Formal Models and Specifications for Systems of Interacting Components

Part 2: Communication-Safe Team Automata and Specification of Team Properties

Rolf Hennicker

Ludwig-Maximilians-Universität München, Germany

DaVinci Spring School, Univ. do Minho, March 24-25, 2022

Thanks to Maurice ter Beek and Jetty Kleijn

Topics of this lecture

- Systems of reactive components which interact through message exchange.
- *Here:* Synchronous communication, i.e., outputs and inputs of the same message are performed simultaneously (*handshake*).
- Consideration of various synchronization types (peer-to-peer, multicast, broadcast, gathering, ...).
- Safe communication, avoidance of communication errors.
- Specification of behavioural system properties using a variant of dynamic logic.

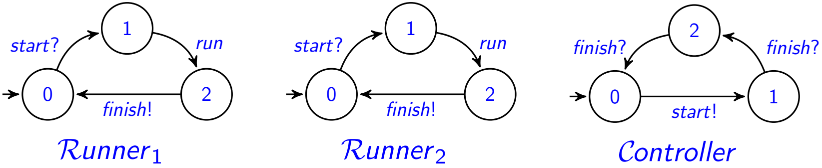
Topics of this lecture

- Systems of reactive components which interact through message exchange.
- *Here:* Synchronous communication, i.e., outputs and inputs of the same message are performed simultaneously (*handshake*).
- Consideration of various synchronization types (peer-to-peer, multicast, broadcast, gathering, ...).
- Safe communication, avoidance of communication errors.
- Specification of behavioural system properties using a variant of dynamic logic.

Topics of this lecture

- Systems of reactive components which interact through message exchange.
- *Here:* Synchronous communication, i.e., outputs and inputs of the same message are performed simultaneously (*handshake*).
- Consideration of various synchronization types (peer-to-peer, multicast, broadcast, gathering, ...).
- Safe communication, avoidance of communication errors.
- Specification of behavioural system properties using a variant of dynamic logic.

Component Systems: Running Example



Component Systems: Definitions

Component automaton is a tuple $\mathcal{A} = (Q, q^0, \Sigma, \to)$ such that

- Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*,
- Σ is the disjoint union of sets Σ_{inp}, Σ_{out}, Σ_{int} of *input*, *output*, and *internal actions*,
- $\to \subseteq Q \times \Sigma \times Q$ is a *labelled transition relation*.

Component system is an indexed set $\mathcal{S} = \{\, \mathcal{A}_i \mid i \in I \,\}$ of component automata $\mathcal{A}_i = (Q_i, q_i^0, \Sigma_i, \to_i)$.
We assume that the index set I is finite.

A system is *closed*, if for any input action $a \in \Sigma_{i,inp}$ of some component $i \in I$ there is a corresponding output action $a \in \Sigma_{j,out}$ of another component $j \in I$, and conversely.

In the sequel we consider closed systems.

Component Systems: Definitions

Component automaton is a tuple $\mathcal{A} = (Q, q^0, \Sigma, \rightarrow)$ such that

- Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*,
- Σ is the disjoint union of sets Σ_{inp}, Σ_{out}, Σ_{int} of *input*, *output*, and *internal actions*,
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is a *labelled transition relation*.

Component system is an indexed set $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$ of component automata $\mathcal{A}_i = (Q_i, q_i^0, \Sigma_i, \rightarrow_i)$.
We assume that the index set I is finite.

A system is *closed*, if for any input action $a \in \Sigma_{i,inp}$ of some component $i \in I$ there is a corresponding output action $a \in \Sigma_{j,out}$ of another component $j \in I$, and conversely.

In the sequel we consider closed systems.

Component Systems (continued)

System state is a tuple $(q_i)_{i \in I}$ with $q_i \in Q_i$ for all $i \in I$.

System transition is a transition between system states

$$(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q'_i)_{i \in I}$$

such that

- either: a is an internal action of some component $\mathcal{A}_i$ and $outs = \{i\}, ins = \emptyset$, $q_i \xrightarrow{a}_i q'_i$ and $q_j = q'_j$ for all $j \in I \setminus \{i\}$,
- or: $outs, ins \subseteq I$, $outs \cap ins \neq \emptyset$, and
  for all $i \in outs$: a is an output action of $\mathcal{A}_i$,
  for all $i \in ins$: a is an input action of $\mathcal{A}_i$,
  $q_i \xrightarrow{a}_i q'_i$ for all $i \in outs \cup ins$, i.e. simultaneous execution, and
  $q_i = q'_i$ for all $i \in I \setminus (outs \cup ins)$.

  $outs$ indicates the senders of a, ins the receivers of a.

Not all system transitions are meaningful in an application!

Component Systems (continued)

System state is a tuple $(q_i)_{i\in I}$ with $q_i \in Q_i$ for all $i \in I$.

System transition is a transition between system states

$$(q_i)_{i\in I} \xrightarrow{(outs,a,ins)} (q_i')_{i\in I}$$

such that

- either: a is an internal action of some component $\mathcal{A}_i$ and $outs = \{i\}, ins = \emptyset$, $q_i \xrightarrow{a}_i q_i'$ and $q_j = q_j'$ for all $j \in I \setminus \{i\}$,
- or: $outs, ins \subseteq I$, $outs \cap ins \neq \emptyset$, and
  for all $i \in outs$: a is an output action of $\mathcal{A}_i$,
  for all $i \in ins$: a is an input action of $\mathcal{A}_i$,
  $q_i \xrightarrow{a}_i q_i'$ for all $i \in outs \cup ins$, i.e. simultaneous execution, and
  $q_i = q_i'$ for all $i \in I \setminus (outs \cup ins)$.

  $outs$ indicates the senders of a, ins the receivers of a.

Not all system transitions are meaningful in an application!

Component Systems (continued)

System state is a tuple $(q_i)_{i \in I}$ with $q_i \in Q_i$ for all $i \in I$.

System transition is a transition between system states

$$(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q'_i)_{i \in I}$$

such that

- either: a is an internal action of some component $\mathcal{A}_i$ and $outs = \{i\}, ins = \emptyset$, $q_i \xrightarrow{a}_i q'_i$ and $q_j = q'_j$ for all $j \in I \setminus \{i\}$,
- or: $outs, ins \subseteq I$, $outs \cap ins \neq \emptyset$, and
  for all $i \in outs$: a is an output action of $\mathcal{A}_i$,
  for all $i \in ins$: a is an input action of $\mathcal{A}_i$,
  $q_i \xrightarrow{a}_i q'_i$ for all $i \in outs \cup ins$, i.e. simultaneous execution, and
  $q_i = q'_i$ for all $i \in I \setminus (outs \cup ins)$.

  $outs$ indicates the senders of a, ins the receivers of a.

Not all system transitions are meaningful in an application!

Component Systems (continued)

System state is a tuple $(q_i)_{i \in I}$ with $q_i \in Q_i$ for all $i \in I$.

System transition is a transition between system states

$$(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q_i')_{i \in I}$$

such that

- either: a is an internal action of some component $\mathcal{A}_i$ and $outs = \{i\}, ins = \emptyset$, $q_i \xrightarrow{a}_i q_i'$ and $q_j = q_j'$ for all $j \in I \setminus \{i\}$,
- or: $outs, ins \subseteq I$, $outs \cap ins \neq \emptyset$, and
  for all $i \in outs$: a is an output action of $\mathcal{A}_i$,
  for all $i \in ins$: a is an input action of $\mathcal{A}_i$,
  $q_i \xrightarrow{a}_i q_i'$ for all $i \in outs \cup ins$, i.e. simultaneous execution, and
  $q_i = q_i'$ for all $i \in I \setminus (outs \cup ins)$.

  $outs$ indicates the senders of a, ins the receivers of a.

Not all system transitions are meaningful in an application!

Component Systems (continued)

System state is a tuple $(q_i)_{i \in I}$ with $q_i \in Q_i$ for all $i \in I$.

System transition is a transition between system states

$$(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q_i')_{i \in I}$$

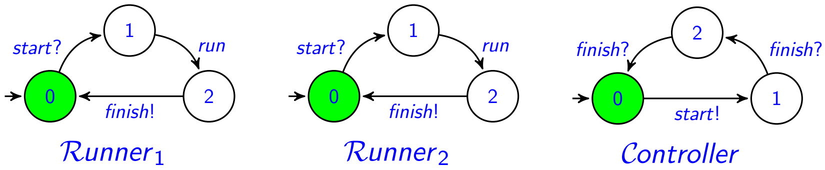
such that

- either: a is an internal action of some component $\mathcal{A}_i$ and $outs = \{i\}, ins = \emptyset$, $q_i \xrightarrow{a}_i q_i'$ and $q_j = q_j'$ for all $j \in I \setminus \{i\}$,
- or: $outs, ins \subseteq I$, $outs \cap ins \neq \emptyset$, and
  for all $i \in outs$: a is an output action of $\mathcal{A}_i$,
  for all $i \in ins$: a is an input action of $\mathcal{A}_i$,
  $q_i \xrightarrow{a}_i q_i'$ for all $i \in outs \cup ins$, i.e. simultaneous execution, and
  $q_i = q_i'$ for all $i \in I \setminus (outs \cup ins)$.

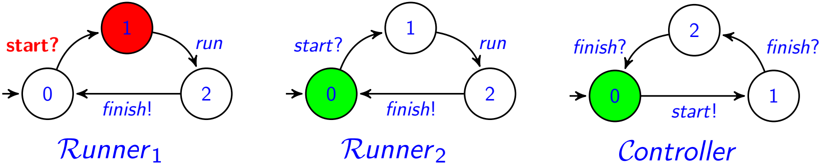
  $outs$ indicates the senders of a, ins the receivers of a.

Not all system transitions are meaningful in an application!

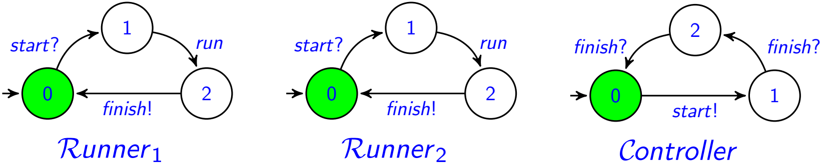
System Transitions: Example



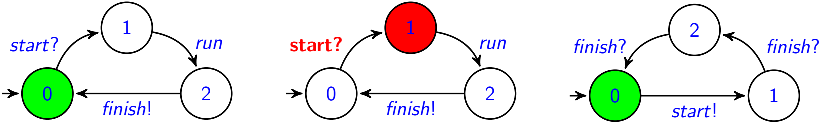
System Transitions: Example



System Transitions: Example

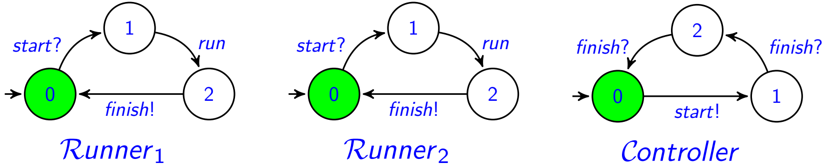


System Transitions: Example

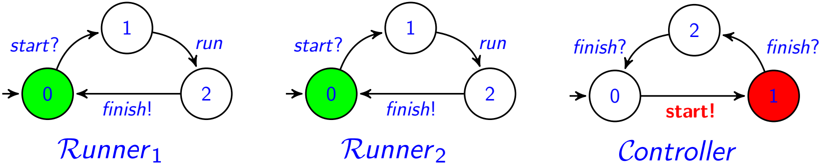


$\mathcal{R}unner_1$ $\mathcal{R}unner_2$ $\mathcal{C}ontroller$

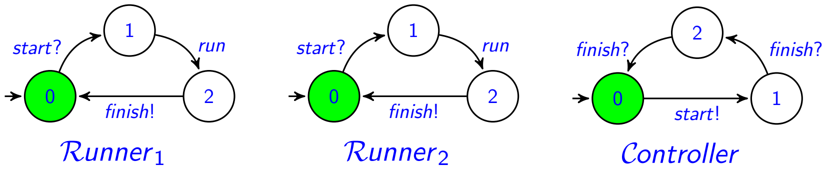
System Transitions: Example



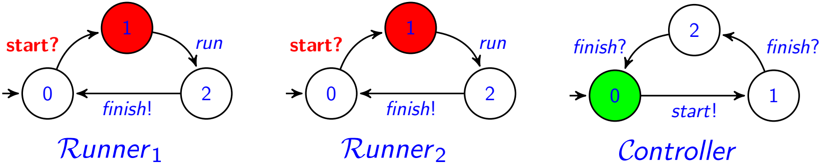
System Transitions: Example



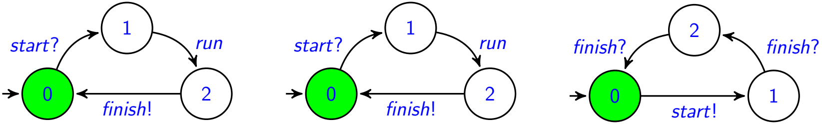
System Transitions: Example



System Transitions: Example

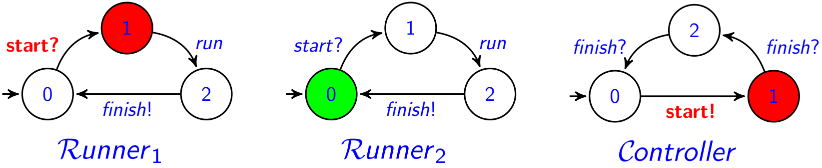


System Transitions: Example

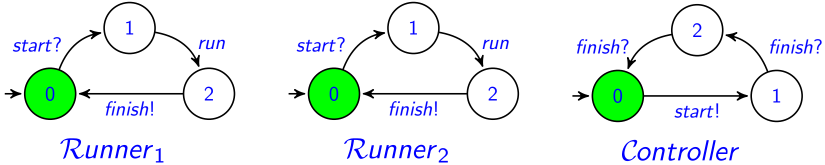


$\mathcal{R}unner_1$ $\mathcal{R}unner_2$ $\mathcal{C}ontroller$

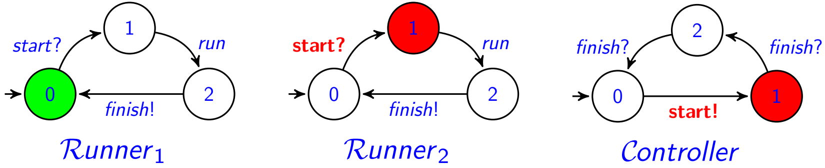
System Transitions: Example



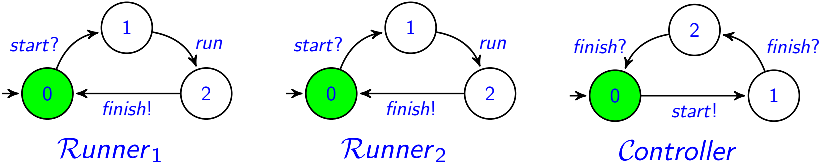
System Transitions: Example



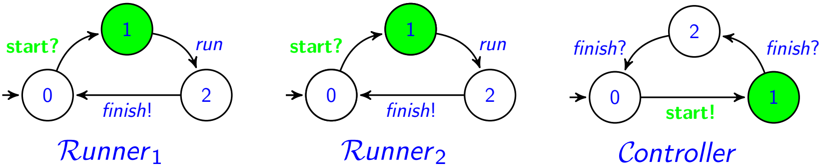
System Transitions: Example



System Transitions: Example



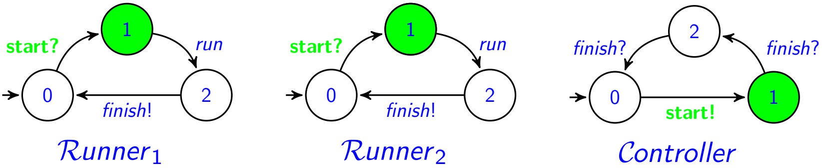
System Transitions: Example



$\mathcal{C}ontroller$ starts the two runners together.

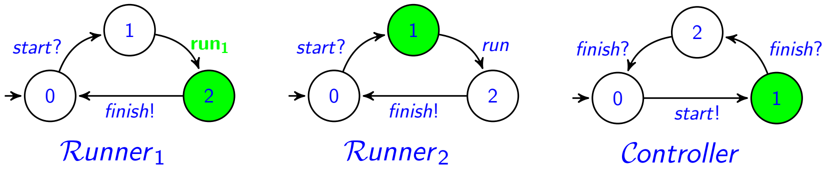
System Transitions: Example

$$(0,0,0) \xrightarrow{(\{\mathcal{C}ontroller\},start,\{\mathcal{R}unner_1,\mathcal{R}unner_2\})} (1,1,1)$$



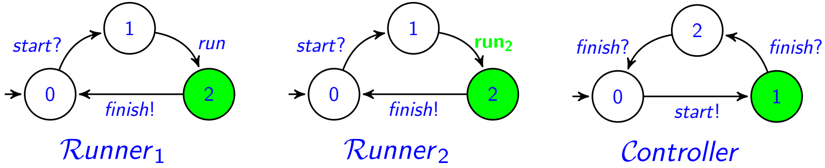
Controller starts the two runners together.

System Transitions: Example



$\mathcal{C}ontroller$ starts the two runners together.
$\mathcal{R}unner_1$ runs.

System Transitions: Example

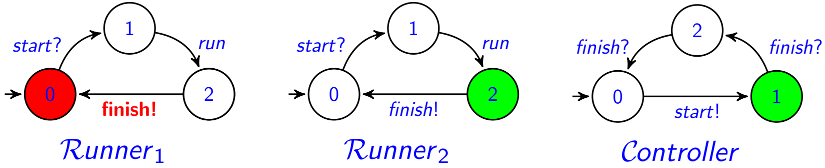


$\mathcal{C}ontroller$ starts the two runners together.

$\mathcal{R}unner_1$ runs.

$\mathcal{R}unner_2$ runs.

System Transitions: Example

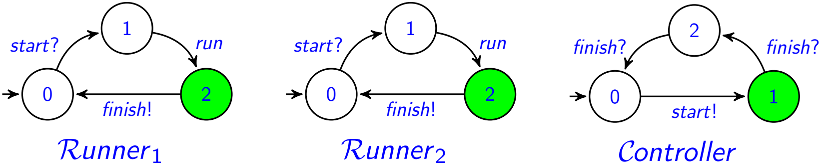


$\mathcal{C}ontroller$ starts the two runners together.

$\mathcal{R}unner_1$ runs.

$\mathcal{R}unner_2$ runs.

System Transitions: Example

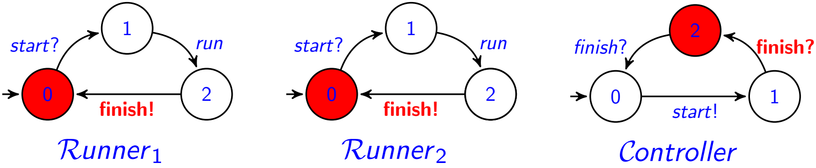


$\mathcal{C}ontroller$ starts the two runners together.

$\mathcal{R}unner_1$ runs.

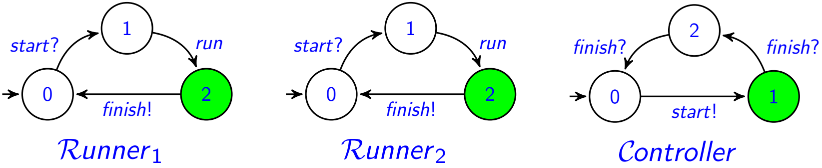
$\mathcal{R}unner_2$ runs.

System Transitions: Example



$\mathcal{C}ontroller$ starts the two runners together.
$\mathcal{R}unner_1$ runs.
$\mathcal{R}unner_2$ runs.

System Transitions: Example

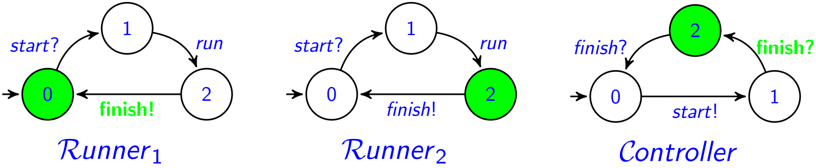


$\mathcal{C}ontroller$ starts the two runners together.

$\mathcal{R}unner_1$ runs.

$\mathcal{R}unner_2$ runs.

System Transitions: Example



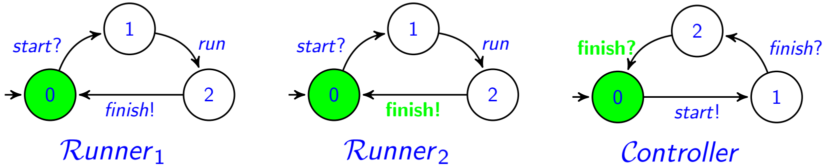
Controller starts the two runners together.

$\mathcal{R}unner_1$ runs.

$\mathcal{R}unner_2$ runs.

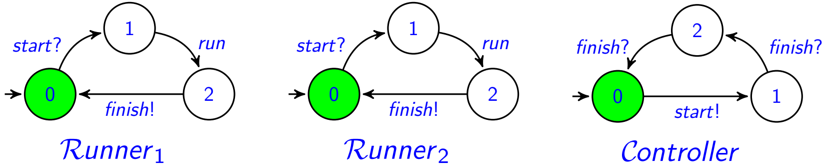
$\mathcal{R}unner_1$ signals *finish* to the *Controller*.

System Transitions: Example



$\mathcal{C}ontroller$ starts the two runners together.
$\mathcal{R}unner_1$ runs.
$\mathcal{R}unner_2$ runs.
$\mathcal{R}unner_1$ signals *finish* to the $\mathcal{C}ontroller$.
$\mathcal{R}unner_2$ signals *finish* to the $\mathcal{C}ontroller$.

System Transitions: Example



Controller starts the two runners together.
$\mathcal{R}unner_1$ runs.
$\mathcal{R}unner_2$ runs.
$\mathcal{R}unner_1$ signals *finish* to the *Controller*.
$\mathcal{R}unner_2$ signals *finish* to the *Controller*.

Idea: Given a system $\mathcal{S}$ of reactive components, choose a *synchronization policy* δ which is a selected subset of system transitions appropriate for your application.

How to specify synchronization policies?

Synchronization Types

Let $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$ be a component system.

Idea:

Specify, for each non-internal action a in $\mathcal{S}$, how many senders and how many receivers are allowed to participate in a system transition $(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q'_i)_{i \in I}$.

Formally:

A *synchronization type* is a pair $\mathtt{snd} \to \mathtt{rcv}$ consisting of

- a sending multiplicity $\mathtt{snd}$ and
- a receiving multiplicity $\mathtt{rcv}$

where a multiplicity has the form

- $[\mathtt{min}, \mathtt{max}]$ with $\mathtt{min} \in \mathbb{N}, \mathtt{max} \in \mathbb{N} \cup \{*\}$ such that $\mathtt{min} \leq \mathtt{max}$,

A *synchronization type specification* st over $\mathcal{S}$ assigns to each non-internal action a in $\mathcal{S}$ a synchronization type $st(a)$.

Synchronization Types

Let $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$ be a component system.

Idea:

Specify, for each non-internal action a in $\mathcal{S}$, how many senders and how many receivers are allowed to participate in a system transition $(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q'_i)_{i \in I}$.

Formally:

A *synchronization type* is a pair $\mathtt{snd} \to \mathtt{rcv}$ consisting of

- a sending multiplicity $\mathtt{snd}$ and
- a receiving multiplicity $\mathtt{rcv}$

where a multiplicity has the form

- $[\mathtt{min}, \mathtt{max}]$ with $\mathtt{min} \in \mathbb{N}, \mathtt{max} \in \mathbb{N} \cup \{*\}$ such that $\mathtt{min} \leq \mathtt{max}$,

A *synchronization type specification* st over $\mathcal{S}$ assigns to each non-internal action a in $\mathcal{S}$ a synchronization type $st(a)$.

Synchronization Types

Let $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$ be a component system.

Idea:

Specify, for each non-internal action a in $\mathcal{S}$, how many senders and how many receivers are allowed to participate in a system transition $(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q'_i)_{i \in I}$.

Formally:

A *synchronization type* is a pair $\mathtt{snd} \to \mathtt{rcv}$ consisting of

- a sending multiplicity $\mathtt{snd}$ and
- a receiving multiplicity $\mathtt{rcv}$

where a multiplicity has the form

- $[\mathtt{min}, \mathtt{max}]$ with $\mathtt{min} \in \mathbb{N}, \mathtt{max} \in \mathbb{N} \cup \{*\}$ such that $\mathtt{min} \leq \mathtt{max}$,

A *synchronization type specification* st over $\mathcal{S}$ assigns to each non-internal action a in $\mathcal{S}$ a synchronization type $st(a)$.

Example: Synchronization Types

$st(start) = [1,1] \to [2,2]$ shortly $1 \to 2$

This means: In a system transition labelled with *start* there must be exactly one sender and two receiver components. According to the alphabets of the components the sender can only be the controller and the receivers can only be the two runners.

$st(finish) = ([1,1],[1,1])$ shortly $1 \to 1$

This means: In any system transition labelled with *finish* there must be exactly one sender and one receiver component. According to the alphabets of the components the sender can only be a runner and the receiver can only be the controller.

Remark: $st(start) = 1 \to [0,*]$ would express that exactly one component (the controller) can send *start* and arbitrarily many receivers (runners) can join, even none.

Example: Synchronization Types

$st(start) = [1,1] \to [2,2]$ shortly $1 \to 2$

This means: In a system transition labelled with *start* there must be exactly one sender and two receiver components. According to the alphabets of the components the sender can only be the controller and the receivers can only be the two runners.

$st(finish) = ([1,1],[1,1])$ shortly $1 \to 1$

This means: In any system transition labelled with *finish* there must be exactly one sender and one receiver component. According to the alphabets of the components the sender can only be a runner and the receiver can only be the controller.

Remark: $st(start) = 1 \to [0,*]$ would express that exactly one component (the controller) can send *start* and arbitrarily many receivers (runners) can join, even none.

Example: Synchronization Types

$st(start) = [1,1] \to [2,2]$ shortly $1 \to 2$

This means: In a system transition labelled with *start* there must be exactly one sender and two receiver components. According to the alphabets of the components the sender can only be the controller and the receivers can only be the two runners.

$st(finish) = ([1,1],[1,1])$ shortly $1 \to 1$

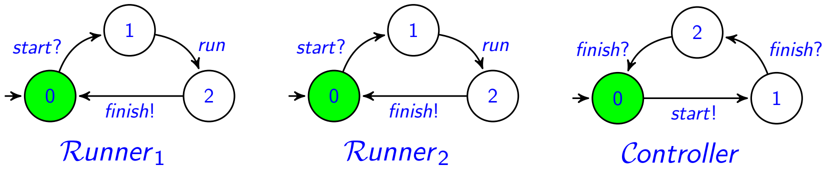
This means: In any system transition labelled with *finish* there must be exactly one sender and one receiver component. According to the alphabets of the components the sender can only be a runner and the receiver can only be the controller.

Remark: $st(start) = 1 \to [0,*]$ would express that exactly one component (the controller) can send *start* and arbitrarily many receivers (runners) can join, even none.

Team Automaton: Example

Synchronization types:

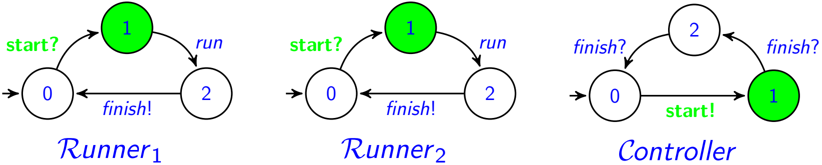
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$



Team Automaton: Example

Synchronization types:

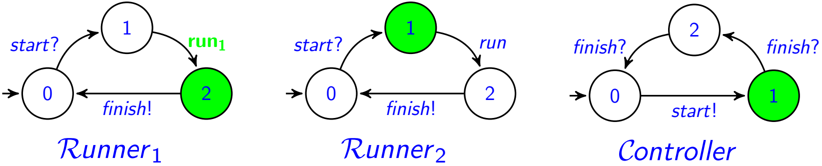
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$



Team Automaton: Example

Synchronization types:

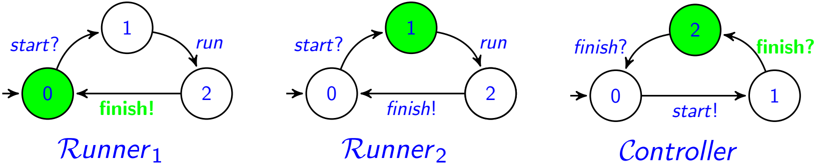
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$



Team Automaton: Example

Synchronization types:

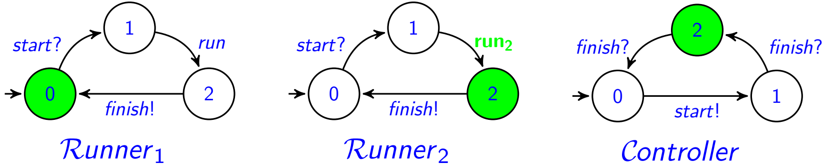
$st(start) = 1 \rightarrow 2$ and $st(finish) = 1 \rightarrow 1$



Team Automaton: Example

Synchronization types:

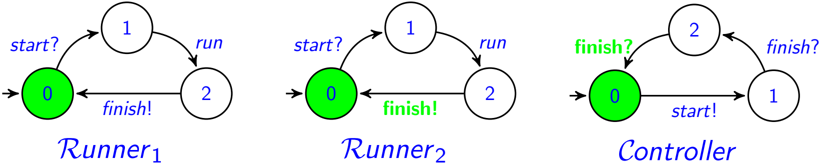
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$



Team Automaton: Example

Synchronization types:

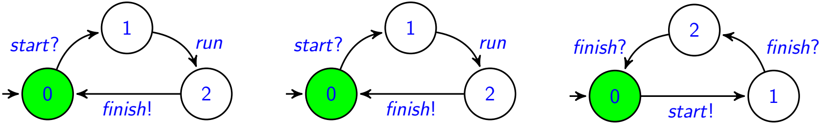
$st(start) = 1 \rightarrow 2$ and $st(finish) = 1 \rightarrow 1$



Team Automaton: Example

Synchronization types:

$st(start) = 1 \rightarrow 2$ and $st(finish) = 1 \rightarrow 1$



$\mathcal{R}unner_1$ $\mathcal{R}unner_2$ $\mathcal{C}ontroller$

Team Automata

Let $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$ be a component system and st be a synchronization type specification over $\mathcal{S}$.

st generates a so-called *team automaton*, denoted by $\mathcal{T}(st)$, over $\mathcal{S}$ such that

- the states of $\mathcal{T}(st)$ are the system states of $\mathcal{S}$, i.e. tuples $(q_i)_{i \in I}$ with q_i component states of $\mathcal{A}_i$ for all $i \in I$,
- the initial state of $\mathcal{T}(st)$ is $(q_i^0)_{i \in I}$ with q_i^0 the initial component state of $\mathcal{A}_i$ for all $i \in I$,
- the actions of $\mathcal{T}(st)$ have the form $(outs, a, ins)$, and
- the transitions of $\mathcal{T}(st)$ are
  - for internal actions a, all possible system transitions $(q_i)_{i \in I} \xrightarrow{(\{j\},a,\emptyset)} (q_i')_{i \in I}$,
  - for non-internal actions a, exactly those system transitions $(q_i)_{i \in I} \xrightarrow{(outs,a,ins)} (q_i')_{i \in I}$ of $\mathcal{S}$ such that $(outs, a, ins)$ **satisfies the synchronization type specification** $st(a)$.

Team Automata

Let $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$ be a component system and st be a synchronization type specification over $\mathcal{S}$.

st generates a so-called *team automaton*, denoted by $\mathcal{T}(st)$, over $\mathcal{S}$ such that

- the states of $\mathcal{T}(st)$ are the system states of $\mathcal{S}$, i.e. tuples $(q_i)_{i \in I}$ with q_i component states of $\mathcal{A}_i$ for all $i \in I$,
- the initial state of $\mathcal{T}(st)$ is $(q_i^0)_{i \in I}$ with q_i^0 the initial component state of $\mathcal{A}_i$ for all $i \in I$,
- the actions of $\mathcal{T}(st)$ have the form $(outs, a, ins)$, and
- the transitions of $\mathcal{T}(st)$ are
  - for internal actions a, all possible system transitions $(q_i)_{i \in I} \xrightarrow{(\{j\}, a, \emptyset)} (q_i')_{i \in I}$,
  - for non-internal actions a, exactly those system transitions $(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q_i')_{i \in I}$ of $\mathcal{S}$ such that $(outs, a, ins)$ **satisfies the synchronization type specification** $st(a)$.

Team Automata

Let $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$ be a component system and st be a synchronization type specification over $\mathcal{S}$.

st generates a so-called *team automaton*, denoted by $\mathcal{T}(st)$, over $\mathcal{S}$ such that

- the states of $\mathcal{T}(st)$ are the system states of $\mathcal{S}$, i.e. tuples $(q_i)_{i \in I}$ with q_i component states of $\mathcal{A}_i$ for all $i \in I$,
- the initial state of $\mathcal{T}(st)$ is $(q_i^0)_{i \in I}$ with q_i^0 the initial component state of $\mathcal{A}_i$ for all $i \in I$,
- the actions of $\mathcal{T}(st)$ have the form $(outs, a, ins)$, and
- the transitions of $\mathcal{T}(st)$ are
  - for internal actions a, all possible system transitions $(q_i)_{i \in I} \xrightarrow{(\{j\}, a, \emptyset)} (q'_i)_{i \in I}$,
  - for non-internal actions a, exactly those system transitions $(q_i)_{i \in I} \xrightarrow{(outs, a, ins)} (q'_i)_{i \in I}$ of $\mathcal{S}$ such that $(outs, a, ins)$ **satisfies the synchronization type specification** $st(a)$.

Familiar Synchronization Types

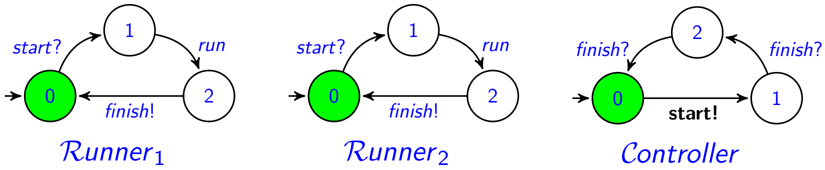
$1 \to 1$	binary, peer-to-peer communication
$[0,1] \to [0,1]$	non-blocking peer-to-peer (CCS)
$1 \to [0,*]$	multicast
$1 \to [1,*]$	strong multicast
$1 \to \#_{in}(a)$	strong broadcast where $\#_{in}(a)$ is the number of components which have a given action a as an input
$\#_{out}(a) \to \#_{in}(a)$	full synchronization (FSP) where $\#_{out}(a)$ is the number of components which have a given action a as an output
$[1,*] \to 1$	gathering
$[0,*] \to [0,*]$	all system transitions

Receptiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of receptiveness:
Whenever a group of components in the team is ready to send (simultaneously) a message a (in accordance with $st(a)$), then there should be components in the team which are ready to receive a (in accordance with $st(a)$).

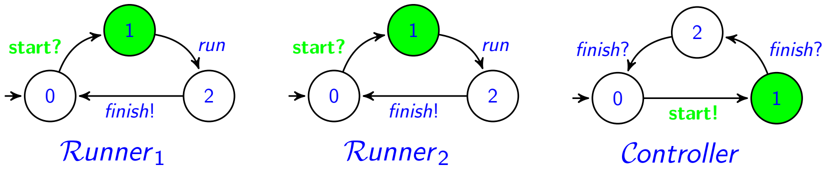
Example with synchronization types
$st(start) = 1 \rightarrow 2$ and $st(finish) = 1 \rightarrow 1$



Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of receptiveness:
Whenever a group of components in the team is ready to send (simultaneously) a message a (in accordance with $st(a)$), then there should be components in the team which are ready to receive a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

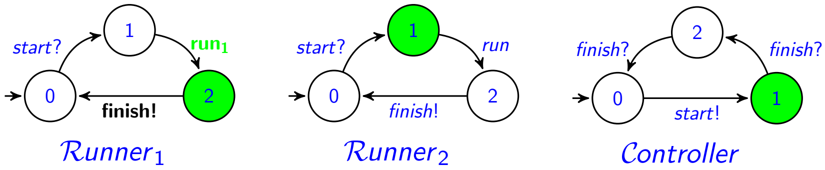


Receptiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of receptiveness:
Whenever a group of components in the team is ready to send (simultaneously) a message a (in accordance with $st(a)$), then there should be components in the team which are ready to receive a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

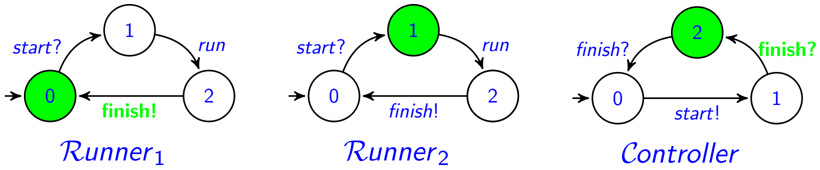


Receptiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of receptiveness:
Whenever a group of components in the team is ready to send (simultaneously) a message a (in accordance with $st(a)$), then there should be components in the team which are ready to receive a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

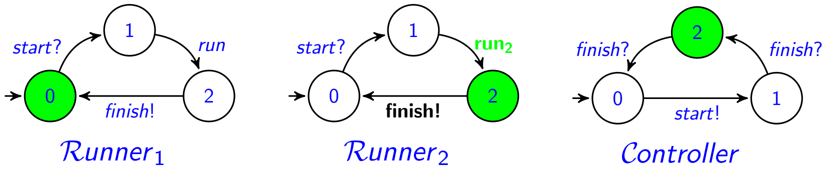


Receptiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{\, \mathcal{A}_i \mid i \in I \,\}$.

Idea of receptiveness:
Whenever a group of components in the team is ready to send (simultaneously) a message a (in accordance with $st(a)$), then there should be components in the team which are ready to receive a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

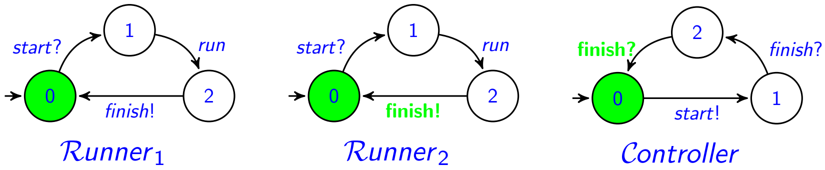


Receptiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{\, \mathcal{A}_i \mid i \in I \,\}$.

Idea of receptiveness:
Whenever a group of components in the team is ready to send (simultaneously) a message a (in accordance with $st(a)$), then there should be components in the team which are ready to receive a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

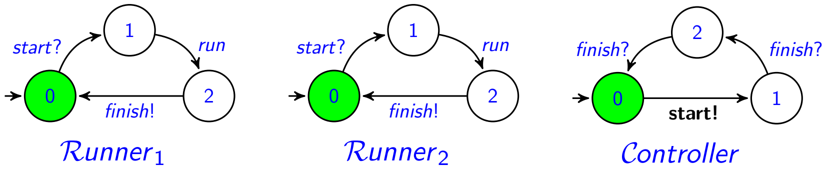


Receptiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of receptiveness:
Whenever a group of components in the team is ready to send (simultaneously) a message a (in accordance with $st(a)$), then there should be components in the team which are ready to receive a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$



Receptiveness Formally

For any reachable state $(q_i)_{i \in I}$ of $\mathcal{T}(st)$ and for any (non-internal) action a with $st(a) = [\mathtt{out}_1, \mathtt{out}_2] \to [\mathtt{in}_1, \mathtt{in}_2]$ we require:

If there is a group of components $\mathcal{G} = \{\, \mathcal{A}_j \mid j \in J \subseteq I \,\}$ having a as an output action such that

- a is enabled in each local state q_j with $j \in J$ and
- $\mathtt{out}_1 \leq |\mathcal{G}| \leq \mathtt{out}_2$

then there exists a group of components $\mathcal{H} = \{\, \mathcal{A}_k \mid k \in K \subseteq I \,\}$ having a as an input action such that

- a is enabled in each local state q_k with $k \in K$ and
- $\mathtt{in}_1 \leq |\mathcal{H}| \leq \mathtt{in}_2$.

Receptiveness Formally

For any reachable state $(q_i)_{i \in I}$ of $\mathcal{T}(st)$ and for any (non-internal) action a with $st(a) = [\mathtt{out}_1, \mathtt{out}_2] \to [\mathtt{in}_1, \mathtt{in}_2]$ we require:

If there is a group of components $\mathcal{G} = \{\, \mathcal{A}_j \mid j \in J \subseteq I \,\}$ having a as an output action such that

- a is enabled in each local state q_j with $j \in J$ and
- $\mathtt{out}_1 \leq |\mathcal{G}| \leq \mathtt{out}_2$

then there exists a group of components $\mathcal{H} = \{\, \mathcal{A}_k \mid k \in K \subseteq I \,\}$ having a as an input action such that

- a is enabled in each local state q_k with $k \in K$ and
- $\mathtt{in}_1 \leq |\mathcal{H}| \leq \mathtt{in}_2$.

Hence, the team $\mathcal{T}(st)$ can perform a transition

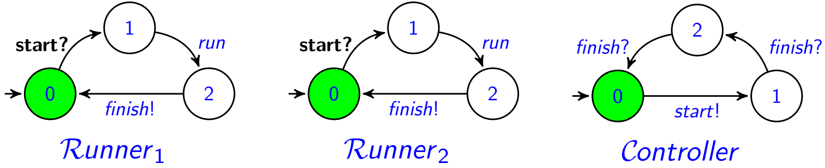
$$(q_i)_{i \in I} \xrightarrow{(J,a,K)} (q_i')_{i \in I}$$

Responsiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of responsiveness:
Whenever a group of components in the team waits to receive a message a (in accordance with $st(a)$), then there should be components in the team which are ready to send a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

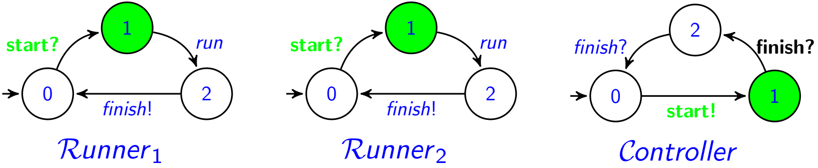


Responsiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of responsiveness:
Whenever a group of components in the team waits to receive a message a (in accordance with $st(a)$), then there should be components in the team which are ready to send a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \rightarrow 2$ and $st(finish) = 1 \rightarrow 1$

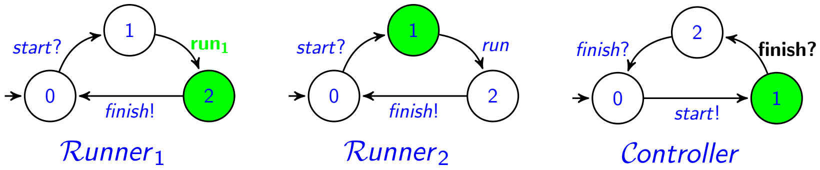


Responsiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{\, \mathcal{A}_i \mid i \in I \,\}$.

Idea of responsiveness:
Whenever a group of components in the team waits to receive a message a (in accordance with $st(a)$), then there should be components in the team which are ready to send a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

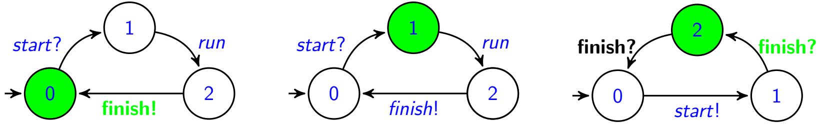


Responsiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of responsiveness:
Whenever a group of components in the team waits to receive a message a (in accordance with $st(a)$), then there should be components in the team which are ready to send a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$



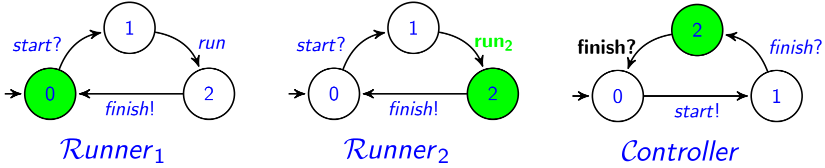
$\mathcal{R}unner_1$ $\mathcal{R}unner_2$ $\mathcal{C}ontroller$

Responsiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of responsiveness:
Whenever a group of components in the team waits to receive a message a (in accordance with $st(a)$), then there should be components in the team which are ready to send a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

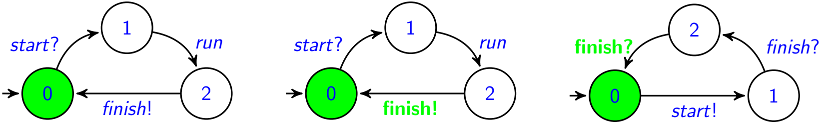


Responsiveness

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of responsiveness:
Whenever a group of components in the team waits to receive a message a (in accordance with $st(a)$), then there should be components in the team which are ready to send a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$

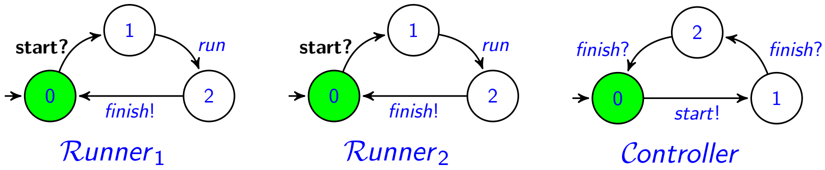


$\mathcal{R}unner_1$ $\mathcal{R}unner_2$ $\mathcal{C}ontroller$

Let $\mathcal{T}(st)$ be a team automaton generated by a synchronization type specification st over system $\mathcal{S} = \{ \mathcal{A}_i \mid i \in I \}$.

Idea of responsiveness:
Whenever a group of components in the team waits to receive a message a (in accordance with $st(a)$), then there should be components in the team which are ready to send a (in accordance with $st(a)$).

Example with synchronization types
$st(start) = 1 \to 2$ and $st(finish) = 1 \to 1$



Responsiveness Formally

For any reachable state $(q_i)_{i \in I}$ of $\mathcal{T}(st)$ and for any (non-internal) action a with $st(a) = [\mathtt{out}_1, \mathtt{out}_2] \to [\mathtt{in}_1, \mathtt{in}_2]$ we require:

If there is a group of components $\mathcal{G} = \{ \mathcal{A}_j \mid j \in J \subseteq I \}$ having a as an input action such that

- a is enabled in each local state q_j with $j \in J$ and
- $\mathtt{in}_1 \leq |\mathcal{G}| \leq \mathtt{in}_2$

then there exists a group of components $\mathcal{H} = \{ \mathcal{A}_k \mid k \in K \subseteq I \}$ having a as an output action such that

- a is enabled in each local state q_k with $k \in K$ and
- $\mathtt{out}_1 \leq |\mathcal{H}| \leq \mathtt{out}_2$.

Responsiveness Formally

For any reachable state $(q_i)_{i \in I}$ of $\mathcal{T}(st)$ and for any (non-internal) action a with $st(a) = [\mathtt{out}_1, \mathtt{out}_2] \to [\mathtt{in}_1, \mathtt{in}_2]$ we require:

If there is a group of components $\mathcal{G} = \{ \mathcal{A}_j \mid j \in J \subseteq I \}$ having a as an input action such that

- a is enabled in each local state q_j with $j \in J$ and
- $\mathtt{in}_1 \leq |\mathcal{G}| \leq \mathtt{in}_2$

then there exists a group of components $\mathcal{H} = \{ \mathcal{A}_k \mid k \in K \subseteq I \}$ having a as an output action such that

- a is enabled in each local state q_k with $k \in K$ and
- $\mathtt{out}_1 \leq |\mathcal{H}| \leq \mathtt{out}_2$.

Hence, the team $\mathcal{T}(st)$ can perform a transition

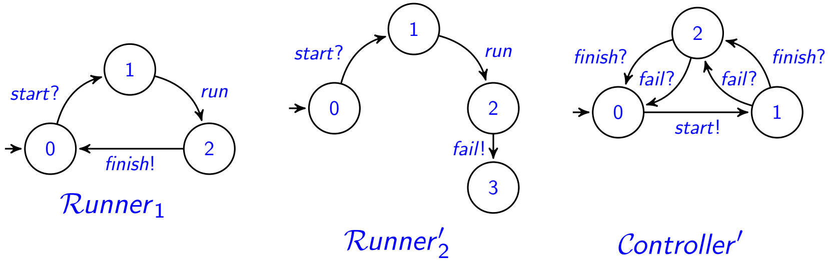
$$(q_i)_{i \in I} \xrightarrow{(K,a,J)} (q'_i)_{i \in I}$$

Responsiveness: Generalized Version

Idea:

If there is a choice of enabled input actions $a_1, \ldots, a_n$, it is sufficient if one of them is served.

Example with synchronization types
$st(start) = 1 \rightarrow 2$ and $st(finish) = 1 \rightarrow 1$



In state $(2, 3, 2)$ the controller has an input selection between *finish* and *fail*. Only for *finish* an input can be delivered (by the first runner) and this is fine.

Communication-Safety

A team automaton $\mathcal{T}(st)$ is *communication-safe* if it is receptive and responsive in all its reachable states.

This means, whenever a group of components in the team issues a request for communication it can successfully find partners in the team to join.

If partners can join only after execution of some intermediate actions the team $\mathcal{T}(st)$ is *weakly receptive* (*weakly responsive* respectively).

Example:

The runners/controller team is receptive and weakly responsive.

Specifications of Team Behaviour

Up to now there were given

- a set of component automata $\mathcal{A}_i$ ($i \in I$), and
- a synchronisation type spec. st (for the non-internal actions).

From this we have generated the team automaton $\mathcal{T}(st)$.

Now: We propose a top-down approach where *first the desired behaviour* of a team is specified (*requirements specification*) and only afterwards the component automata are designed to meet the requirements.

Example:

- No runner should begin running before she has been started by the controller.
- For any started runner it should be possible to finish her run.

Specifications of Team Behaviour

Up to now there were given

- a set of component automata $\mathcal{A}_i$ ($i \in I$), and
- a synchronisation type spec. st (for the non-internal actions).

From this we have generated the team automaton $\mathcal{T}(st)$.

Now: We propose a top-down approach where *first the desired behaviour* of a team is specified (*requirements specification*) and only afterwards the component automata are designed to meet the requirements.

Example:

- No runner should begin running before she has been started by the controller.
- For any started runner it should be possible to finish her run.

Formal Requirements Specification with Dynamic Logic

We assume given:

- a finite set I of component names,
- for each $i \in I$, disjoint finite sets of actions $\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}$ (to be supported by component i),
- a synchronisation type spec. st (for the non-internal actions).

Atomic team actions have the form $(outs, a, ins)$ where

- either: $outs = \{i\}, i \in I, ins = \emptyset$ and $a \in \Sigma_{i,int}$,
- or: $outs, ins \subseteq I$, $outs \cap ins \neq \emptyset$, and

  for all $i \in outs$: $a \in \Sigma_{i,out}$,
  for all $i \in ins$: $a \in \Sigma_{i,in}$, and

  if $st(a) = [\mathrm{out}_1, \mathrm{out}_2] \rightarrow [\mathrm{in}_1, \mathrm{in}_2]$ then
  $\mathrm{out}_1 \leq |outs| \leq \mathrm{out}_2$, $\mathrm{in}_1 \leq |ins| \leq \mathrm{out}_2$.

Formal Requirements Specification with Dynamic Logic

We assume given:

- a finite set I of component names,
- for each $i \in I$, disjoint finite sets of actions $\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}$ (to be supported by component i),
- a synchronisation type spec. st (for the non-internal actions).

Atomic team actions have the form $(outs, a, ins)$ where

- either: $outs = \{i\}, i \in I, ins = \emptyset$ and $a \in \Sigma_{i,int}$,
- or: $outs, ins \subseteq I$, $outs \cap ins \neq \emptyset$, and

  for all $i \in outs$: $a \in \Sigma_{i,out}$,
  for all $i \in ins$: $a \in \Sigma_{i,in}$, and

  if $st(a) = [\mathtt{out}_1, \mathtt{out}_2] \rightarrow [\mathtt{in}_1, \mathtt{in}_2]$ then
  $\mathtt{out}_1 \leq |outs| \leq \mathtt{out}_2$, $\mathtt{in}_1 \leq |ins| \leq \mathtt{out}_2$.

Example

$I = \{\text{Runner}_1, \text{Runner}_2, \text{Controller}\}$,

$\Sigma_{\text{Runner}_1,inp} = \Sigma_{\text{Runner}_2,inp} = \{start\} = \Sigma_{\text{Controller},out}$

$\Sigma_{\text{Runner}_1,out} = \Sigma_{\text{Runner}_2,out} = \{finish\} = \Sigma_{\text{Controller},inp}$

$\Sigma_{\text{Runner}_1,int} = \Sigma_{\text{Runner}_2,int} = \{run\}$

$\Sigma_{\text{Controller},int} = \emptyset$

Synchronization types:

$st(start) = 1 \to 2, \quad st(finish) = 1 \to 1$

Atomic team actions:

$(\{\text{Controller}\}, start, \{\text{Runner}_1, \text{Runner}_2\})$,

$(\{\text{Runner}_1\}, finish, \{\text{Controller}\})$, $(\{\text{Runner}_2\}, finish, \{\text{Controller}\})$,

$(\{\text{Runner}_1\}, run, \emptyset)$, $(\{\text{Runner}_2\}, run, \emptyset)$

Composed Actions

Composed actions are defined by the grammar

$$\alpha ::= \texttt{atomic action} \mid \alpha ; \alpha \mid \alpha + \alpha \mid \alpha^*$$

e.g. *sequential composition:*
$(\mathsf{Controller}, \mathit{start}, \{\mathsf{Runner}_1, \mathsf{Runner}_2\})\,;(\mathsf{Runner}_1, \mathit{run}, \emptyset)$

non-deterministic choice:
$(\mathsf{Runner}_1, \mathit{run}, \emptyset) \;+\; (\mathsf{Runner}_2, \mathit{finish}, \mathsf{Controller})$

iteration:
$(\mathbf{some}^*; (\mathsf{Runner}_2, \mathit{finish}, \mathsf{Controller}))^*$

Abbreviations:
Let $A = \{a_1, \ldots, a_n\}$ be the (finite) set of atomic team actions.

some stands for $a_1 + \ldots + a_n$ and, for $j \in \{1, \ldots, n\}$,

$-a_j$ stands for $a_1 + \ldots + a_{j-1} + a_{j+1} + \ldots + a_n$.

Dynamic Logic Formulas

Formulas: $\varphi ::= \textbf{true} \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\alpha\rangle\varphi$

$\langle\alpha\rangle\varphi$ expresses "in the current state it is possible to execute action α and after that φ holds"

Usual abbreviations: $\textbf{false} = \neg\textbf{true}$, $\varphi_1 \Rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$ and

$[\alpha]\varphi = \neg\langle\alpha\rangle\neg\varphi$ expresses "whenever α is executed in the current state then φ holds in the subsequent state"

e.g. a safety property is expressed by: $[\textbf{some}^*]\varphi$

a liveness property would be: $[\textbf{some}^*; a]\langle b\rangle\textbf{true}$

a forbidden behaviour would be: $[\textbf{some}^*; a; \textbf{some}^*; b]\textbf{false}$

deadlock-freeness would be: $[\textbf{some}^*]\langle\textbf{some}\rangle\textbf{true}$

Dynamic Logic Formulas

Formulas: $\varphi ::= \textbf{true} \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\alpha\rangle\varphi$

$\langle\alpha\rangle\varphi$ expresses "in the current state it is possible to execute action α and after that φ holds"

Usual abbreviations: $\textbf{false} = \neg\textbf{true}$, $\varphi_1 \Rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$ and

$[\alpha]\varphi = \neg\langle\alpha\rangle\neg\varphi$ expresses "whenever α is executed in the current state then φ holds in the subsequent state"

e.g. a safety property is expressed by: $[\textbf{some}^*]\varphi$

a liveness property would be: $[\textbf{some}^*; a]\langle b\rangle\textbf{true}$

a forbidden behaviour would be: $[\textbf{some}^*; a; \textbf{some}^*; b]\textbf{false}$

deadlock-freeness would be: $[\textbf{some}^*]\langle\textbf{some}\rangle\textbf{true}$

Example: Formal Requirements Specification

- No runner should begin running before she has been started by the controller:

  $[(-(\text{Controller}, \textit{start}, \{\text{Runner}_1, \text{Runner}_2\}))^*\,;$
  $(\text{Runner}_1, \textit{run}, \emptyset) + (\text{Runner}_2, \textit{run}, \emptyset)]\textbf{false}$

- For any started runner it should be possible to finish her run:

  $[\textbf{some}^*\,; (\text{Controller}, \textit{start}, \{\text{Runner}_1, \text{Runner}_2\})]$
  $(\langle\textbf{some}^*\,; (\text{Runner}_1, \textit{finish}, \text{Controller})\rangle\textbf{true} \wedge$
  $\langle(\textbf{some}^*\,; (\text{Runner}_2, \textit{finish}, \text{Controller})\rangle\textbf{true})$

Example: Formal Requirements Specification

- No runner should begin running before she has been started by the controller:

  $[(-(\mathsf{Controller}, \mathit{start}, \{\mathsf{Runner}_1, \mathsf{Runner}_2\}))^* \,;$
  $(\mathsf{Runner}_1, \mathit{run}, \emptyset) + (\mathsf{Runner}_2, \mathit{run}, \emptyset)]\mathbf{false}$

- For any started runner it should be possible to finish her run:

  $[\mathbf{some}^* \,; (\mathsf{Controller}, \mathit{start}, \{\mathsf{Runner}_1, \mathsf{Runner}_2\})]$
  $(\langle \mathbf{some}^* \,; (\mathsf{Runner}_1, \mathit{finish}, \mathsf{Controller})\rangle \mathbf{true} \wedge$
  $\langle (\mathbf{some}^* \,; (\mathsf{Runner}_2, \mathit{finish}, \mathsf{Controller})\rangle \mathbf{true})$

Semantics: Satisfaction of Dynamic Logic Formulas

We define $\mathcal{T}, q \models \varphi$ for

- a team automaton $\mathcal{T} = \mathcal{T}(st)$ generated over st,
- a system (team) state $q = (q_i)_{i \in I}$, and
- a dynamic logic formula φ.

- $\mathcal{T}, q \models \textbf{true}$,
- $\mathcal{T}, q \models \neg\varphi$ if **not** $\mathcal{T}(st), q \models \varphi$,
- $\mathcal{T}, q \models \varphi_1 \vee \varphi_2$ if $\mathcal{T}, q \models \varphi_1$ or $\mathcal{T}, q \models \varphi_2$,
- $\mathcal{T}, q \models \langle\alpha\rangle\varphi$ if there exists a team state q' and such that $q \xrightarrow{\alpha} q'$ and $\mathcal{T}, q' \models \varphi$,

$\mathcal{T}$ *satisfies* a formula φ, denoted by $\mathcal{T} \models \varphi$, if $\mathcal{T}, q^0 \models \varphi$ where $q^0 = (q_i^0)_{i \in I}$ is the initial system state.

$\mathcal{T}$ is a *correct realization* of a requirements specification $\{\varphi_1, \ldots, \varphi_m\}$, if $\mathcal{T} \models \varphi_j$ for all $j \in \{1, \ldots, m\}$.

Semantics: Satisfaction of Dynamic Logic Formulas

We define $\mathcal{T}, q \models \varphi$ for

- a team automaton $\mathcal{T} = \mathcal{T}(st)$ generated over st,
- a system (team) state $q = (q_i)_{i \in I}$, and
- a dynamic logic formula φ.

- $\mathcal{T}, q \models \textbf{true}$,
- $\mathcal{T}, q \models \neg\varphi$ if **not** $\mathcal{T}(st), q \models \varphi$,
- $\mathcal{T}, q \models \varphi_1 \vee \varphi_2$ if $\mathcal{T}, q \models \varphi_1$ or $\mathcal{T}, q \models \varphi_2$,
- $\mathcal{T}, q \models \langle\alpha\rangle\varphi$ if there exists a team state q' and such that $q \xrightarrow{\alpha} q'$ and $\mathcal{T}, q' \models \varphi$,

$\mathcal{T}$ *satisfies* a formula φ, denoted by $\mathcal{T} \models \varphi$,
if $\mathcal{T}, q^0 \models \varphi$ where $q^0 = (q_i^0)_{i \in I}$ is the initial system state.

$\mathcal{T}$ is a *correct realization* of a requirements specification $\{\varphi_1, \ldots, \varphi_m\}$, if $\mathcal{T} \models \varphi_j$ for all $j \in \{1, \ldots, m\}$.

Semantics: Satisfaction of Dynamic Logic Formulas

We define $\mathcal{T}, q \models \varphi$ for

- a team automaton $\mathcal{T} = \mathcal{T}(st)$ generated over st,
- a system (team) state $q = (q_i)_{i \in I}$, and
- a dynamic logic formula φ.

- $\mathcal{T}, q \models$ **true**,
- $\mathcal{T}, q \models \neg\varphi$ if **not** $\mathcal{T}(st), q \models \varphi$,
- $\mathcal{T}, q \models \varphi_1 \vee \varphi_2$ if $\mathcal{T}, q \models \varphi_1$ or $\mathcal{T}, q \models \varphi_2$,
- $\mathcal{T}, q \models \langle\alpha\rangle\varphi$ if there exists a team state q' and such that $q \xrightarrow{\alpha} q'$ and $\mathcal{T}, q' \models \varphi$,

$\mathcal{T}$ *satisfies* a formula φ, denoted by $\mathcal{T} \models \varphi$, if $\mathcal{T}, q^0 \models \varphi$ where $q^0 = (q_i^0)_{i \in I}$ is the initial system state.

$\mathcal{T}$ is a *correct realization* of a requirements specification $\{\varphi_1, \dots, \varphi_m\}$, if $\mathcal{T} \models \varphi_j$ for all $j \in \{1, \dots, m\}$.

State Transitions for Composed Actions

Definition of $q \xrightarrow{\alpha} q'$ by structural induction on the form of α:

for $\alpha = (\mathit{outs}, a, \mathit{ins})$: $q \xrightarrow{\alpha} q'$ is the team transition defined earlier,

for $\alpha = \alpha_1 ; \alpha_2$: $q \xrightarrow{\alpha} q'$ holds if there are $q \xrightarrow{\alpha_1} \hat{q}$ and $\hat{q} \xrightarrow{\alpha_2} q'$,

for $\alpha = \alpha_1 + \alpha_2$: $q \xrightarrow{\alpha} q'$ holds if there is $q \xrightarrow{\alpha_1} q'$ or $q \xrightarrow{\alpha_2} q'$,

for $\alpha = (\alpha_1)^*$: $q \xrightarrow{\alpha} q'$ holds if $q = q'$ or if there are $q \xrightarrow{(\alpha_1)^*} \hat{q}$ and $\hat{q} \xrightarrow{\alpha_1} q'$.

Conclusion

- Investigation of compatibility notions for various synchronization types.
- A team automaton is a correct realization of a dynamic logic specification, if it satisfies all its formulas.
- Further steps:
  - composition of teams,
  - asynchronous communication,
  - tool support,
  - team automata with variable instantiations (featured team automata) for product lines of component systems [ter Beek, Cledou, Hennicker, Proença 2021]

Some Literature

M. H. ter Beek, C. A. Ellis, J. Kleijn, and G. Rozenberg: Synchronizations in Team Automata for Groupware Systems. Comput. Sup. Coop. Work 12(1), pages 21-69, 2003.

M. H. ter Beek, R. Hennicker, J. Kleijn: Compositionality of Safe Communication in Systems of Team Automata. In *Proc. Int. Coll. Theor. Aspects of Computing (ICTAC'20)*, volume 12545 of *Lecture Notes in Computer Science*, pages 200-220, Springer, 2020.

M. H. ter Beek, G. Cledou, R. Hennicker, and J. Proença: Featured Team Automata. In *Proc. Formal Methods - 24th International Symposium (FM'21)*, volume 13047 of *Lecture Notes in Computer Science*, pages 483-502, Springer, 2021.