as:

```
stop when
    ∃p: Packet p ∈ queue ∧ (now = p.deadline)
evolve
    d(now) = 1
```

  we had specified it as:

```
stop when
    queue ≠ ∅ ∧ (now = head(queue).deadline)
evolve
    d(now) = 1
```

  we would have had some more work to prove the invariant
- The "low level" of TIOA is a mixed blessing:
  - On one hand, it forces us to consider every detail, making it hard to "prove" something that is not true
  - On the other hand, proving even an "obvious assertion" requires a lot of work

Outline

Composition of TIOA: Introduction

- Allows an automaton representing a complex system to be constructed by composing automata representing individual system components
- The composed automaton is built by matching:
  - each output action of the component automata
  - with input actions with the same name in different component automata
- When any component automaton performs a discrete step involving an action a, so do all component automata that have a as an external action
  - I.e. automata in a composed automaton **synchronize** on external actions with the same name

Composition of TIOA: Definition

Compatible Automata TIOA $\mathcal{A}_1$ and $\mathcal{A}_2$ are **compatible** if:

1. $X_1 \cap X_2 = \emptyset$, i.e. their (internal) variables are disjoint
2. $H_1 \cap A_2 = H_2 \cap A_1 = \emptyset$, i.e. the internal actions of one TA is disjoint from the actions of the other TA
3. $O_1 \cap O_2 = \emptyset$, i.e. their output actions are disjoint

Composition If $\mathcal{A}_1$ and $\mathcal{A}_2$ are **compatible** then their composition $\mathcal{A}_1 || \mathcal{A}_2$ is defined to be the TA $\mathcal{A} = (\mathcal{X}, \mathcal{Q}, \times, \mathcal{I}, \mathcal{O}, \mathcal{H}, \mathcal{D}, \mathcal{T})$ where:

- $X = X_1 \cup X_2$
- $Q = \{x \in val(X) | x\lceil X_i \in Q_i, i \in \{1,2\}\}$, i.e. $Q = Q_1 \times Q_2$
- $\Theta = \{x \in Q | x\lceil X_i \in \Theta_i, i \in \{1,2\}\}$, i.e. $\Theta = \Theta_1 \times \Theta_2$
- $O = O_1 \cup O_2$, $I = (I_1 \cup I_2) - O$ and $H = H_1 \cup H_2$
- For each $x, x' \in Q$ and each $a \in A$, $x \xrightarrow{a}_{\mathcal{A}} x'$ iff for $i \in \{1,2\}$ either (1) $a \in A_i$ and $x\lceil X_i \xrightarrow{a}_{\mathcal{A}_i} x'\lceil X_i$, or (2) $a \notin A_i$ and $x\lceil X_i = x'\lceil X_i$
- $\mathcal{T} \subseteq trajs(Q)$ is given by $\tau \in \mathcal{T} \Leftrightarrow \tau \downarrow X_i \in \mathcal{T}_i, i \in 1,2$

Composition: Fundamental Properties

Composition The composition of TIOAs is a TIOA

Theorem 7.2 If $\mathcal{A}_1$ and $\mathcal{A}_2$ are TIOAs, then $\mathcal{A}_1||\mathcal{A}_2$ is a TIOA

Executions the execution fragments of a composition of TIOA **project** to give execution fragments of the component automata.

Lemma 5.2 Let $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$ and let α be an execution fragment of $\mathcal{A}$. Then $\alpha\lceil(A_1, X_1)$ and $\alpha\lceil(A_2, X_2)$ are execution fragments of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively.

Traces satisfy the following **projection** and **pasting** result:

Theorem 7.3 Let $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$. Then $traces_{\mathcal{A}}$ is exacty the set of $(E, \emptyset)$-sequences whose restrictions to $\mathcal{A}_1$ and $\mathcal{A}_2$ are traces of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. That is,

$$traces_{\mathcal{A}} = \{\beta | \beta \text{ is an } (E, \emptyset)\text{-sequence and } \beta\lceil(E_i, \emptyset) \in traces_{\mathcal{A}_i}, i \in \{1, 2\}\}$$

Composition Ex. 5.5: Periodic Process w/ Timeouts (1/8)

Notation To avoid name clashes, when necessary, we refer to an internal variable v of TA $\mathcal{A}$ in the composite TA as $\mathcal{A}.v$

Let $\mathcal{C}$ be the composition of three automata from examples 4.1, 4.2, 4.4

$$\mathcal{C} = \text{PeriodicSend} \,||\, \text{TimedChannel} \,||\, \text{Timeout}$$

where $\text{M} = \{\text{m1}, \ldots, \text{mn}\}$ and $\text{b} + \text{PeriodicSend.u} < \text{Timeout.u}$
If $\text{b} < \text{u1}$, where $\text{u1} = \text{PeriodicSend.u}$, the following sequence is a trace of $\mathcal{C}$:

$$\alpha = \overline{\text{u1}}\ \text{send(m1)}\ \overline{\text{b}}\ \text{receive(m1)}\ \overline{\text{u1-b}}\ \text{send(m2)}\ \overline{\text{b}}\ \text{receive(m2)}\ \overline{\text{u1-b}} \ldots$$

where $\bar{\text{t}}$ denotes the trace with domain $[0, \text{t}]$ and as range the set consisting of the function with the empty domain

Composition Ex. 5.5: Periodic Process w/ Timeouts (2/8)

Invariant 1 In any reachable state x of $\mathcal{C}$, $x(\text{suspected}) = \text{false}$

Given that suspected is set to true upon occurrence of a timeout action, we will prove the following invariant:

Invariant 2 In any reachable state x of $\mathcal{C}$,

1. if $x(\text{queue})$ is not empty then there is a packet p such that $p \in x(\text{queue})$ and $p.\text{deadline} - x(\text{now}) < u2 - x(\text{Timeout.clock})$
2. if $x(\text{queue})$ is empty then $u1 - x(\text{PeriodicSend.clock}) + b < u2 - x(\text{Timeout.clock})$

where $u1 = \text{PeriodicSend.u}$ and $u2 = \text{Timeout.u}$

which states that the variable Timeout.clock never reaches the point at which a timeout action occurs:

1. ensures that if there is any message in transit it will be delivered before there is a timeout
2. ensures that if there is no message in transit, a send action will occcur early enough, so that no timeout will occur

Composition Ex. 5.5: Periodic Process w/ Timeouts (3/8)

Invariant 2 In any reachable state x of $\mathcal{C}$,

1. if $x(\text{queue})$ is not empty then there is a packet p such that $p \in x(\text{queue})$ and $p.\text{deadline} - x(\text{now}) < u2 - x(\text{Timeout.clock})$
2. if $x(\text{queue})$ is empty then $u1 - x(\text{PeriodicSend.clock}) + b < u2 - x(\text{Timeout.clock})$

where $u1 = \text{PeriodicSend.u}$ and $u2 = \text{Timeout.u}$

To prove this invariant we will follow the same approach as in the proof of invariant in Example 4.11

- I.e., we'll use induction on the number of **elementary** execution fragments, i.e.
  - either a closed trajectory
  - or a discrete action surrounded by point trajectories

Composition Ex. 5.5: Periodic Process w/ Timeouts (4/8)

Invariant 2 In any reachable state x of $\mathcal{C}$,

1. if x(queue) is not empty then there is a packet p such that
   $p \in x(queue)$ and $p.deadline - x(now) < u2 - x(Timeout.clock)$
2. if x(queue) is empty then
   $u1 - x(PeriodicSend.clock) + b < u2 - x(Timeout.clock)$

where $u1 = PeriodicSend.u$ and $u2 = Timeout.u$

Base case, k = 1 In this case, α_0 must be a trajectory, and throughout this trajectory we have:

$$x(queue) = \{\}$$
$$x(PeriodicSend.clock) = x(Timeout.clock)$$

Given that $u1 + b < u2$, it follows that

$$u1 - x(PeriodicSend.clock) + b < u2 - x(Timeout.clock)$$

Composition Ex. 5.5: Periodic Process w/ Timeouts (5/8)

Invariant 2 In any reachable state x of $\mathcal{C}$,

1. if x(queue) is not empty then there is a packet p such that
   $p \in x(queue)$ and $p.deadline - x(now) < u2 - x(Timeout.clock)$
2. if x(queue) is empty then
   $u1 - x(PeriodicSend.clock) + b < u2 - x(Timeout.clock)$

where $u1 = PeriodicSend.u$ and $u2 = Timeout.u$

Induction step We consider two cases:

$\alpha_{k-1}.lstate(queue) = \{\}$
$\alpha_{k-1}.lstate(queue) \neq \{\}$

and for each of these two cases we need to consider the two possible types of execution fragments:

- either a closed trajectory
- or a discrete action surrounded by point trajectories

Composition Ex. 5.5: Periodic Process w/ Timeouts (6/8)

Induction step: $\alpha_{k-1}.lstate(queue) = \{\}$ By the inductive hypothesis:

$$u1 - x(\text{PeriodicSend.clock}) + b < u2 - x(\text{Timeout.clock})$$

α_k is a closed trajectory whose initial state satisfies this inequality. Given that the derivatives of both PeriodicSend.clock and Timeout.clock are 1, it holds true for all states in α_k

α_k is an action surrounded by point trajectories In this case the only possible action is a send(m). As a result, for $x = \alpha_k.lstate$ we have:

$$x(queue) = \{[m, x(now)+b]\}$$

Thus $p.deadline - x(now) = b$ for this packet. Given that time does not advance in α_k, and that $u1 - x(\text{PeriodicSend.clock}) \geq 0$, from the induction hypothesis it follows that

$$p.deadline - x(now) < u2 - x(\text{Timeout.clock})$$

is satisfied by the only packet in the queue at $x = \alpha_k.lstate$

Composition Ex. 5.5: Periodic Process w/ Timeouts (7/8)

Induction step: $\alpha_{k-1}.lstate(queue) \neq \{\}$ Thus, by the inductive hypothesis at $\alpha_k.fstate$ there is a packet $\mathsf{p} \in \mathsf{x(queue)}$ such that

$$\mathsf{p.deadline} - \mathsf{x(now)} < \mathsf{u2} - \mathsf{x(Timeout.clock)}$$

α_k is a closed trajectory then given that the derivatives of both now and Timeout.clock are 1, the inequality above holds true in all states of α_k for that packet

α_k is an action surrounded by point trajectories In this case the action can be either

send(m) in this case, given that the time does not advance, the inequality will continue to hold true for that packet at $x = \alpha_k.lstate$

receive(m) in this case we need to consider two cases, either the queue becomes empty or it does not.

Composition Ex. 5.5: Periodic Process w/ Timeouts (8/8)

Induction step: $\alpha_{k-1}.lstate(queue) \neq \{\}$ Thus, by the inductive hypothesis at $\alpha_k.fstate$ there is a packet $p \in x(queue)$ such that

$$p.deadline - x(now) < u2 - x(Timeout.clock)$$

α_k is receive(m) surrounded by point trajectories Either:

queue becomes empty From the parameter assumptions, we have: $u1 + b < u2$. Furthermore, at $x = \alpha_k.lstate$, $x(Timeout.clock) = 0$, and for all states x we have and $x(PeriodicSend.clock) >= 0$, thus it follows that at $x = \alpha_k.lstate$:

$$u1 - x(PeriodicSend.clock) + b < u2 - x(Timeout.clock)$$

otherwise in this case from invariant in Example 4.11 we have that for all packets in queue and all sates:: $x(now) \leq p.deadline \leq x(now)+b$

Thus, $p.dealine - x(now) \leq b$

From, the parameters assumption, and given that $u1 \geq 0$ it follows that: $p.dealine - x(now) \leq b \leq u1 + b < u2$

Finally, for $x = \alpha_k.lstate$ given that $x(Timeout.clock) = 0$ it follows that

$$p.deadline - x(now) < u2 - x(Timeout.clock)$$

Outline

Timed I/O Automata with Bounds: Rationale

- **TIOA with bounds** are a new class of TIOA that extends TIOA with:

  Tasks which are sets of locally controlled actions

  Bounds which impose constraints on the time when an action may be performed

- This class makes it easier to present many results on the partially synchronous model that assume that there are bounds (both lower and upper) on the time processes take to perform a step of an algorithm
- We'll restrict our attention to a class of automata where every action:

  1. either is enabled/disabled throughout an entire trajectory
  2. or becomes enabled once during a trajectory and remains so until the end of that trajectory

  A trajectory $\mathcal{T}$ that satisfies this property wrt to a set of actions C is said to be **well-formed** wrt C

Time I/O Automata with Bounds: Definition

A TIOA with bounds $\mathcal{A} = (X, Q, \Theta, I, O, H, \mathcal{D}, \mathcal{T}, C, l, u)$ consists of:

$(X, Q, \Theta, I, O, H, \mathcal{D}, \mathcal{T})$ a TIOA

$C \subseteq I \cup O \cup H$ i.e. a set of actions

- C is called a **task**
- $\mathcal{T}$ is well-formed wrt C

l a lower bound $l \in R^{\geq 0}$

u an upper bound $l \in R^{\geq 0} \cup \{\infty\}$, with $l \leq u$

- Lower and upper bounds are used to specify how much time is allowed to pass between the enabling and the performance of an action:

  Lower bound l is the minimum time that an action must be enabled before it is performed

  Upper bound u is the maximum time that an action may be enabled without being performed:

  - i.e. it must either be performed or become disabled after u time units

TIOA with Bounds Example: Timeout Process (1/2)

- P_2 waits for the reception of a message from another process P_1
- If no such message arrives within a certain amount of time, P_2 performs a *timeout* action.
- P_2 measures the elapsed time by counting a fixed number $k \geq 1$ of its own steps, which are supposed to have known lower and upper bounds $\ell_1, \ell_2, 0 < \ell_1 \leq \ell_2 < \infty$:
  - In Example 4.4 above, we used a local clock
- Its *timeout* action is performed at most time ℓ after its *count* reaches 0.

Note that the definition of the TIOA with bounds assumes the existence of only one task per TIOA, but it can be easily generalized to TIOA with an arbitrary number of tasks

TIOA with Bounds Example: Timeout Process (2/2)

```
automaton Timeout(k : Int, M: Type)
  signature
    input  receive(m: M)
    internal decrement
    output timeout
  states
    suspected: Bool := false,
    counter: Int := k,
    initially k ≥ 1
  transitions
    input receive(m,j,i)
      eff
        counter := k
        suspected := false
    output timeout
      pre
        suspected = false ∧ counter = 0
      eff
        suspected = true
    internal decrement
      pre
        counter ≠ 0
      eff
        counter := counter − 1
  tasks
    decr = {decrement};
    susp = {timeout}
  bounds
    decr = [ℓ1,ℓ2];
    susp = [0, ℓ]
```

Time I/O Automata with Bounds: The **Extend** Operation

The **Extend** operation transforms a TIOA $\mathcal{A} = (\mathcal{B}, C, l, u)$ with bounds to another TIOA $\mathcal{A}' = (X', Q', \Theta', I, O, H, \mathcal{D}', \mathcal{T}'$ that incorporates the bounds in addition to the timing constraints already present in $\mathcal{B}$.

Basically:

- $X' = X \cup \{now, first, last\}$, where

  now is an analog variable such that $type(now) = R$

  $first$ and $last$ are discrete variables where $type(first) = R$ and $type(last) = R \cup \{\infty\}$

  Variables $now, first, last$ are new variables that do not appear in X
- $Q' = Q \times val(now, first, last)$
- Θ' is obtained from Θ by assigning the following valuations to $x \in \Theta$:

  $x(now) = 0$

  $$x(first) = \begin{cases} l & \text{if } C \text{is enabled in } x \\ 0 & \text{otherwise} \end{cases}$$

  $$x(last) = \begin{cases} u & \text{if } C \text{is enabled in } x \\ \infty & \text{otherwise} \end{cases}$$

Time I/O Automata with Bounds: The **Extend** Operation

- $\mathcal{D}'$ is obtained from $\mathcal{D}$ by adding the predicate $x(\mathit{first}) \leq x(\mathit{now})$ for transitions $x \xrightarrow{a} x'$, when $a \in C$
- $\mathcal{T}'$ is obtained from $\mathcal{T}$ by adding the following trajectory:

```
stop when now ≤ last
evolve d(now) = 1
```

Note that *now*, *first* and *last* all represent absolute time:

- A full formal definition requires that the values of *first* and *last* be updated whenever actions in C are enabled, disabled or performed
  - I.e. the "definition" above omitted details on how $\mathcal{D}'$ and $\mathcal{T}'$ are obtained from $\mathcal{D}$ and $\mathcal{T}$

TIOA with Bounds and the Extend Operation

Theorem 5.18 Suppose that $\mathcal{A}$ is a TIOA with bounds. Then

$$traces_{\text{Extend}(\mathcal{A})} \subseteq traces_{\mathcal{A}}$$

I.e. it is possible to **implement** a TIOA with bounds with a TIOA without bounds.

Note the definition of the Extend operation assumes the existence of only one task per TIOA, but, like the definition of the TIOA with bounds, it can be easily generalized to TIOA with an arbitrary number of tasks

Outline

Further Reading

- Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The Theory of Timed I/O Automata. Technical Report MIT-LCS-TR-917a, MIT Laboratory for Computer Science, Cambridge, MA, November, 2004.
- Stephen Garland TIOA User Guide and Reference Manual. September 15, 2005.
- Stephen Garland, Dilsun Kaynar, Nancy Lynch, Joshua Tauber, and Mandana Vaziri. TIOA Tutorial. May 22, 2005.
- Chapter 23, *Modelling V: Partially Synchronous System Models*, of Nancy Lynch's *Distributed Algorithms*.