Modelling and Analysis of Collective Adaptive Systems

Michele Loreti

Outline

Outline

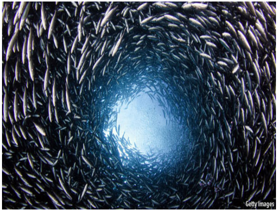
Collective Systems

We are surrounded by examples of collective systems:

Collective Systems

We are surrounded by examples of collective systems:

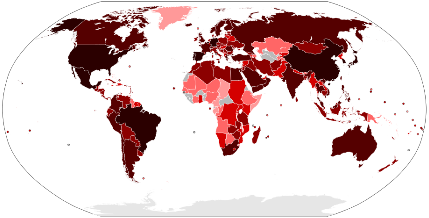
in the natural world ....



Collective Systems

We are surrounded by examples of collective systems:

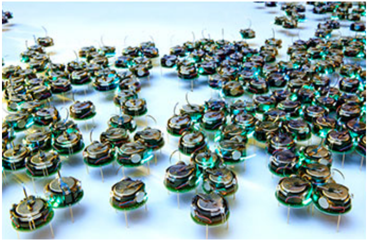
.... and in the man-made world



Collective Systems

We are surrounded by examples of collective systems:

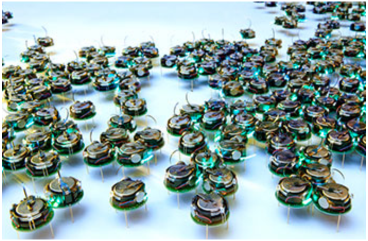
.... and in the man-made world



Collective Systems

We are surrounded by examples of collective systems:

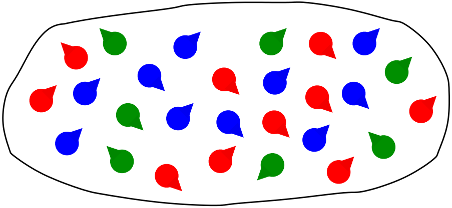
.... and in the man-made world



Most of these systems are also adaptive to their environment

Collective Adaptive Systems

From a computer science perspective these systems can be viewed as being made up of a large number of interacting entities.

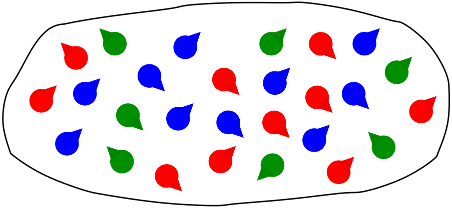


Each entity may have its own properties, objectives and actions.

At the system level these combine to create the collective behaviour.

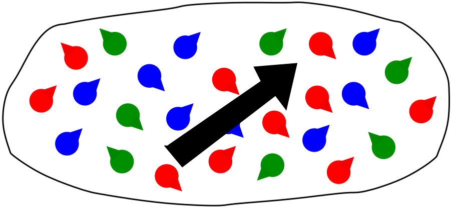
Collective Adaptive Systems

The behaviour of the system is thus dependent on the behaviour of the individual entities.



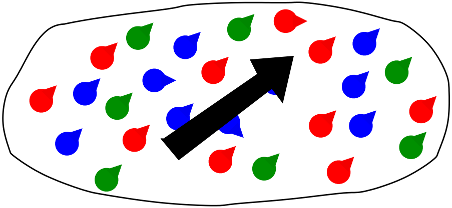
Collective Adaptive Systems

The behaviour of the system is thus dependent on the behaviour of the individual entities.



Collective Adaptive Systems

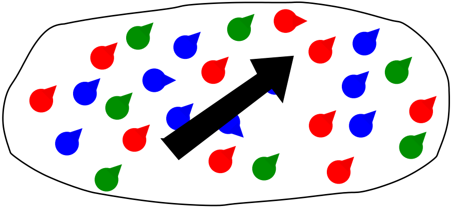
The behaviour of the system is thus dependent on the behaviour of the individual entities.



And the behaviour of the individuals will be influenced by the state of the overall system.

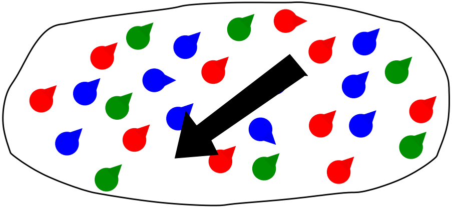
Collective Adaptive Systems

Such systems are often embedded in our environment and need to operate without centralised control or direction.



Collective Adaptive Systems

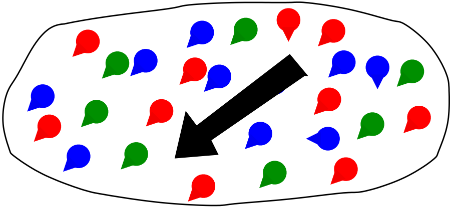
Such systems are often embedded in our environment and need to operate without centralised control or direction.



Moreover when conditions within the system change it may not be feasible to have human intervention to adjust behaviour appropriately.

Collective Adaptive Systems

Such systems are often embedded in our environment and need to operate without centralised control or direction.



Moreover when conditions within the system change it may not be feasible to have human intervention to adjust behaviour appropriately.

Thus systems must be able to autonomously adapt.

The Informatic Environment

Robin Milner coined the term of informatics environment — pervasive computing elements embedded in the human environment, invisibly providing services and responding to requirements.

The Informatic Environment

Robin Milner coined the term of informatics environment — pervasive computing elements embedded in the human environment, invisibly providing services and responding to requirements.

The Informatic Environment

Robin Milner coined the term of informatics environment — pervasive computing elements embedded in the human environment, invisibly providing services and responding to requirements.

Such systems are now becoming the reality, and many form collective adaptive systems, in which large numbers of computing elements collaborate to meet the human need.

The Informatic Environment

Robin Milner coined the term of informatics environment — pervasive computing elements embedded in the human environment, invisibly providing services and responding to requirements.

Such systems are now becoming the reality, and many form collective adaptive systems, in which large numbers of computing elements collaborate to meet the human need.

For instance, may examples of such systems can be found in components of Smart Cities, such as smart urban transport and smart grid electricity generation and storage.

Outline

Challenges for modelling CAS

Work over the last twenty years on stochastic process algebra provides a solid basic framework for modelling CAS but there remain a number of challenges:

- Richer forms of interaction
- The influence of space on behaviour
- Capturing adaptivity

Richer forms of interaction

If we consider real collective adaptive systems, especially those with emergent behaviour, they embody rich forms of interaction, often based on asynchronous communication.

Richer forms of interaction

If we consider real collective adaptive systems, especially those with emergent behaviour, they embody rich forms of interaction, often based on asynchronous communication.

For example, pheromone trails left by social insects.

Richer forms of interaction

If we consider real collective adaptive systems, especially those with emergent behaviour, they embody rich forms of interaction, often based on asynchronous communication.

For example, pheromone trails left by social insects.

Languages like SCEL offer these richer communication patterns, with components which include a knowledge store which can be manipulated by other components and attribute-based communication.

R.De Nicola, G.Ferrari, M.Loreti, R.Pugliese. A Language-Based Approach to Autonomic Computing. FMCO 2011.

Richer forms of interaction

If we consider real collective adaptive systems, especially those with emergent behaviour, they embody rich forms of interaction, often based on asynchronous communication.

For example, pheromone trails left by social insects.

Languages like SCEL offer these richer communication patterns, with components which include a knowledge store which can be manipulated by other components and attribute-based communication.

R.De Nicola, G.Ferrari, M.Loreti, R.Pugliese. A Language-Based Approach to Autonomic Computing. FMCO 2011.

But languages designed for other purposes typically contain too much detail to be used as the basis of quantitative modelling and analysis.

Modelling space

Location and movement play an important role within many CAS, e.g. smart cities.

Modelling space

Location and movement play an important role within many CAS, e.g. smart cities.

We can impose the effects of space by encoding it into the behaviour of the actions of components and distinguishing the same component in different location as distinct types, but this is modelling space implicitly.

Modelling space

Location and movement play an important role within many CAS, e.g. smart cities.

We can impose the effects of space by encoding it into the behaviour of the actions of components and distinguishing the same component in different location as distinct types, but this is modelling space implicitly.

It is preferable to model space explicitly although this poses significant challenges both for model expression and model solution.

Modelling space

Location and movement play an important role within many CAS, e.g. smart cities.

We can impose the effects of space by encoding it into the behaviour of the actions of components and distinguishing the same component in different location as distinct types, but this is modelling space implicitly.

It is preferable to model space explicitly although this poses significant challenges both for model expression and model solution.

There is a tension with scalable analysis which is often based on an implicit assumption that all components are co-located.

Capturing adaptivity

Existing process algebras, tend to work with a fixed set of actions for each entity type.

Capturing adaptivity

Existing process algebras, tend to work with a fixed set of actions for each entity type.

Some stochastic process algebras allow the rate of activity to be dependent on the state of the system.

Capturing adaptivity

Existing process algebras, tend to work with a fixed set of actions for each entity type.

Some stochastic process algebras allow the rate of activity to be dependent on the state of the system.

But for truly adaptive systems there should also be some way to identify the goal or objective of an entity in addition to its behaviour.

Outline

A new language for CAS

The QUANTICOL project seeks to develop a coherent, integrated set of linguistic primitives, methods and tools to build systems that can operate in open-ended, unpredictable environments.

A new language for CAS

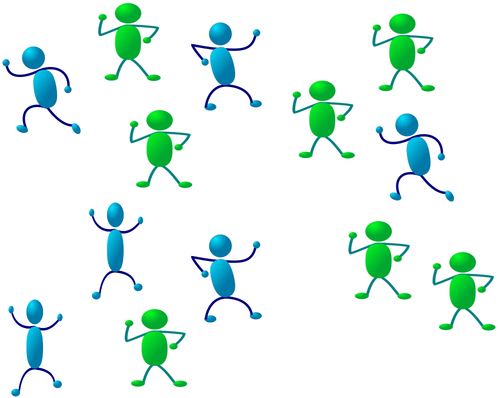
The QUANTICOL project seeks to develop a coherent, integrated set of linguistic primitives, methods and tools to build systems that can operate in open-ended, unpredictable environments.

A new language for CAS

The QUANTICOL project seeks to develop a coherent, integrated set of linguistic primitives, methods and tools to build systems that can operate in open-ended, unpredictable environments.

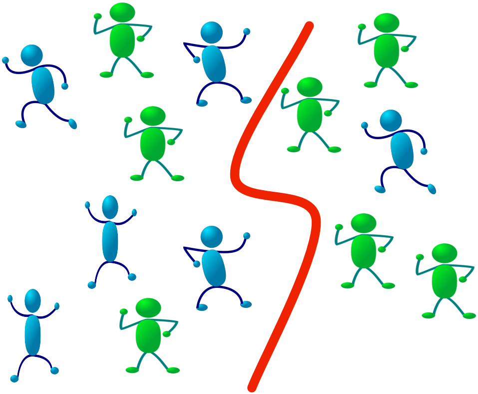
A new language for CAS

The QUANTICOL project seeks to develop a coherent, integrated set of linguistic primitives, methods and tools to build systems that can operate in open-ended, unpredictable environments.



A new language for CAS

The QUANTICOL project seeks to develop a coherent, integrated set of linguistic primitives, methods and tools to build systems that can operate in open-ended, unpredictable environments.



CARMA

A key element of the QUANTICOL framework is the language, CARMA (Collective Adaptive Resource-sharing Markovian Agents), which handles:

CARMA

A key element of the QUANTICOL framework is the language, CARMA (Collective Adaptive Resource-sharing Markovian Agents), which handles:

1. The behaviours of agents and their interactions;

CARMA

A key element of the QUANTICOL framework is the language, CARMA (Collective Adaptive Resource-sharing Markovian Agents), which handles:

1. The behaviours of agents and their interactions;
2. The global knowledge of the system and that of its agents;

CARMA

A key element of the QUANTICOL framework is the language, CARMA (Collective Adaptive Resource-sharing Markovian Agents), which handles:

1. The behaviours of agents and their interactions;
2. The global knowledge of the system and that of its agents;
3. The environment where agents operate...

CARMA

A key element of the QUANTICOL framework is the language, CARMA (Collective Adaptive Resource-sharing Markovian Agents), which handles:

1. The behaviours of agents and their interactions;
2. The global knowledge of the system and that of its agents;
3. The environment where agents operate. . .
   - taking into account open ended-ness and adaptation;

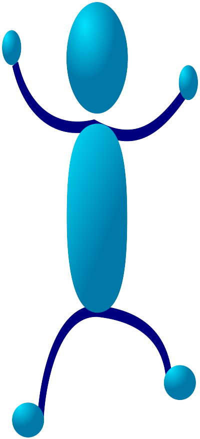
CARMA

A key element of the QUANTICOL framework is the language, CARMA (Collective Adaptive Resource-sharing Markovian Agents), which handles:

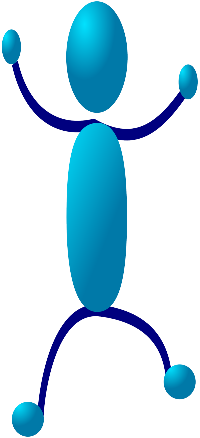
1. The behaviours of agents and their interactions;
2. The global knowledge of the system and that of its agents;
3. The environment where agents operate...
   - taking into account open ended-ness and adaptation;
   - taking into account resources, locations and visibility/reachability issues.

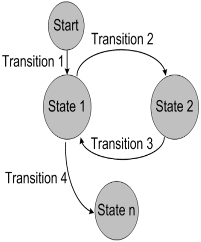
M.Loreti et al. CARMA: Collective Adaptive Resource-sharing Markovian Agents. QAPL 2015.

Agents in CARMA

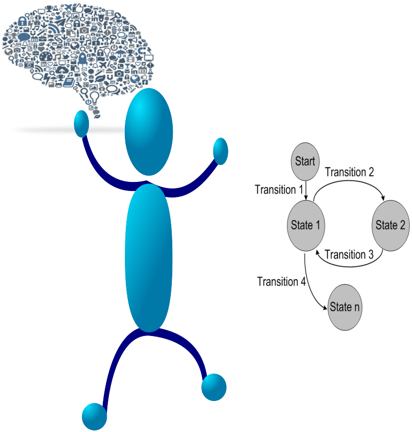


Agents in CARMA

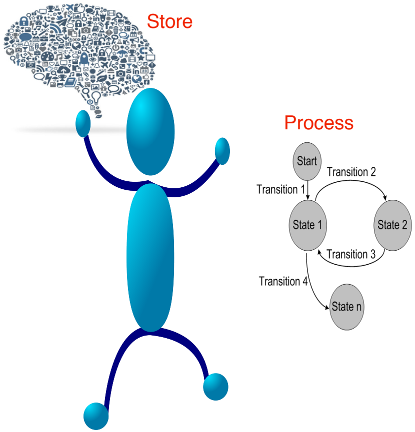




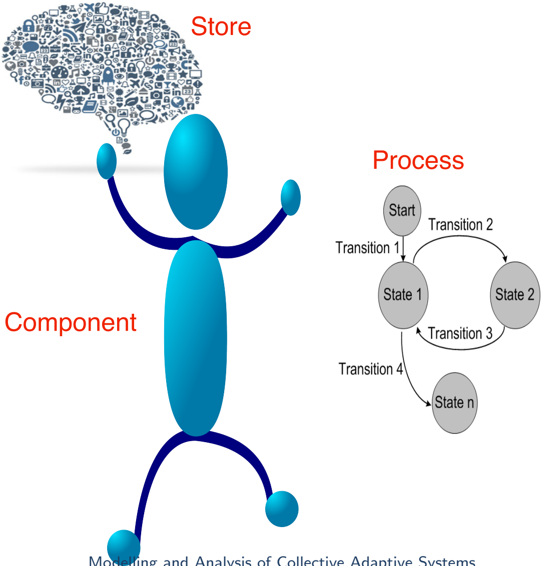
Agents in CARMA



Agents in CARMA



Components


Store
Component
Process
Start
Transition 1
Transition 2
State 1
State 2
Transition 3
Transition 4
State n

Components

Agents in CARMA are defined as components C of the form (P, γ) where...

- P is a process, representing agent behaviour;
- γ is a store, modelling agent knowledge.

Components

Agents in CARMA are defined as components C of the form (P, γ) where...

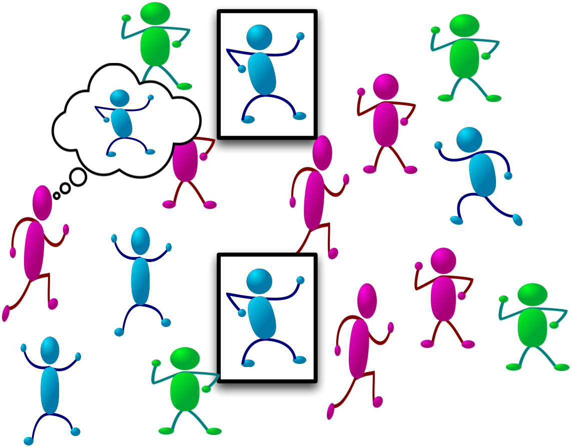
- P is a process, representing agent behaviour;
- γ is a store, modelling agent knowledge.

The participants of an interaction are identified via predicates...

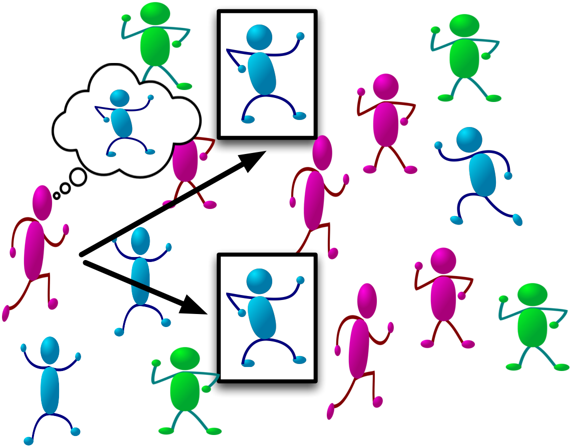
- the counterpart of a communication is selected according its properties

Attribute-based communication

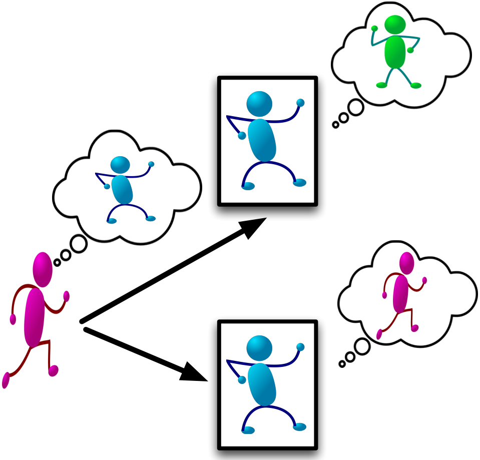
Attribute-based communication



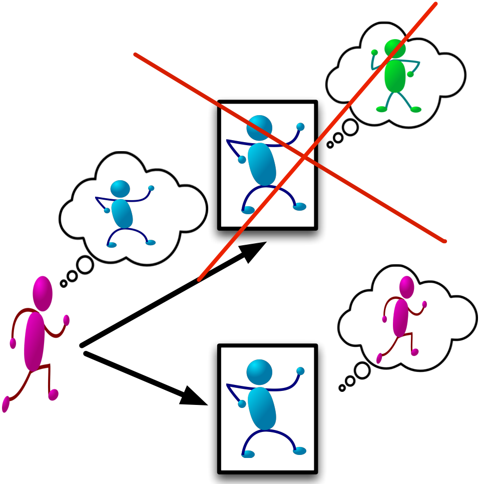
Attribute-based communication



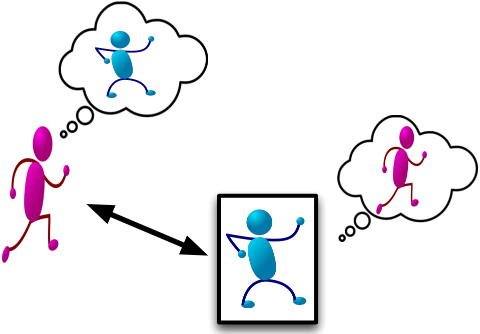
Attribute-based communication



Attribute-based communication



Attribute-based communication



Interaction primitives

Processes interact via attribute based communications...

Interaction primitives

Processes interact via attribute based communications...

- **Broadcast output**: a message is sent to all the components satisfying a predicate π;

Interaction primitives

Processes interact via attribute based communications. . .

- **Broadcast output**: a message is sent to all the components satisfying a predicate π;
- **Broadcast input**: a process is willing to receive a broadcast message from a component satisfying a predicate π;

Interaction primitives

Processes interact via attribute based communications. . .

- **Broadcast output**: a message is sent to all the components satisfying a predicate π;
- **Broadcast input**: a process is willing to receive a broadcast message from a component satisfying a predicate π;
- **Unicast output**: a message is sent to one of the components satisfying a predicate π;

Interaction primitives

Processes interact via attribute based communications...

- **Broadcast output**: a message is sent to all the components satisfying a predicate π;
- **Broadcast input**: a process is willing to receive a broadcast message from a component satisfying a predicate π;
- **Unicast output**: a message is sent to one of the components satisfying a predicate π;
- **Unicast input**: a process is willing to receive a message from a component satisfying a predicate π.

Interaction primitives

Processes interact via attribute based communications...

- **Broadcast output**: a message is sent to all the components satisfying a predicate π;
- **Broadcast input**: a process is willing to receive a broadcast message from a component satisfying a predicate π;
- **Unicast output**: a message is sent to one of the components satisfying a predicate π;
- **Unicast input**: a process is willing to receive a message from a component satisfying a predicate π.

The execution of an action takes an exponentially distributed time; the rate of each action is determined by the environment.

Interaction primitives

Syntax

$$
\begin{array}{rcll}
act & ::= & \alpha^{\star}[\pi]\langle \overrightarrow{e} \rangle \sigma & \text{Broadcast output} \\
 & \mid & \alpha^{\star}[\pi](\overrightarrow{x})\sigma & \text{Broadcast input} \\
 & \mid & \alpha[\pi]\langle \overrightarrow{e} \rangle \sigma & \text{Unicast output} \\
 & \mid & \alpha[\pi](\overrightarrow{x})\sigma & \text{Unicast input}
\end{array}
$$

Interaction primitives
Syntax

act	$::=$	$\alpha^{\star}[\pi]\langle\overrightarrow{e}\rangle\sigma$	Broadcast output
	$\mid$	$\alpha^{\star}[\pi](\overrightarrow{x})\sigma$	Broadcast input
	$\mid$	$\alpha[\pi]\langle\overrightarrow{e}\rangle\sigma$	Unicast output
	$\mid$	$\alpha[\pi](\overrightarrow{x})\sigma$	Unicast input

- α is an action type;

Interaction primitives

Syntax

$$
\begin{array}{rcll}
act & ::= & \alpha^{\star}[\pi]\langle \overrightarrow{e} \rangle \sigma & \text{Broadcast output} \\
& | & \alpha^{\star}[\pi](\overrightarrow{x})\sigma & \text{Broadcast input} \\
& | & \alpha[\pi]\langle \overrightarrow{e} \rangle \sigma & \text{Unicast output} \\
& | & \alpha[\pi](\overrightarrow{x})\sigma & \text{Unicast input}
\end{array}
$$

- α is an action type;
- π is a predicate;

Interaction primitives
Syntax

$$
\begin{array}{lcll}
act & ::= & \alpha^{\star}[\pi]\langle\overrightarrow{e}\rangle\sigma & \text{Broadcast output} \\
 & \mid & \alpha^{\star}[\pi](\overrightarrow{x})\sigma & \text{Broadcast input} \\
 & \mid & \alpha[\pi]\langle\overrightarrow{e}\rangle\sigma & \text{Unicast output} \\
 & \mid & \alpha[\pi](\overrightarrow{x})\sigma & \text{Unicast input}
\end{array}
$$

- α is an action type;
- π is a predicate;
- σ is the effect of the action on the store.

Simple example

$$\begin{array}{rcl} \mathit{Lecturer} & = & \mathit{teach}^{\star}[\mathit{awake} = \mathit{true}]\langle \mathit{fact} \rangle \{\} \ldots \\ & + & \mathit{coffee}^{\star}[\mathit{true}]\langle \cdot \rangle \{\mathit{boring} := \mathit{false}\} \ldots \end{array}$$

$$\begin{array}{rcl} \mathit{Student} & = & \mathit{teach}^{\star}[\mathit{boring} = \mathit{false}](\mathit{fact})\{\mathit{know} := \mathit{know} + \mathit{fact}\} \ldots \\ & + & \mathit{coffee}^{\star}[\mathit{true}](\cdot)\{\mathit{awake} := \mathit{true}\} \ldots \end{array}$$

Simple example

$$
\begin{aligned}
\mathit{Lecturer} &= \mathit{teach}^{\star}[\mathit{awake} = \mathit{true}]\langle \mathit{fact}\rangle\{\}\ldots \\
&+ \mathit{coffee}^{\star}[\mathit{true}]\langle\cdot\rangle\{\mathit{boring} := \mathit{false}\}\ldots
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Student} &= \mathit{teach}^{\star}[\mathit{boring} = \mathit{false}](\mathit{fact})\{\mathit{know} := \mathit{know} + \mathit{fact}\}\ldots \\
&+ \mathit{coffee}^{\star}[\mathit{true}](\cdot)\{\mathit{awake} := \mathit{true}\}\ldots
\end{aligned}
$$

Simple example

$$
\begin{aligned}
Lecturer &= teach^{\star}[awake = true]\langle fact\rangle\{\}\ldots \\
&+ coffee^{\star}[true]\langle\cdot\rangle\{boring := false\}\ldots
\end{aligned}
$$

$$
\begin{aligned}
Student &= teach^{\star}[boring = false](fact)\{know := know + fact\}\ldots \\
&+ coffee^{\star}[true](\cdot)\{awake := true\}\ldots
\end{aligned}
$$

Simple example

$$
\begin{aligned}
\mathit{Lecturer} &= \mathit{teach}^{\star}[\mathit{awake} = \mathit{true}]\langle \mathit{fact} \rangle \{\} \ldots \\
&+ \mathit{coffee}^{\star}[\mathit{true}]\langle \cdot \rangle \{\mathit{boring} := \mathit{false}\} \ldots \\
\\
\mathit{Student} &= \mathit{teach}^{\star}[\mathit{boring} = \mathit{false}](\mathit{fact})\{\mathit{know} := \mathit{know} + \mathit{fact}\} \ldots \\
&+ \mathit{coffee}^{\star}[\mathit{true}](\cdot)\{\mathit{awake} := \mathit{true}\} \ldots
\end{aligned}
$$

Simple example

$$\begin{array}{rcl} \mathit{Lecturer} & = & \mathit{teach}^{\star}[\mathit{awake} = \mathit{true}]\langle \mathit{fact} \rangle \{\} \ldots \\ & + & \mathit{coffee}^{\star}[\mathit{true}]\langle \cdot \rangle \{\mathit{boring} := \mathit{false}\} \ldots \end{array}$$

$$\begin{array}{rcl} \mathit{Student} & = & \mathit{teach}^{\star}[\mathit{boring} = \mathit{false}](\mathit{fact})\{\mathit{know} := \mathit{know} + \mathit{fact}\} \ldots \\ & + & \mathit{coffee}^{\star}[\mathit{true}](\cdot)\{\mathit{awake} := \mathit{true}\} \ldots \end{array}$$

(5 minutes)

Interaction patterns in CAS

Typically, CAS exhibit two kinds of interaction pattern:

Interaction patterns in CAS

Typically, CAS exhibit two kinds of interaction pattern:

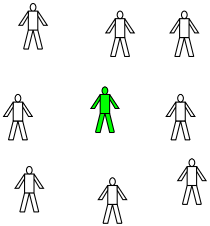
1. **Spreading**: one agent spreads relevant information to a given group of other agents

Interaction patterns in CAS

Typically, CAS exhibit two kinds of interaction pattern:

1. **Spreading**: one agent spreads relevant information to a given group of other agents

Spreading: 1-to-many

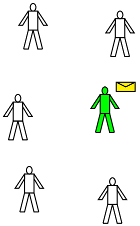


Interaction patterns in CAS

Typically, CAS exhibit two kinds of interaction pattern:

1. **Spreading**: one agent spreads relevant information to a given group of other agents

Spreading: 1-to-many

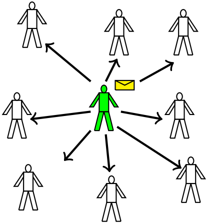


Interaction patterns in CAS

Typically, CAS exhibit two kinds of interaction pattern:

1. **Spreading**: one agent spreads relevant information to a given group of other agents

Spreading: 1-to-many

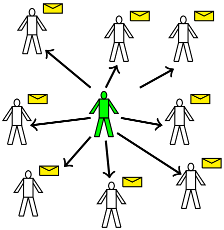


Interaction patterns in CAS

Typically, CAS exhibit two kinds of interaction pattern:

1. **Spreading**: one agent spreads relevant information to a given group of other agents

Spreading: 1-to-many

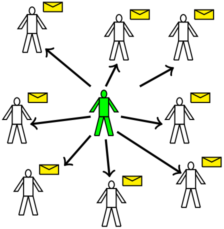


Interaction patterns in CAS

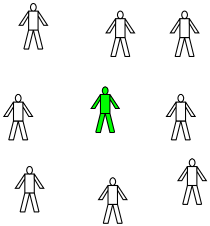
Typically, CAS exhibit two kinds of interaction pattern:

1. **Spreading**: one agent spreads relevant information to a given group of other agents
2. **Collecting**: one agent changes its behaviour according to data collected from one agent belonging to a given group of agents.

Spreading: 1-to-many



Collecting: 1-to-1

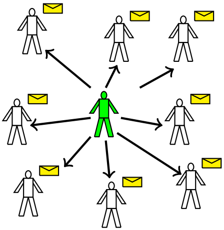


Interaction patterns in CAS

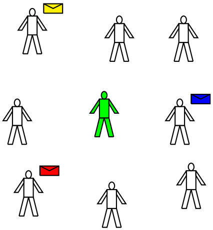
Typically, CAS exhibit two kinds of interaction pattern:

1. **Spreading**: one agent spreads relevant information to a given group of other agents
2. **Collecting**: one agent changes its behaviour according to data collected from one agent belonging to a given group of agents.

Spreading: 1-to-many



Collecting: 1-to-1

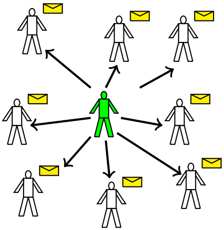


Interaction patterns in CAS

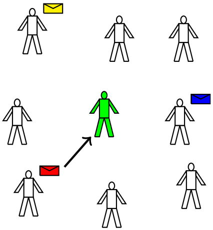
Typically, CAS exhibit two kinds of interaction pattern:

1. **Spreading**: one agent spreads relevant information to a given group of other agents
2. **Collecting**: one agent changes its behaviour according to data collected from one agent belonging to a given group of agents.

Spreading: 1-to-many



Collecting: 1-to-1

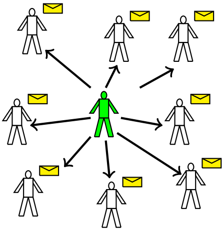


Interaction patterns in CAS

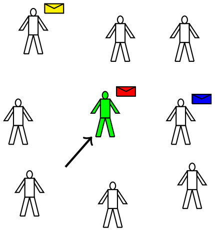
Typically, CAS exhibit two kinds of interaction pattern:

1. **Spreading**: one agent spreads relevant information to a given group of other agents
2. **Collecting**: one agent changes its behaviour according to data collected from one agent belonging to a given group of agents.

Spreading: 1-to-many

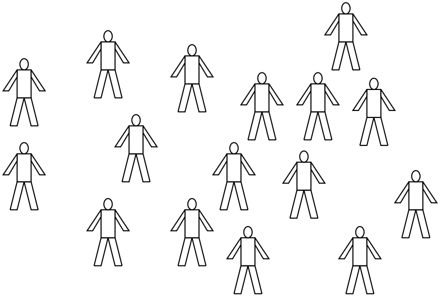


Collecting: 1-to-1



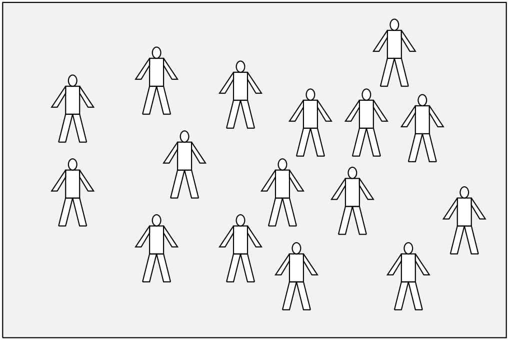
CAS: CARMA perspective

Collective



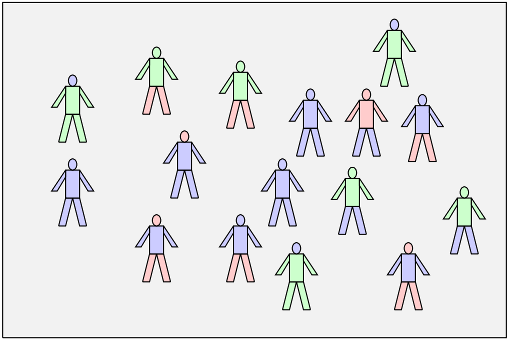
CAS: CARMA perspective

Collective **Environment**



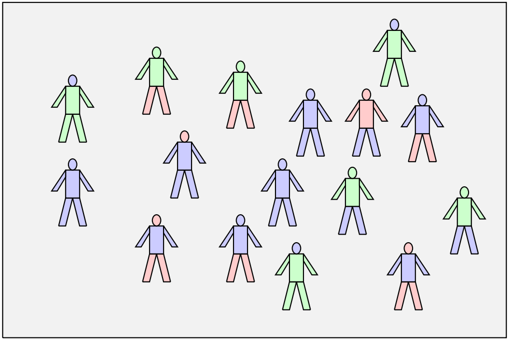
CAS: CARMA perspective

Collective **Environment** **Attributes**



CAS: CARMA perspective

Collective **Environment** **Attributes**



Processes are referenced via their attributes!

CARMA

A CARMA system consists of

A Carma system consists of

- a collective (N)...

CARMA

A CARMA system consists of

- a collective (N)...
- ...operating in an environment ($\mathscr{E}$).

CARMA

A CARMA system consists of

- a collective (N)...
- ...operating in an environment ($\mathscr{E}$).

Collective...

- is composed by a set of components, i.e. the Markovian agents that compete and/or cooperate to achieve a set of given tasks
- models the behavioural part of a system

CARMA

A CARMA system consists of

- a collective (N)...
- ...operating in an environment ($\mathscr{E}$).

Collective...

- is composed by a set of components, i.e. the Markovian agents that compete and/or cooperate to achieve a set of given tasks
- models the behavioural part of a system

Environment...

- models the rules intrinsic to the context where agents operate;
- mediates and regulates agent interactions.

Components

Agents in CARMA are defined as components C of the form (P, γ) where...

- P is a process, representing agent behaviour;
- γ is a store, modelling agent knowledge.

Components

Agents in CARMA are defined as components C of the form (P, γ) where...

- P is a process, representing agent behaviour;
- γ is a store, modelling agent knowledge.

The participants of an interaction are identified via predicates...

- the counterpart of a communication is selected according its properties

Interaction primitives
Syntax

act	$::=$	$\alpha^{\star}[\pi]\langle\overrightarrow{e}\rangle\sigma$	Broadcast output
	$\mid$	$\alpha^{\star}[\pi](\overrightarrow{x})\sigma$	Broadcast input
	$\mid$	$\alpha[\pi]\langle\overrightarrow{e}\rangle\sigma$	Unicast output
	$\mid$	$\alpha[\pi](\overrightarrow{x})\sigma$	Unicast input

- α is an action type;
- π is a predicate;
- σ is the effect of the action on the store.

Updating the store

After the execution of an action, a process can update the component store:

- σ denotes a function mapping each γ to a probability distribution over possible stores.

Updating the store

After the execution of an action, a process can update the component store:

- σ denotes a function mapping each γ to a probability distribution over possible stores.

$$\mathrm{move}^{\star}[\pi]\langle v\rangle\{x := x + U(-1,+1)\}$$

Updating the store

After the execution of an action, a process can update the component store:

- σ denotes a function mapping each γ to a probability distribution over possible stores.

$$\mathrm{move}^{\star}[\pi]\langle v\rangle\{x := x + U(-1,+1)\}$$

Remark:

- Processes running in the same component can implicitly interact via the local store;
- Updates are instantaneous.

More on synchronisation

Predicates regulating broadcast/unicast inputs can refer also to the received values.

More on synchronisation

Predicates regulating broadcast/unicast inputs can refer also to the received values.

Example:

A value greater than 0 is expected from a component with a *trust_level* less than 3:

$$\alpha^{\star}[(x > 0) \wedge (trust\_level < 3)](x)\sigma.P$$

Examples of interactions. . .

Broadcast synchronisation:

$$
\begin{array}{l}
(\ \mathsf{stop}^{\star}[\mathsf{bl} < 5\%]\langle v\rangle\sigma_1.P\ ,\{\mathit{role} = \text{“}\mathit{master}\text{”}\})\ \|\\
\quad(\ \mathsf{stop}^{\star}[\mathsf{role} = \text{“}\mathit{master}\text{”}](x)\sigma_2\ .Q_1\ ,\{\mathsf{bl} = 4\%\})\ \|\\
\qquad(\ \mathsf{stop}^{\star}[\mathsf{role} = \text{“}\mathit{super}\text{”}](x)\sigma_3.Q_2\ ,\{\mathsf{bl} = 2\%\})\ \|\\
\qquad\qquad\qquad(\ \mathsf{stop}^{\star}[\top](x)\sigma_4.Q_3\ ,\{\mathsf{bl} = 2\%\})
\end{array}
$$

Examples of interactions...

Broadcast synchronisation:

$$
\begin{array}{l}
(\ \mathsf{stop}^{\star}[\mathsf{bl} < 5\%]\langle v\rangle\sigma_1.P\ ,\{role = \text{``}master\text{''}\})\ \|\\
\quad (\ \mathsf{stop}^{\star}[\mathsf{role} = \text{``}master\text{''}](x)\sigma_2\ .Q_1\ ,\{\mathsf{bl} = 4\%\})\ \|\\
\qquad (\ \mathsf{stop}^{\star}[\mathsf{role} = \text{``}super\text{''}](x)\sigma_3.Q_2\ ,\{\mathsf{bl} = 2\%\})\ \|\\
\qquad\qquad (\ \mathsf{stop}^{\star}[\top](x)\sigma_4.Q_3\ ,\{\mathsf{bl} = 2\%\})
\end{array}
$$

Examples of interactions. . .

Broadcast synchronisation:

$$(\ \mathsf{stop}^{\star}[\mathsf{bl} < 5\%]\langle v\rangle\sigma_1.P \ , \{role = \text{``}master\text{''}\}) \parallel$$
$$(\ \mathsf{stop}^{\star}[\mathsf{role} = \text{``}master\text{''}](x)\sigma_2 \ .Q_1 \ , \{\mathsf{bl} = 4\%\}) \parallel$$
$$(\ \mathsf{stop}^{\star}[\mathsf{role} = \text{``}super\text{''}](x)\sigma_3.Q_2 \ , \{\mathsf{bl} = 2\%\}) \parallel$$
$$(\ \mathsf{stop}^{\star}[\top](x)\sigma_4.Q_3 \ , \{\mathsf{bl} = 2\%\})$$

Examples of interactions...

Broadcast synchronisation:

$$(\text{stop}^\star[\text{bl} < 5\%]\langle v\rangle\sigma_1.P \;, \{\mathit{role} = \text{``}\mathit{master}\text{''}\}) \parallel$$
$$(\text{stop}^\star[\text{role} = \text{``}\mathit{master}\text{''}](x)\sigma_2 \;.Q_1 \;, \{\text{bl} = 4\%\}) \parallel$$
$$(\text{stop}^\star[\text{role} = \text{``}\mathit{super}\text{''}](x)\sigma_3.Q_2 \;, \{\text{bl} = 2\%\}) \parallel$$
$$(\text{stop}^\star[\top](x)\sigma_4.Q_3 \;, \{\text{bl} = 2\%\})$$

$$\Downarrow$$

$$(P, \sigma_1(\{\mathit{role} = \text{``}\mathit{master}\text{''}\})) \parallel$$
$$(Q_1[v/x], \sigma_2(\{\text{bl} = 4\%\})) \parallel$$
$$(\text{stop}^\star[\text{role} = \text{``}\mathit{super}\text{''}](x)\sigma_3.Q_2, \{\text{bl} = 2\%\}) \parallel$$
$$(Q_3[v/x], \sigma_4(\{\text{bl} = 2\%\}))$$

Examples of interactions...

Broadcast synchronisation:

$$(\mathsf{stop}^{\star}[\mathsf{bl} < 5\%]\langle v\rangle\sigma_1.P, \{role = \text{“}master\text{”}\}) \parallel$$
$$(\mathsf{stop}^{\star}[\mathsf{role} = \text{“}master\text{”}](x)\sigma_2.Q_1, \{\mathsf{bl} = 45\%\}) \parallel$$
$$(\mathsf{stop}^{\star}[\mathsf{role} = \text{“}super\text{”}](x)\sigma_3.Q_2, \{\mathsf{bl} = 2\%\}) \parallel$$
$$(\mathsf{stop}^{\star}[\top](x)\sigma_4.Q_3, \{\mathsf{bl} = 25\%\})$$

Examples of interactions. . .

Broadcast synchronisation:

$$(\text{stop}^\star[\text{bl} < 5\%]\langle v\rangle\sigma_1.P, \{role = \text{“}master\text{”}\}) \parallel$$
$$(\text{stop}^\star[\text{role} = \text{“}master\text{”}](x)\sigma_2.Q_1, \{\text{bl} = 45\%\}) \parallel$$
$$(\text{stop}^\star[\text{role} = \text{“}super\text{”}](x)\sigma_3.Q_2, \{\text{bl} = 2\%\}) \parallel$$
$$(\text{stop}^\star[\top](x)\sigma_4.Q_3, \{\text{bl} = 25\%\})$$

Examples of interactions...

Broadcast synchronisation:

$$
\begin{array}{l}
(\mathsf{stop}^{\star}[\mathsf{bl} < 5\%]\langle v\rangle\sigma_1.P, \{role = \text{``}master\text{''}\}) \parallel \\
\quad (\mathsf{stop}^{\star}[\mathsf{role} = \text{``}master\text{''}](x)\sigma_2.Q_1, \{\mathsf{bl} = 45\%\}) \parallel \\
\qquad (\mathsf{stop}^{\star}[\mathsf{role} = \text{``}super\text{''}](x)\sigma_3.Q_2, \{\mathsf{bl} = 2\%\}) \parallel \\
\qquad\qquad (\mathsf{stop}^{\star}[\top](x)\sigma_4.Q_3, \{\mathsf{bl} = 25\%\})
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
(P, \sigma_1(\{role = \text{``}master\text{''}\})) \parallel \\
\quad (\mathsf{stop}^{\star}[\mathsf{role} = \text{``}master\text{''}](x)\sigma_2.Q_1, \{\mathsf{bl} = 45\%\}) \parallel \\
\qquad (\mathsf{stop}^{\star}[\mathsf{role} = \text{``}super\text{''}](x)\sigma_3.Q_2, \{\mathsf{bl} = 2\%\}) \parallel \\
\qquad\qquad (\mathsf{stop}^{\star}[\top](x)\sigma_4.Q_3, \{\mathsf{bl} = 25\%\})
\end{array}
$$

Examples of interactions. . .

Unicast synchronisation:

$$(\mathsf{stop}[\mathsf{bl} < 5\%]\langle\bullet\rangle\sigma_1.P, \{role = \text{“}master\text{”}\}) \parallel$$
$$(\mathsf{stop}[\mathsf{role} = \text{“}master\text{”}](x)\sigma_2.Q_1, \{\mathsf{bl} = 4\%\}) \parallel$$
$$(\mathsf{stop}[\mathsf{role} = \text{“}super\text{”}](x)\sigma_3.Q_2, \{\mathsf{bl} = 2\%\}) \parallel$$
$$(\mathsf{stop}[\top](x)\sigma_4.Q_3, \{\mathsf{bl} = 2\%\})$$

Examples of interactions...

Unicast synchronisation:

$$(\text{stop}[\text{bl} < 5\%]\langle\bullet\rangle\sigma_1.P, \{\mathit{role} = \text{``}\mathit{master}\text{''}\}) \parallel$$
$$(\text{stop}[\text{role} = \text{``}\mathit{master}\text{''}](x)\sigma_2.Q_1, \{\text{bl} = 4\%\}) \parallel$$
$$(\text{stop}[\text{role} = \text{``}\mathit{super}\text{''}](x)\sigma_3.Q_2, \{\text{bl} = 2\%\}) \parallel$$
$$(\text{stop}[\top](x)\sigma_4.Q_3, \{\text{bl} = 2\%\})$$

Examples of interactions. . .

Unicast synchronisation:

$$
\begin{aligned}
&(\mathsf{stop}[\mathsf{bl} < 5\%]\langle\bullet\rangle\sigma_1.P, \{role = \text{“}master\text{”}\}) \parallel \\
&\quad (\mathsf{stop}[\mathsf{role} = \text{“}master\text{”}](x)\sigma_2.Q_1, \{\mathsf{bl} = 4\%\}) \parallel \\
&\qquad (\mathsf{stop}[\mathsf{role} = \text{“}super\text{”}](x)\sigma_3.Q_2, \{\mathsf{bl} = 2\%\}) \parallel \\
&\qquad\qquad (\mathsf{stop}[\top](x)\sigma_4.Q_3, \{\mathsf{bl} = 2\%\})
\end{aligned}
$$

Examples of interactions. . .

Unicast synchronisation:

$$(\text{stop}[\text{bl} < 5\%]\langle\bullet\rangle\sigma_1.P, \{role = \text{“}master\text{”}\}) \parallel$$
$$(\text{stop}[\text{role} = \text{“}master\text{”}](x)\sigma_2.Q_1, \{\text{bl} = 4\%\}) \parallel$$
$$(\text{stop}[\text{role} = \text{“}super\text{”}](x)\sigma_3.Q_2, \{\text{bl} = 2\%\}) \parallel$$
$$(\text{stop}[\top](x)\sigma_4.Q_3, \{\text{bl} = 2\%\})$$

$$\Downarrow$$

$$(P, \sigma_1(\{role = \text{“}master\text{”}\})) \parallel$$
$$(\text{stop}[\text{role} = \text{“}master\text{”}](x)\sigma_2.Q_1, \{\text{bl} = 4\%\}) \parallel$$
$$(\text{stop}[\text{role} = \text{“}super\text{”}](x)\sigma_3.Q_2, \{\text{bl} = 2\%\}) \parallel$$
$$(Q_3, \sigma_4(\{\text{bl} = 2\%\}))$$

Modelling the environment

Interactions between components can be affected by the environment:

- a wall can inhibit wireless interactions;
- two components are too distant to interact;
- ...

Modelling the environment

Interactions between components can be affected by the environment:

- a wall can inhibit wireless interactions;
- two components are too distant to interact;
- ...

The environment...

- is used to model the intrinsic rules that govern the physical context;

Modelling the environment

Interactions between components can be affected by the environment:

- a wall can inhibit wireless interactions;
- two components are too distant to interact;
- ...

The environment...

- is used to model the intrinsic rules that govern the physical context;
- consists of a pair (γ, ρ):

Modelling the environment

Interactions between components can be affected by the environment:

- a wall can inhibit wireless interactions;
- two components are too distant to interact;
- ...

The environment...

- is used to model the intrinsic rules that govern the physical context;
- consists of a pair (γ, ρ):
  - a global store γ, that models the overall state of the system;

Modelling the environment

Interactions between components can be affected by the environment:

- a wall can inhibit wireless interactions;
- two components are too distant to interact;
- ...

The environment...

- is used to model the intrinsic rules that govern the physical context;
- consists of a pair (γ, ρ):
  - a global store γ, that models the overall state of the system;
  - an evolution rule ρ that regulates component interactions (receiving probabilities, action rates,...).

The evolution rule

It is assumed that all actions in CARMA take some time complete and that this duration is governed by an exponential distribution.

The evolution rule

It is assumed that all actions in CARMA take some time complete and that this duration is governed by an exponential distribution.

However the action descriptions do not include any information about the timing (unlike many other stochastic process algebras).

The evolution rule

It is assumed that all actions in CARMA take some time complete and that this duration is governed by an exponential distribution.

However the action descriptions do not include any information about the timing (unlike many other stochastic process algebras).

We also do not assume perfect communication, i.e. there may be a probability that an interaction will fail to complete even between components with appropriately match attributes.

The evolution rule

It is assumed that all actions in CARMA take some time complete and that this duration is governed by an exponential distribution.

However the action descriptions do not include any information about the timing (unlike many other stochastic process algebras).

We also do not assume perfect communication, i.e. there may be a probability that an interaction will fail to complete even between components with appropriately match attributes.

The environment manages these aspects of system behaviour, and others in the evolution rule.

The evolution rule ρ

ρ is a function, dependent on current time, the global store and the current state of the collective, returns a tuple of functions $\varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle$ known as the evaluation context

- $\mu_p(\gamma_s, \gamma_r, \alpha)$: the probability that a component with store γ_r can receive a broadcast message α from a component with store γ_s;
- $\mu_w(\gamma_s, \gamma_r, \alpha)$: the weight to be used to compute the probability that a component with store γ_r can receive a unicast message α from a component with store γ_s;
- $\mu_r(\gamma_s, \alpha)$ computes the execution rate of action α executed at a component with store γ_s;
- $\mu_u(\gamma_s, \alpha)$ determines the updates on the environment (global store and collective) induced by the execution of action α at a component with store γ_s.

Outline

Operational Semantics of PA

Behaviour of process algebras is classically represented via transition relations, e.g. $\alpha.P \xrightarrow{\alpha} P$.

Operational Semantics of PA

Behaviour of process algebras is classically represented via transition relations, e.g. $\alpha.P \xrightarrow{\alpha} P$.

These relations, defined following a Plotkin-style, are used to infer possible computations of a process.

$$\frac{\mathit{premise}}{\mathit{conclusion}}\ \textbf{Rule}$$

Operational Semantics of PA

Behaviour of process algebras is classically represented via transition relations, e.g. $\alpha.P \xrightarrow{\alpha} P$.

These relations, defined following a Plotkin-style, are used to infer possible computations of a process.

$$\frac{premise}{conclusion}\ \textbf{Rule}$$

Note that, due to nondeterminism, starting from the same process, different evolutions can be inferred.

Operational Semantics of PA

Behaviour of process algebras is classically represented via transition relations, e.g. $\alpha.P \xrightarrow{\alpha} P$.

These relations, defined following a Plotkin-style, are used to infer possible computations of a process.

$$\frac{premise}{conclusion}\ \textbf{Rule}$$

Note that, due to nondeterminism, starting from the same process, different evolutions can be inferred.

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}\ \textbf{Choice1} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}\ \textbf{Choice2}$$

Operational Semantics of SPA

However, in Stochastic Process Algebras, like CARMA, there is not any form of nondeterminism...

Operational Semantics of SPA

However, in Stochastic Process Algebras, like CARMA, there is not any form of nondeterminism...

...while the selection of possible next state is governed by a probability distribution.

Operational Semantics of SPA

However, in Stochastic Process Algebras, like CARMA, there is not any form of nondeterminism...

...while the selection of possible next state is governed by a probability distribution.

Standard compositional approaches are cumbersome and may fail when rich SPA are considered (e.g., when the multiplicity of transitions is important).

Operational Semantics of SPA

The operational semantics of CARMA is defined in the FuTS style.

Operational Semantics of SPA

The operational semantics of CARMA is defined in the FuTS style.

In FuTS the behaviour of a term is described using a function that, given a term and a transition label, yields a function associating each component, collective, or system with a non-negative number.

Operational Semantics of SPA

The operational semantics of CARMA is defined in the FuTS style.

In FuTS the behaviour of a term is described using a function that, given a term and a transition label, yields a function associating each component, collective, or system with a non-negative number.

The meaning of this value depends on the context:

- the rate of the exponential distribution characterising the time needed for the execution of an action;

Operational Semantics of SPA

The operational semantics of CARMA is defined in the FuTS style.

In FuTS the behaviour of a term is described using a function that, given a term and a transition label, yields a function associating each component, collective, or system with a non-negative number.

The meaning of this value depends on the context:

- the rate of the exponential distribution characterising the time needed for the execution of an action;
- the probability of receiving a given broadcast message;

Operational Semantics of SPA

The operational semantics of CARMA is defined in the FuTS style.

In FuTS the behaviour of a term is described using a function that, given a term and a transition label, yields a function associating each component, collective, or system with a non-negative number.

The meaning of this value depends on the context:

- the rate of the exponential distribution characterising the time needed for the execution of an action;
- the probability of receiving a given broadcast message;
- the weight used to compute the probability that a given component is selected for the synchronisation.

CARMA Operational Semantics

The operational semantics of CARMA specifications is defined in terms of three functions that compute the possible next states of a component, a collective and a system:

CARMA Operational Semantics

The operational semantics of CARMA specifications is defined in terms of three functions that compute the possible next states of a component, a collective and a system:

1. the function $\mathbb{C}$ that describes the behaviour of a single component;

CARMA Operational Semantics

The operational semantics of CARMA specifications is defined in terms of three functions that compute the possible next states of a component, a collective and a system:

1. the function $\mathbb{C}$ that describes the behaviour of a single component;
2. the function $\mathbb{N}_{\varepsilon}$ builds on $\mathbb{C}$ to describe the behaviour of collectives;

Carma Operational Semantics

The operational semantics of Carma specifications is defined in terms of three functions that compute the possible next states of a component, a collective and a system:

1. the function $\mathbb{C}$ that describes the behaviour of a single component;
2. the function $\mathbb{N}_{\varepsilon}$ builds on $\mathbb{C}$ to describe the behaviour of collectives;
3. the function $\mathbb{S}_{t}$ that shows how Carma systems evolve.

CARMA Operational Semantics

The operational semantics of CARMA specifications is defined in terms of three functions that compute the possible next states of a component, a collective and a system:

1. the function $\mathbb{C}$ that describes the behaviour of a single component;
2. the function $\mathbb{N}_{\varepsilon}$ builds on $\mathbb{C}$ to describe the behaviour of collectives;
3. the function $\mathbb{S}_t$ that shows how CARMA systems evolve.

In all cases the value zero is associated with unreachable terms.

Component Semantics

The behaviour of a single component is defined by a function

$$\mathbb{C} : \textsc{Comp} \times \textsc{Lab} \rightarrow [\textsc{Comp} \rightarrow \mathbb{R}_{\geq 0}]$$

Component Semantics

The behaviour of a single component is defined by a function

$$\mathbb{C} : \textsc{Comp} \times \textsc{Lab} \to [\textsc{Comp} \to \mathbb{R}_{\geq 0}]$$

where $\textsc{Lab}$ denotes the set of transition labels ℓ:

$$\begin{array}{lcll} \ell & ::= & \alpha^{\star}[\pi_s]\langle \overrightarrow{v} \rangle, \gamma & \text{Broadcast Output} \\ & \mid & \alpha^{\star}[\pi_s](\overrightarrow{v}), \gamma & \text{Broadcast Input} \\ & \mid & \alpha[\pi_s]\langle \overrightarrow{v} \rangle, \gamma & \text{Unicast Output} \\ & \mid & \alpha[\pi_s](\overrightarrow{v}), \gamma & \text{Unicast Input} \end{array}$$

Component Semantics

The behaviour of a single component is defined by a function

$$\mathbb{C} : \textsc{Comp} \times \textsc{Lab} \rightarrow [\textsc{Comp} \rightarrow \mathbb{R}_{\geq 0}]$$

where $\textsc{Lab}$ denotes the set of transition labels ℓ:

$$\begin{array}{lcll} \ell & ::= & \alpha^{\star}[\pi_s]\langle \overrightarrow{v} \rangle, \gamma & \text{Broadcast Output} \\ & \mid & \alpha^{\star}[\pi_s](\overrightarrow{v}), \gamma & \text{Broadcast Input} \\ & \mid & \alpha[\pi_s]\langle \overrightarrow{v} \rangle, \gamma & \text{Unicast Output} \\ & \mid & \alpha[\pi_s](\overrightarrow{v}), \gamma & \text{Unicast Input} \end{array}$$

If $\mathbb{C}[C, \ell] = \mathscr{C}$ and $\mathscr{C}(C') = p$ then C evolves to C' with a weight p when ℓ is executed.

Component Semantics

Some rules. . .

$$\frac{}{\mathbb{C}[(\mathbf{nil},\gamma),\ell] = \emptyset}\ \textbf{Nil} \qquad \frac{}{\mathbb{C}[\mathbf{0},\ell] = \emptyset}\ \textbf{Zero}$$

$$\frac{[\![\pi_s]\!]_\gamma = \pi_s' \quad [\![\overrightarrow{e}]\!]_\gamma = \overrightarrow{v} \quad \mathbf{p} = \sigma(\gamma)}{\mathbb{C}[(\alpha^\star[\pi_s]\langle\overrightarrow{e}\rangle\sigma.P,\gamma),\alpha^\star[\pi_s']\langle\overrightarrow{v}\rangle,\gamma] = (P,\mathbf{p})}\ \textbf{B-Out}$$

$$\frac{\gamma_r \models \pi_s \quad \gamma_s \models \pi_r[\overrightarrow{v}/\overrightarrow{x}] \quad \mathbf{p} = \sigma[\overrightarrow{v}/\overrightarrow{x}](\gamma_2)}{\mathbb{C}[(\alpha^\star[\pi_r](\overrightarrow{x})\sigma.P,\gamma_r),\alpha^\star[\pi_s](\overrightarrow{v}),\gamma_s] = (P[\overrightarrow{v}/\overrightarrow{x}],\mathbf{p})}\ \textbf{B-In}$$

$$\frac{\mathbb{C}[(P,\gamma),\ell] = \mathscr{C}_1 \quad \mathbb{C}[(Q,\gamma),\ell] = \mathscr{C}_2}{\mathbb{C}[(P+Q,\gamma),\ell] = \mathscr{C}_1 \oplus \mathscr{C}_2}\ \textbf{Plus}$$

$$\frac{\mathbb{C}[(P,\gamma),\ell] = \mathscr{C}_1 \quad \mathbb{C}[(Q,\gamma),\ell] = \mathscr{C}_2}{\mathbb{C}[(P|Q,\gamma),\ell] = \mathscr{C}_1|Q \oplus P|\mathscr{C}_2}\ \textbf{Par}$$

Semantics of Collectives

The operational semantics of a collective is defined via the function

$$\mathbb{N}_{\varepsilon} : \textsc{Col} \times \textsc{Lab}_{I} \rightarrow [\textsc{Col} \rightarrow \mathbb{R}_{\geq 0}]$$

defining how a collective reacts when a broadcast/unicast message is received.

Semantics of Collectives

The operational semantics of a collective is defined via the function

$$\mathbb{N}_{\varepsilon} : \textsc{Col} \times \textsc{Lab}_I \rightarrow [\textsc{Col} \rightarrow \mathbb{R}_{\geq 0}]$$

defining how a collective reacts when a broadcast/unicast message is received.

$\textsc{Lab}_I$ denotes the subset of $\textsc{Lab}$ with only input labels:

$$\begin{array}{rcll} \ell & ::= & \alpha^{\star}[\pi_s](\overrightarrow{v}), \gamma & \text{Broadcast Input} \\ & \mid & \alpha[\pi_s](\overrightarrow{v}), \gamma & \text{Unicast Input} \end{array}$$

Semantics of Collectives

Some rules...

$$\frac{}{\mathbb{N}_{\varepsilon}[\mathbf{0},\ell] = \emptyset} \textbf{ Zero}$$

$$\frac{\mathbb{C}[(P,\gamma),\alpha^{\star}[\pi_s](\overrightarrow{v}),\gamma] = \mathscr{N} \quad \mathscr{N} \neq \emptyset \quad \varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle}{\mathbb{N}_{\varepsilon}[(P,\gamma),\alpha^{\star}[\pi_s](\overrightarrow{v}),\gamma] = \frac{\mu_p(\gamma,\alpha^{\star})}{\oplus \mathscr{N}} \cdot \mathscr{N} + [(P,\gamma) \mapsto (1-\mu_p(\gamma,\alpha^{\star})]} \textbf{ Comp-B-In}$$

$$\frac{\mathbb{N}_{\varepsilon}[N_1,\alpha^{\star}[\pi_s](\overrightarrow{v}),\gamma] = \mathscr{N}_1 \quad \mathbb{N}_{\varepsilon}[N_2,\alpha^{\star}[\pi_s](\overrightarrow{v}),\gamma] = \mathscr{N}_2}{\mathbb{N}_{\varepsilon}[N_1 \parallel N_2,\alpha^{\star}[\pi_s](\overrightarrow{v}),\gamma] = \mathscr{N}_1 \parallel \mathscr{N}_2} \textbf{ B-In-Sync}$$

Semantics of Systems

The operational semantics of systems is defined via the function

$$\mathbb{S}_t : \textsc{Sys} \times \textsc{Lab}_S \to [\textsc{Sys} \to \mathbb{R}_{\geq 0}]$$

Semantics of Systems

The operational semantics of systems is defined via the function

$$\mathbb{S}_t : \textsc{Sys} \times \textsc{Lab}_S \rightarrow [\textsc{Sys} \rightarrow \mathbb{R}_{\geq 0}]$$

This function only considers synchronisation labels $\textsc{Lab}_S$:

$$\begin{array}{lcll} \ell & ::= & \alpha^{\star}[\pi_s]\langle\overrightarrow{v}\rangle, \gamma & \text{Broadcast Output} \\ & \mid & \tau[\alpha[\pi_s]\langle\overrightarrow{v}\rangle, \gamma] & \text{Unicast Synchronization} \end{array}$$

Semantics of Systems

Some rules. . .

$$\frac{\rho(t,\gamma_g,N) = \varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle \quad \mu_u(\gamma_g, \alpha^\star) = (\sigma, N') \\ \sum_{C \in N} N(C) \cdot \mathrm{bSync}(C, N - C, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma) = \mathscr{N}}{\mathbb{S}_t[N \textbf{ in } (\gamma_g, \rho), \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma] = \mathscr{N} \parallel N' \textbf{ in } (\sigma(\gamma_g), \rho)} \ \textbf{Sys-B}$$

where

$$\frac{\varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle \quad \mathbb{C}[C, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma] = \mathscr{C} \quad \mathbb{N}_\varepsilon[N, \alpha^\star[\pi_s](\overrightarrow{v}), \gamma] = \mathscr{N}}{\mathrm{bSync}_\varepsilon(C, N, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma) \quad = \quad \mu_r(\gamma, \alpha^\star[\pi_s]\langle \overrightarrow{v} \rangle, \gamma) \cdot \mathscr{C} \parallel \mathscr{N}}$$

Quantitative Analysis

The semantics of CARMA gives rise to a Continuous Time Markov Chain (CTMC).

This can be analysed by

- by numerical analysis of the CTMC for small systems;
- by stochastic simulation of the CTMC;
- by fluid approximation of the CTMC under certain restrictions (particularly on the environment).

(10 minutes)

Outline

Recap. . .

Recap. . .

CARMA is equipped with linguistic constructs specifically developed for modelling and programming systems that can operate in open-ended and unpredictable environments.

Recap...

CARMA is equipped with linguistic constructs specifically developed for modelling and programming systems that can operate in open-ended and unpredictable environments.

However, CARMA is a SPA:

- concise syntax;
- parametric with respect to the used expressions and data types.

Recap...

CARMA is equipped with linguistic constructs specifically developed for modelling and programming systems that can operate in open-ended and unpredictable environments.

However, CARMA is a SPA:

- concise syntax;
- parametric with respect to the used expressions and data types.

To facilitate the use of CARMA in the specification/analysis process of CAS we developed:

- a specification language;
- an Eclipse plug-in as a container for CARMA tools.

A running example...

Bike Sharing System...

We want to use CARMA to model a bike sharing system where:

- bikes are made available in a number of stations that are placed in various areas of a city;
- Users that plan to use a bike for a short trip
  - can pick up a bike at a suitable origin station
  - return it to any other station close to their planned destination.
- we assume that the city is partitioned in homogeneous zones...
  - and that all the stations in the same zone can be equivalently used by any user in that zone.

Outline

CARMA Specification Language. . .

Each CARMA specification, also named CARMA model, provides definitions for:

- structured data types;
- constants and functions;
- prototypes of components;
- collective of components;
- systems composed by collective and environment;
- measures, that identify the relevant data to retrieve during simulation runs.

CaSL: Data types

CASL: Data types

Basic data types

- `bool`, for booleans;
- `int`, for integers;
- `real`, for real values.

CaSL: Data types

Basic data types

- `bool`, for booleans;
- `int`, for integers;
- `real`, for real values.

Collections

- Sets: `{` $exp_1, \ldots, exp_n$ `}`
- Arrays: `[:` $exp_1, \ldots, exp_n$ `:]`

CaSL: Data types

Basic data types

- `bool`, for booleans;
- `int`, for integers;
- `real`, for real values.

Collections

- Sets: `{` exp_1`,...,`exp_n `}`
- Arrays: `[:` exp_1`,...,`exp_n `:]`

Custom data types

- Enumerations: `enum` *name* `=` $elem_1$`,...,`$elem_n$`;`
- Records: `record` *name* `= [` $type_1$ $field_1$`,...,` $type_n$ $field_n$`];`

CaSL: Expressions

In CaSL syntax of *expressions* includes:

- standard arithmetic and logical operations $(+, -, \ldots)$
- common functions (log,sin,...)
- conditional expression `(`*exp*1`?`*exp*2`:`*exp*3`)`
- special value `now`, indicating the current time
- ...

A limited set of expressions has to be used when specific analysis tools (fluid semantics) are used.

CaSL: Operators in expressions

Arithmetic Operators: `+`, `-`, `*`, `+`

Unary Operators: `+`, `-`, `!`;

Equality and Relational Operators: `==`, `>`, `>=`, `!=`, `<`, `<=`

Conditional Operators: `&&`, `||`

Compact if-then-else: `( ? : )`

Math functions: `abs`, `sin`, `cos`,...

Set Operations: `&&`, `||`

List/Array Operations: `+`, `x[e]`

Collection Operations: `map`, `find`, `filter`, `exists`, `forall`,...

CaSL: Constants and Functions

A Carma specification can also contain constants and functions declarations having the following syntax:

```
const name = expression;

fun type name( type_1 arg_1 ,..., type_k arg_k ) {
   ...
}
```

CASL: Constants and Functions

Function statements...

Variable declaration, assignment and return:

$type\ \ var\ = exp\,;$

$var\ = exp\,;$

return $exp\,;$

If-then-else:

if $(exp)\ \{\ \ldots\ \}$ **else** $\{\ \ldots\ \}$

Iterators:

for $(var{=}exp_1\,;exp_2\,;exp_3)\{\ \ldots\ \}$

for var **in** $exp\ \{\ \ldots\ \}$

CASL: Constants and Functions

Example. . .

```
const int MAX_VALUE = 100;

fun int maxValue( set<int> s ) {
  if (size(s)==0) {
    return 0;
  }
  int result = MAX_VALUE;
  for v in s {
    result = max( result , v );
  }
  return result;
}
```

CaSL: Component Prototypes

CASL: Component Prototypes

A component prototype defines the general structure of a component:

$$
\begin{array}{l}
\textbf{component}\ \ \mathit{name}(\ \ \mathit{type}_1\ \ \mathit{arg}_1, \ldots,\ \ \mathit{type}_n\ \ \mathit{arg}_n)\ \ \{\\
\quad \textbf{store}\ \ \{\ \cdots\\
\quad\quad \textbf{attrib}\ \ \mathit{aname}_i\ := \ \mathit{expression}_i\,;\ \ \cdots\\
\quad \}\\
\quad \textbf{behaviour}\ \ \{\ \cdots\\
\quad\quad \mathit{proc}_i\ =\ \mathit{pdef}_i\,;\ \ \cdots\\
\quad \}\\
\quad \textbf{init}\ \ \{\ \ P_1 \mid \cdots \mid P_w\ \ \}\\
\}
\end{array}
$$

CASL: Component Prototypes

The BSS scenario...

Two kinds of components, one for each of the two groups of agents involved in our BSS, can be considered:

- parking stations;
- users.

PS attributes:

- **zone**: indicates where the station is located;
- **capacity**: the number of slots installed in the station;
- **available**: the number of available bikes.

User attributes:

- **zone**: current user location;
- **dest**: user destination.

CaSL: Component Prototypes

Example: users and stations

```
component Station( int zone , int capacity , int
    available ) {
  store {
    zone = zone;
    available = available;
    capacity = capacity;
  }
  ...
}
component User( int zone , int dest ) {
  store {
    zone = zone;
    dest = dest;
  }
  ...
}
```

Behaviour...

The block `behaviour` is used to define the component behaviour...

... it consists of a sequence of process definitions

```
behaviour {
   proc_1 = pdef_1 ;
   ...
   proc_n = pdef_n ;
}
```

... that associate each process name with alternative actions.

Output actions:

$$
\begin{array}{l}
[\ guard\ ]\ \mathtt{act}\ [\ pred\ ] < exp_1\ ,\ \ldots\ ,\ exp_n > \{ \\
\quad attr_1 = exp_1\,; \\
\quad \ldots \\
\quad attr_n = exp_n\,; \\
\}
\end{array}
$$

Input actions:

$$
\begin{array}{l}
[\ guard\ ]\ \mathtt{act}\ [\ pred\ ]\ (\ x_1\ ,\ \ldots\ ,\ x_n\ )\ \{ \\
\quad attr_1 = exp_1\,; \\
\quad \ldots \\
\quad attr_n = exp_n\,; \\
\}
\end{array}
$$

CASL: Component Prototypes

Example: Station behaviour

Two processes are defined at the `Station` component that model the procedures to get and returning a bike:

```
G = [my.available > 0] get< >
      { my.available := my.available-1 }.G;

R = [my.available < my.capacity] ret< >
      { my.available := my.available+1  }.R;
```

Procedures get and returning are modelled via unicast output over `get` and `ret`:

- `get` is enabled when there are bikes available (`my.available > 0`);
- `ret` is enabled when there are available slots (`my.available < my.capacity`).

CaSL: Component Prototypes

Example: User behaviour

Each user can be in three different states...

CaSL: Component Prototypes

Example: User behaviour

Each user can be in three different states...

... P, denoting a pedestrian:

```
P = get[ my.zone == zone ]().B;
```

CASL: Component Prototypes

Example: User behaviour

Each user can be in three different states...

... P, denoting a pedestrian:

```
P = get[ my.zone == zone ]().B;
```

... a pedestrian executes unicast input get to collect a bike from a station located in his/her current zone (my.zone == zone) and then becomes a biker:

```
B = move*[ false ]<>{ my.zone := my.dest }.W;
```

CaSL: Component Prototypes

Example: User behaviour

Each user can be in three different states...

... P, denoting a pedestrian:

```
P = get[ my.zone == zone ]().B;
```

... a pedestrian executes unicast input get to collect a bike from a station located in his/her current zone (my.zone == zone) and then becomes a biker:

```
B = move*[ false ]<>{ my.zone := my.dest }.W;
```

... in that state a user moves to the final destination and then waits for a slot:

```
W = ret[ my.zone == zone ]().kill;
```

CaSL: Component Prototypes

Example: User behaviour

Each user can be in three different states...

... P, denoting a pedestrian:

```
P = get[ my.zone == zone ]().B;
```

... a pedestrian executes unicast input get to collect a bike from a station located in his/her current zone (my.zone == zone) and then becomes a biker:

```
B = move*[ false ]<>{ my.zone := my.dest }.W;
```

... in that state a user moves to the final destination and then waits for a slot: SPONTANEOUS ACTION!

```
W = ret[ my.zone == zone ]().kill;
```

CASL: Component Prototypes

Example: User behaviour

Each user can be in three different states...

... P, denoting a pedestrian:

```
P = get[ my.zone == zone ]().B;
```

... a pedestrian executes unicast input get to collect a bike from a station located in his/her current zone (my.zone == zone) and then becomes a biker:

```
B = move*[ false ]<>{ my.zone := my.dest }.W;
```

... in that state a user moves to the final destination and then waits for a slot: SPONTANEOUS ACTION!

```
W = ret[ my.zone == zone ]().kill;
```

... when a slot in the same zone is found, the user disappear.

Example: Station

```
component Station( int zone , int capacity , int
    available ) {
  store {
      zone = zone;
      available = available;
      capacity = capacity;
    }
    behaviour {
    G = [my.available > 0] get< > {
        my.available := my.available-1
    }.G;
    R = [my.available < my.capacity] ret< > {
        my.available := my.available+1
    }.R;
  }
  init { G|R }
}
```

Example: User

```
component User( int zone , int dest ) {
  store {
    zone = zone;
    dest = dest;
  }
  behaviour {
    P = get[ my.zone == zone ]().B;
    B = move*[ false ]<>{ my.zone := my.dest; }.W;
    W = ret[ my.zone == zone ]().kill;
  }
  init {
    P
  }
}
```

Example: Arrival

To model the arrival of new users, another component is considered in our model:

```
component Arrival( int zone ) {
  store {
    zone = zone;
  }
  behaviour {
    A = arrival*[false]<>.A;
  }
  init {
    A
  }
}
```

Example: Arrival

To model the arrival of new users, another component is considered in our model:

```
component Arrival( int zone ) {
  store {
    zone = zone;
  }
  behaviour {
    A = arrival*[false]<>.A;
  }
  init {
    A
  }
}
```

Above `zone` indicates the location where users arrive.

CaSL: Collective

Block `collective` can be used to define groups of components:

$$\textbf{collective}\ \mathit{name}\ (\ \mathit{type}_1\ \mathit{var}_1\ ,\ \ldots\ ,\ \mathit{type}_n\ \mathit{var}_n\ )\ \{$$
$$\ldots$$
$$\}$$

CASL: Collective

Block `collective` can be used to define groups of components:

```
collective name ( type1 var1 , ... , typen varn ) {
  ...
}
```

BSS Collective:

```
collective bssCollective( int zones , int n ) {
  for ( i ; i<zones ; 1 ) {
    new Station( i , C, A )< n >;
  }
  new Arrival( i );
}
```

CaSL: System Definition

A system definition consists of two blocks, namely `collective` and `environment`:

```
system name {
  collective collective
  environment { ···
  }
}
```

CASL: System Definition

A system definition consists of two blocks, namely `collective` and `environment`:

```
system name {
  collective collective
  environment { ···
  }
}
```

BSS System:

```
system Scenario {
  collective bssCollective( 10 , 10 , 10 )
  environment { ···
  }
}
```

CASL: Environment

The block `environment` is used to define system environment:

```
environment {
  store { ··· }
  prob { ··· }
  weight { ··· }
  rate { ··· }
  update { ··· }
}
```

Block `store` defines the global store while blocks `weight`, `prob`, `rate` and `update` define the evolution rule ρ.

The evolution rule ρ

Déjà vu...

ρ is a function, dependent on current time, the global store and the current state of the collective, returns a tuple of functions $\varepsilon = \langle \mu_p, \mu_w, \mu_r, \mu_u \rangle$ known as the evaluation context.

- $\mu_p(\gamma_s, \gamma_r, \alpha)$: the probability that a component with store γ_r can receive a broadcast message α from a component with store γ_s;
- $\mu_w(\gamma_s, \gamma_r, \alpha)$: the weight to be used to compute the probability that a component with store γ_r can receive a unicast message α from a component with store γ_s;
- $\mu_r(\gamma_s, \alpha)$ computes the execution rate of action α executed at a component with store γ_s;
- $\mu_u(\gamma_s, \alpha)$ determines the updates on the environment (global store and collective) induced by the execution of action α at a component with store γ_s.

The block `store` is used to declare global attributes.

CaSL: Environment
Example: BSS Store

The block `store` is used to declare global attributes.

In our scenario we use a global attribute to count the number of active users in the system:

```
store {
  attrib users := 0;
}
```

CaSL: Environment
The store)

The block `weight` associates each unicast input with a (positive) real value.

CASL: Environment
The store)

The block `weight` associates each unicast input with a (positive) real value.

```
weight {
  get {
    return ReceivingProb(#{User[P]|my.zone==sender.zone})
  }
  ret {
    return ReceivingProb(#{User[W]|my.zone==sender.zone})
        ;
  }
}
```

CaSL: Environment
The store)

To define the probability that a component receives a broadcast message, block `prob` is used.

CASL: Environment

The store)

To define the probability that a component receives a broadcast message, block `prob` is used.

```
prob {
  default {
    return 1;
  }
}
```

Action rates is computed in the `rate` block.

Example: BSS Environment (2/3)

Action rates is computed in the `rate` block.

```
rate {
  get { return get_rate; }
  ret { return ret_rate; }
  move* { return move_rate; }
  arrival* {
if (global.users<TOTAL_USERS) {
  return arrival_rate;
} else {
  return 0.0;
}
  }
}
```

CASL: Environment

Example: BSS Environment (3/3)

Block `update` is used to define how environment reacts to collective evolution

CaSL: Environment
Example: BSS Environment (3/3)

Block `update` is used to define how environment reacts to collective evolution

```
update {
  arrival* {
    users := global.users+1;
    new User( sender.zone , U[0:ZONES-1] );
  }
  ret {
    users := global.users-1;
  }
}
```

CASL: Measures

To extract observations from a model, a CASL specification also contains a set of measures:

$$\textbf{measure} \ \ m\_name(\ type_1\ var_1, \ \ldots, \ type_1\ var_n) = expr;$$

```
measure AverageBikes( int z ) =
    avg{ my.available | my.zone == z };
measure MinBikes( int z ) =
    min{ my.available | my.zone == z };
measure MaxBikes( int z ) =
    max{ my.available | my.zone == z };
```

(5 minutes)

Outline

The CARMA Eclipse Plug-in

http://quanticol.sourceforge.net/

CARMA Eclipse Plug-In provide tools for specification and quantitative analysis of CARMA models.

The Carma Eclipse Plug-in

http://quanticol.sourceforge.net/

Carma Eclipse Plug-In provide tools for specification and quantitative analysis of Carma models.

Frontend

The CARMA Eclipse Plug-in

http://quanticol.sourceforge.net/

CARMA Eclipse Plug-In provide tools for specification and quantitative analysis of CARMA models.

Frontend

CARMA Editor
(Xtext based)

The Carma Eclipse Plug-in

http://quanticol.sourceforge.net/

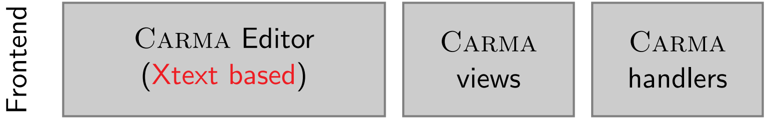
Carma Eclipse Plug-In provide tools for specification and quantitative analysis of Carma models.

Frontend	Carma Editor (Xtext based)	Carma views

The Carma Eclipse Plug-in

http://quanticol.sourceforge.net/

Carma Eclipse Plug-In provide tools for specification and quantitative analysis of Carma models.



The CARMA Eclipse Plug-in

http://quanticol.sourceforge.net/

CARMA Eclipse Plug-In provide tools for specification and quantitative analysis of CARMA models.

Frontend

CARMA Editor (Xtext based)

CARMA views

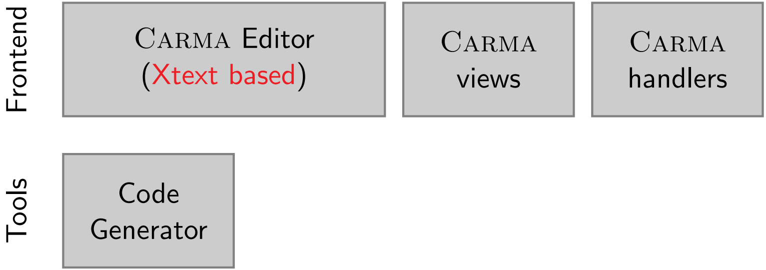
CARMA handlers

Tools

The Carma Eclipse Plug-in

http://quanticol.sourceforge.net/

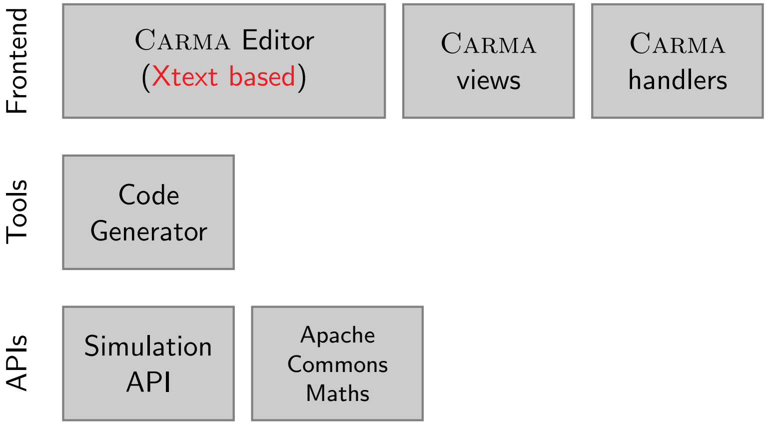
Carma Eclipse Plug-In provide tools for specification and quantitative analysis of Carma models.



The CARMA Eclipse Plug-in

http://quanticol.sourceforge.net/

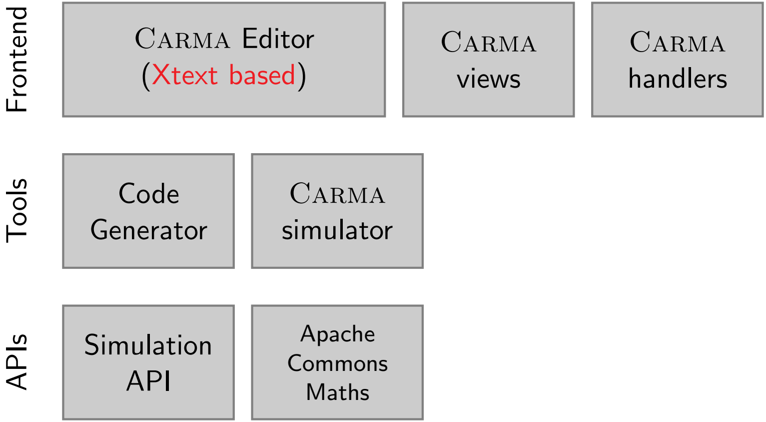
CARMA Eclipse Plug-In provide tools for specification and quantitative analysis of CARMA models.



The Carma Eclipse Plug-in

http://quanticol.sourceforge.net/

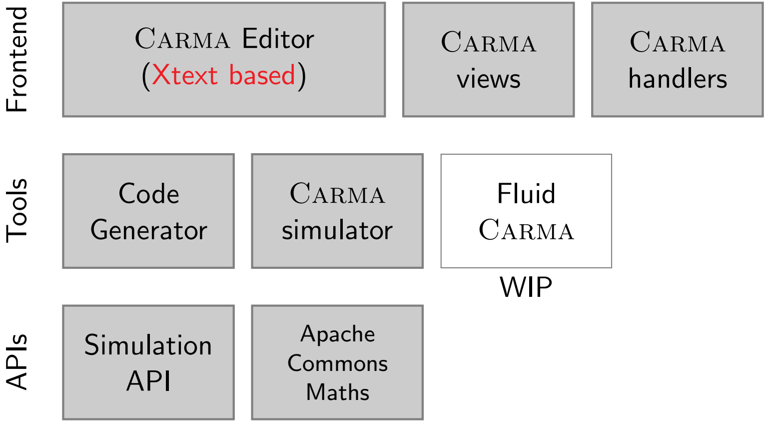
Carma Eclipse Plug-In provide tools for specification and quantitative analysis of Carma models.



The Carma Eclipse Plug-in

http://quanticol.sourceforge.net/

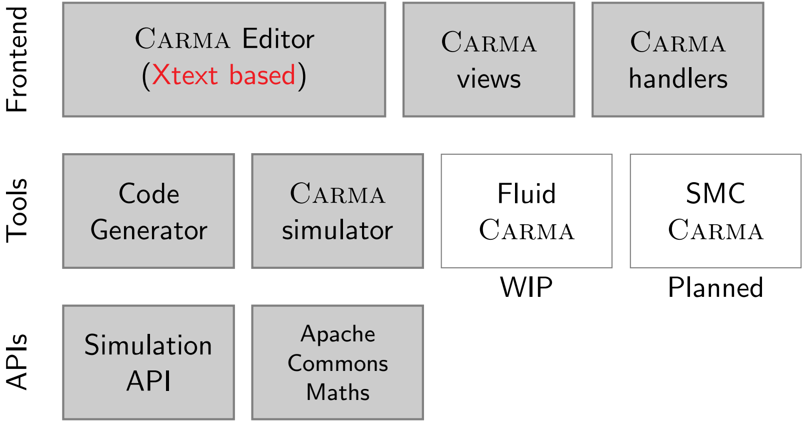
Carma Eclipse Plug-In provide tools for specification and quantitative analysis of Carma models.



The CARMA Eclipse Plug-in

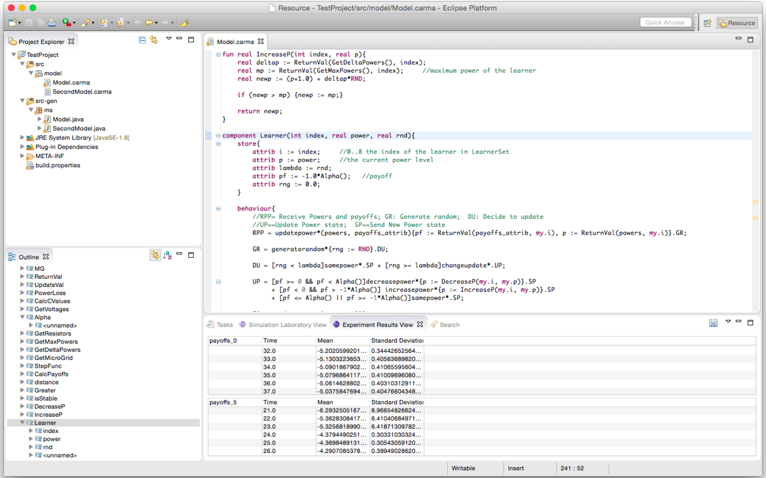
http://quanticol.sourceforge.net/

CARMA Eclipse Plug-In provide tools for specification and quantitative analysis of CARMA models.

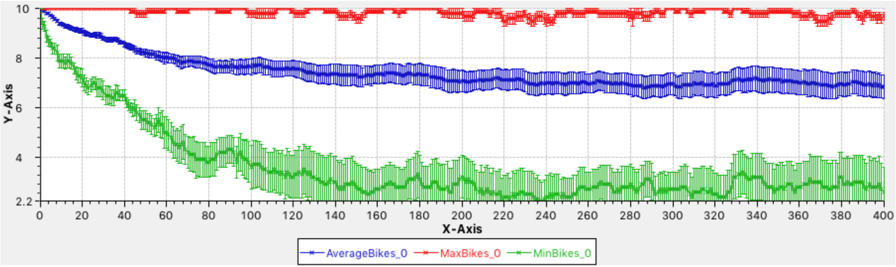


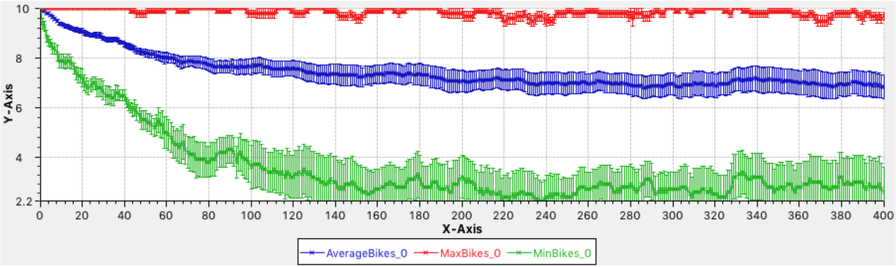
The CARMA Eclipse Plug-in

LIVE DEMO



CaSL: BSS Analysis





In this scenario the use of stations is not well balanced!

CaSL: an alternative model for BSS

To overcome this problem we can consider a model where stations located at the same zone do not compete but cooperate.

CaSL: an alternative model for BSS

To overcome this problem we can consider a model where stations located at the same zone do not compete but cooperate.

We consider `CollaborativeStation`s that, when located at the same zone, interact to avoid unbalanced use of resources.

CaSL: an alternative model for BSS

To overcome this problem we can consider a model where stations located at the same zone do not compete but cooperate.

We consider `CollaborativeStations` that, when located at the same zone, interact to avoid unbalanced use of resources.

Each station can use broadcast to advertise other agents about the use of resources!

CaSL: an alternative model for BSS

In a `CollaborativeStation` actions `get` and `ret` can be enabled or not. . .

CaSL: an alternative model for BSS

In a `CollaborativeStation` actions `get` and `ret` can be enabled or not...

```
component CollaborativeStation( int zone , int capacity ,
    int available ) {
  store {
    zone = zone;
    available = available;
    capacity = capacity;
    get_enabled = true;
    ret_enabled = true;
  }
```

CaSL: BSS Alternative model

...local attributes `get_enabled` and `ret_enabled` are used to control behaviour of processes `G` and `R`...

CaSL: BSS Alternative model

...local attributes `get_enabled` and `ret_enabled` are used to control behaviour of processes `G` and `R`...

```
behaviour {
  G = [my.available>0 && my.get_enabled]
        get<>{ my.available := my.available-1 }.G;
  R = [my.available<my.capacity && my.ret_enabled]
        ret<>{ my.available := my.available+1 }.R;
```

CASL: BSS Alternative model

…these attributes are changed when info about available resources in the same zone are received.

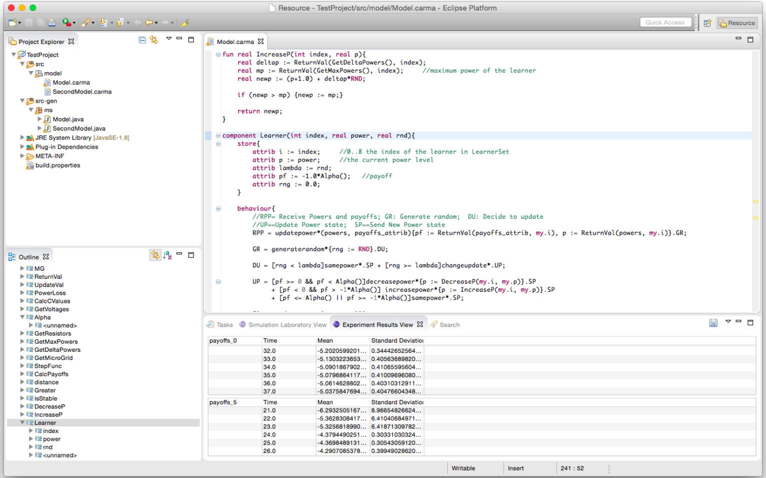
CaSL: BSS Alternative model

. . . these attributes are changed when info about available resources in the same zone are received.

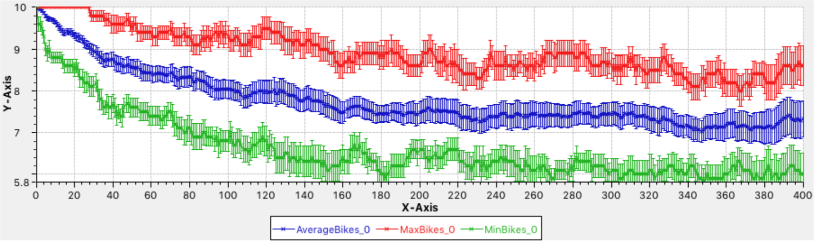
```
C =
  [my.get_enabled || my.ret_enabled]
                         spread*< my.available >.C
  +
  spread*[ my.zone == zone ]( x )
       { my.get_enabled := my.available >= x ,
         my.ret_enabled := my.available <= x }.C;
}

init {
  G|R|C
}

}
```

The CARMA Eclipse Plug-in

LIVE DEMO



CaSL: modified BSS model Analysis



Outline

BSS Scenario: User arrival rate

In the BSS scenarios considered users arrive at a constant rate. . .

BSS Scenario: User arrival rate

In the BSS scenarios considered users arrive at a constant rate. . .

. . . this is not complete realistic for a real system where, for instance, bikes could be used mainly in the morning. . .

BSS Scenario: User arrival rate

In the BSS scenarios considered users arrive at a constant rate. . .

. . . this is not complete realistic for a real system where, for instance, bikes could be used mainly in the morning. . .

We can change our environment definition so that the user arrival rate is higher at the beginning and then decreases.

Time dependent rates...

Expression `now` can be used to model time-dependent environment:

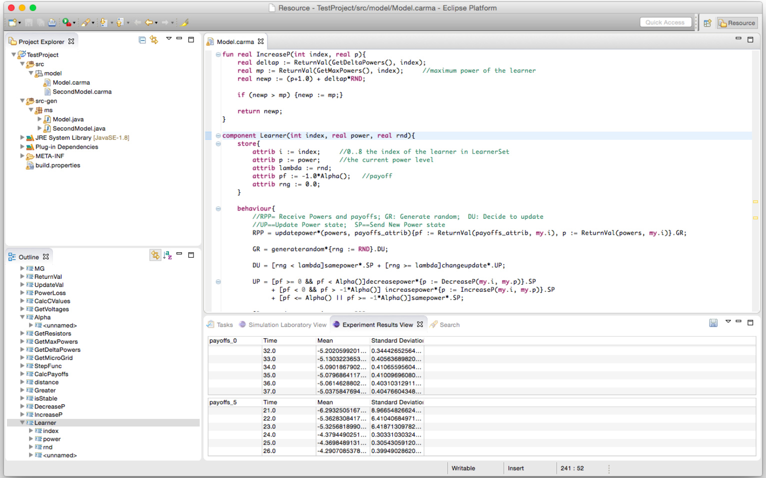
Time dependent rates...

Expression `now` can be used to model time-dependent environment:

```
rate {
  get { return get_rate; }
  ret { return ret_rate; }
  move* { return move_rate; }
  arrival* {
    if (global.users<TOTAL_USERS) {
          if (now<360) { return 4*arrival_rate;
          } else { return arrival_rate/2; }
    } else { return 0.0; }
  }
}
```

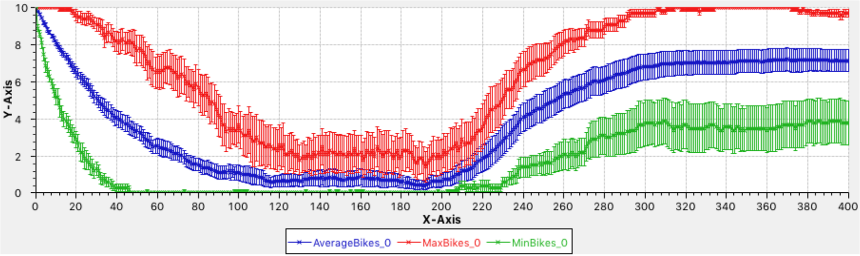
The CARMA Eclipse Plug-in

LIVE DEMO

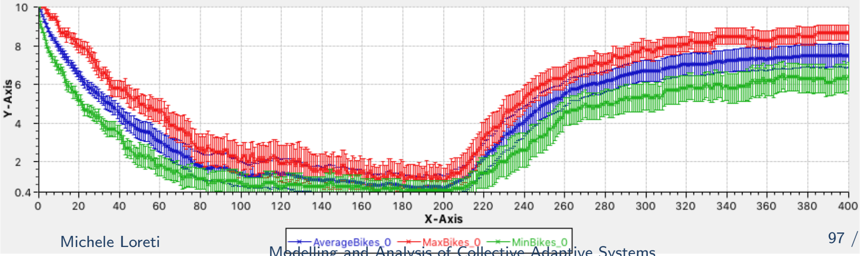


CaSL: BSS model Analysis

Competitive Stations



Collaborative Stations



Outline

Specifying spatial models in CaSL

In CARMA/CaSL, there is a separation between system behaviour, identified by components, from the specification of the context (the environment) which regulates the interaction of components.

Specifying spatial models in CaSL

In Carma/CaSL, there is a separation between system behaviour, identified by components, from the specification of the context (the environment) which regulates the interaction of components.

It is important to avoid hardcoding the environment inside the components' store or behaviour!

Specifying spatial models in CaSL

In Carma/CaSL, there is a separation between system behaviour, identified by components, from the specification of the context (the environment) which regulates the interaction of components.

It is important to avoid hardcoding the environment inside the components' store or behaviour!

When system behaviour is clearly separated from the environment the resulting models are flexible in terms of being able to easily represent the performance of the components when subjected to different kinds of external conditions.

Specifying spatial models in CASL

In CARMA/CASL, there is a separation between system behaviour, identified by components, from the specification of the context (the environment) which regulates the interaction of components.

It is important to avoid hardcoding the environment inside the components' store or behaviour!

When system behaviour is clearly separated from the environment the resulting models are flexible in terms of being able to easily represent the performance of the components when subjected to different kinds of external conditions.

CASL includes constructs that can be used to define spatial structures, based on graphs, in system models.

Space declaration. . .

Space declaration...

The space where a system operates can be defined as a graph in which edges have labels that contain tuples of properties.

```
space <name>( <parameters> ) {
    locations <location_dec>
    edges { ... }
    labels { ... }
}
```

A grid can be declared as follows:

```
space grid( int height , int width ) {
    ...
}
```

Space declaration. . .

Locations. . .

Locations in the space can be declared either as an enumeration. . .

```
locations { <name_1 >,...,<name_n> }
```

Space declaration...

Locations...

Locations in the space can be declared either as an enumeration...

```
locations { <name_1>,...,<name_n> }
```

...or as a tuple of features

```
locations: <type_1>*...*<type_k>;
```

where `<type_i>` indicates possible values for the corresponding element in the tuple.

Space declaration...

Locations...

Locations in the space can be declared either as an enumeration...

```
locations { <name_1>,...,<name_n> }
```

...or as a tuple of features

```
locations: <type_1>*...*<type_k>;
```

where <type_i> indicates possible values for the corresponding element in the tuple.

Our grid can be extended:

```
space grid( int width , int height ) {
   locations: [0:width]*[0:height];
   ...
}
```

Space declaration...

Edges...

In the `edges` block the edges of our model can be listed either explicitly...

```
v1 -> v2: w; //Directed edge;
v1 <-> v2: w; //Undirected edge;
```

...or via the properties of involved locations:

```
<?x1,...,?xn>:g -> <e1,...,en>: w; //Directed edge;
<?x1,...,?xn>:g <-> <e1,...,en>: w; //Undirected edge;
```

Space declaration...

Edges...

In the `edges` block the edges of our model can be listed either explicitly...

```
v1 -> v2: w; //Directed edge;
v1 <-> v2: w; //Undirected edge;
```

...or via the properties of involved locations:

```
<?x1,...,?xn>:g -> <e1,...,en>: w; //Directed edge;
<?x1,...,?xn>:g <-> <e1,...,en>: w; //Undirected edge;
```

In our grid model:

```
edges {
    <?x,?y>: x<width-1 <-> <x+1,y>: 1.0;
    <?x,?y>: y<height-1 <-> <x,y+1>: 1.0;
}
```

Space declaration. . .

Labels. . .

Labels are used to identify sets of locations. Vertices can be associated with a label either via enumeration. . .

```
<label_name>: v1,...,vn;
```

or by declaring their properties:

```
<label_name>: <?x1,...,?xn> [g];
```

Space declaration...

Labels...

Labels are used to identify sets of locations. Vertices can be associated with a label either via enumeration...

```
<label_name>: v1,...,vn;
```

or by declaring their properties:

```
<label_name>: <?x1,...,?xn> [g];
```

In our grid model we a label can be used to identify locations at the border or in the center:

```
labels {
  border: <?x,?y> [x==0||y==0||x==width-1||y==height-1];
  center: <?x,?y> [x==width/2 && y==height/2];
}
```

Space declaration...

Example: Grid Model

```
space grid( int width , int height ) {
  locations [0:width]*[0:height]
  edges {
    <?x,?y>  [ x<width-1 ]   <-> <x+1,y> : 1.0;
    <?x,?y>  [ y<height-1 ]  <-> <x,y+1>: 1.0;
  }
  labels {
    border: <?x,?y> [ x==0 || x==width- || y==0
                            || y==height-1 ];
    center: <?x,?y> [ x==width/2 && y==height/2 ];
  }
}
```

CaSL: expressions on locations

Let `l` be an expression of type location:

- `loc` is the attribute used to refer to component location;
- `l.post` indicates the of locations in the postset of `l`;
- `l.pre` indicates the of locations in the preset of l;
- `l.label_name` is a boolean expression that can be used to check if location `l` has label `label_name`;
- `locations` is used to refer to the list of all locations in the space;
- `label_name` refers to the set of locations labeled `label_name`.

Outline

Example: Smart Taxi System

Example: Smart Taxi System

System description:

- We consider a set of taxis operating in a city, providing service to users;
- Both taxis and users are modelled as components.
- The city is subdivided into a number of patches arranged in a grid over the geography of the city.
- The users arrive randomly in different patches.
- After arrival, a user makes a call for a taxi and then waits in that patch until they successfully engage a taxi and move to another randomly chosen patch.
- Unengaged taxis move about the city, influenced by the calls made by users.

J.Hillston and M.Loreti. Specification and analysis of open-ended systems with CARMA. In LNCS 9068, 2015.

Taxi Scenario

We consider a grid of 3×3 patches, i.e., a set of locations (i, j) where $0 \leq i, j \leq 2$, and two different scenarios:

Assumptions:

- Users arrive in all the patches at the same rate;
- Users at the periphery travel to the centre;
- Users at the centre travel to a randomly selected patch in the periphery.

Taxis and Users: stores

Taxis

- **loc**: identifies current taxi location;
- **free**: describes if a taxi is free ($free = true$) or engaged ($free = false$);
- **dest**: if occupied, this attribute indicates the destination of the taxi journey.

Taxis and Users: stores

Taxis

- **loc**: identifies current taxi location;
- **free**: describes if a taxi is free ($free = true$) or engaged ($free = false$);
- **dest**: if occupied, this attribute indicates the destination of the taxi journey.

Users

- **loc**: identifies user location;
- **dest**: indicates user destination.

```
component User(location dest){
    store{
        loc = g;
        dest = dest;
    }
    behaviour{
        Wait = call*[true]<loc>.Wait
               + take[my.loc == loc] <my.dest>.kill;
  }

    init{
        Wait
    }
}
```

Taxi agent

```
component Taxi( ){
  store{ location dest = none;
    free = true;
  }
    behaviour{
    F = [free] take[true](x){dest = x;
                                free = false;}.G
      + call[(my.loc != pos)](pos)
                              {dest = pos;}.G;
    G = move*[false]<>{
            loc = dest;
            dest = none;
            free = true;
          }.F;
    }
    init{ F }
}
```

Modelling arrivals

```
component Arrival(){

  store{
  }

  behaviour{
    A = arrival*[false]<>.A;
  }

  init{
    A
  }
}
```

The collective

Arrivals process for users

```
collective TaxiCollective {

  for l in locations {
    new Taxi()@l< K >;
    new Arrival()@l;
  }

}
```

Taxi Scenario

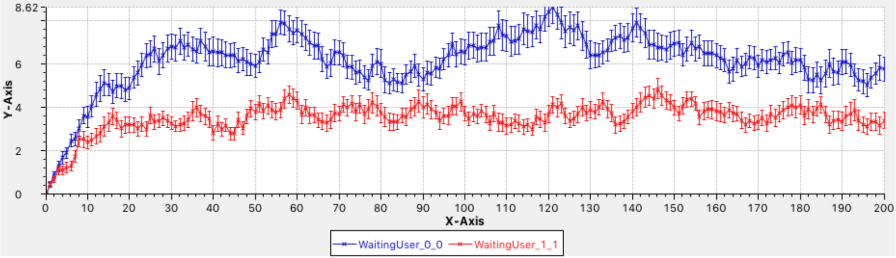
```
system Scenario1{
  space Grid( 3 , 3 );
    collective TaxiCollective
    environment{
      weight {
            default { return 1.0; }
      }
      prob{
            call* { return 1-P_LOST; }
            default { return 1.0; }
      }
      ...
```

Taxi Scenario

```
...
    rate{
        take { return R_T; }
        call* { return R_C; }
        move* { return Mrate(sender._loc,sender.dest); }
        arrival* { return R_A * (1.0 / real( SIZE * SIZE )
            ) ; }
        default { return 0.0; }
        }
        update{
            arrival* {
                new User()@( sender.loc )
            }
        }
    }
}
```

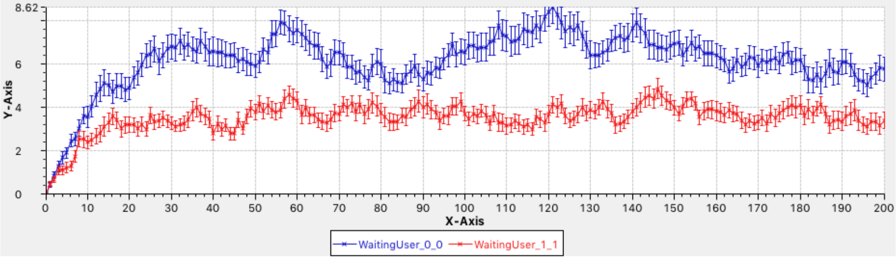
Simulation results

Average number of users waiting at (1,1) and (0,0)



Simulation results

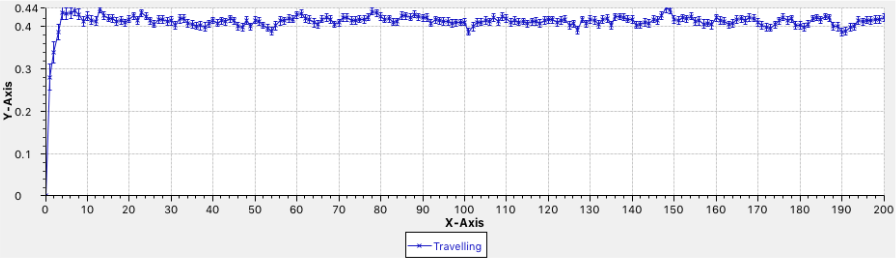
Average number of users waiting at $(1,1)$ and $(0,0)$



System is almost balanced.

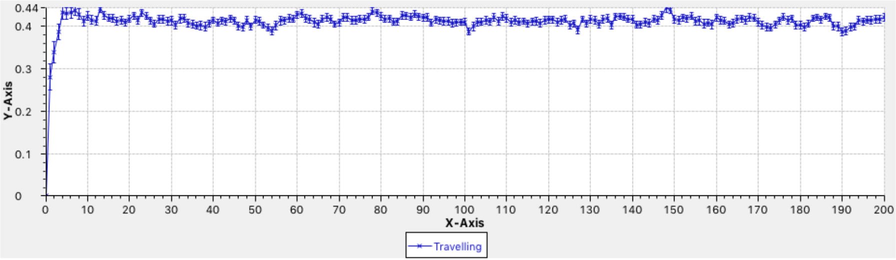
Simulation results

Proportion of transit taxis.



Simulation results

Proportion of transit taxis.



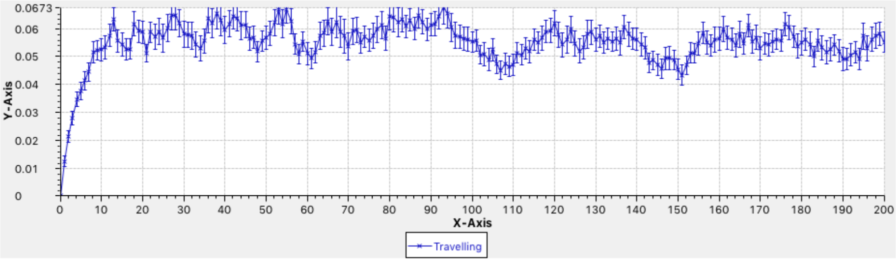
A large fraction of taxis are travelling empty!

Alternative model...

Each user does not call a taxi with a broadcast, but he/she uses a unicast ouput.

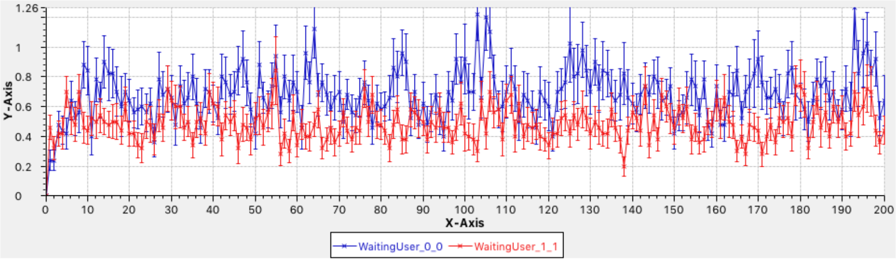
Alternative model. . .

Each user does not call a taxi with a broadcast, but he/she uses a unicast ouput.



Alternative model. . .

Proportion of free taxis at (1,1) and (0,0) and in transit



Outline

Concluding remarks

- Collective Systems are an interesting and challenging class of systems to design and construct.

Concluding remarks

- Collective Systems are an interesting and challenging class of systems to design and construct.
- Their role within infrastructure, such as within smart cities, make it essential that quantitative aspects of behaviour are taken into consideration, as well as functional correctness.

Concluding remarks

- Collective Systems are an interesting and challenging class of systems to design and construct.
- Their role within infrastructure, such as within smart cities, make it essential that quantitative aspects of behaviour are taken into consideration, as well as functional correctness.
- The complexity of these systems poses challenges both for model construction and model analysis.

Concluding remarks

- Collective Systems are an interesting and challenging class of systems to design and construct.
- Their role within infrastructure, such as within smart cities, make it essential that quantitative aspects of behaviour are taken into consideration, as well as functional correctness.
- The complexity of these systems poses challenges both for model construction and model analysis.
- CARMA (and CASL) aims to address many of these challenges, supporting rich forms of interaction, using attributes to capture explicit locations and the environment to allow adaptivity.

thank you!