Collective Systems

Collective Systems (CS) : systems of entities.
Collective Adaptive Systems (CAS) : CS that are also *adaptive* to their environment.

We are surrounded by collective systems both in nature world and in the man-made world.

A CAS can be viewed as being made up of a large number of **interacting entities** and where each entity may have its own **properties**, **objectives**, **actions**. At the system level these combine to create the collective behaviour.

> The **behaviour** of the system is thus dependent to the behaviour of the individuals.
> The behaviour of the individuals will be influenced by the state of the system.

Such systems are often embedded in our environment and need to **operate without centralized 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***.

Informatics environment

Robin Milner coined the term of **informatics environment,** in which 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.

Quantitative Modelling

Quantitative Analysis aims to understand or predict prehaviour or events through the use of mathematical measurements and calculations, statistical modeling and research. Aims to represent a given reality in terms of a numerical value.

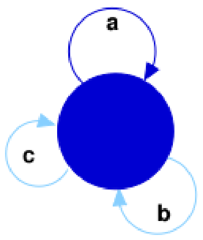
Quatitative vs qualitative analysis. Quantitative analysis is often combined with the complementary research and evaluation tool of qualitative analysis. Qualitative models do not require mathematical formilistm, but are used to “draw, diagram or represent visually ideas, hunches, perceived patterns or relationships between parts of the project, discoveries in data, etc.

Performance modelling aims to construct models of the dynamic behaviour of systems in order to support the *fair* and *efficient* sharing of resources.

Markovian-based discrete event models have been applied to computer systems since the mid-1960s and communication systems since the early 20th century.

TODO: catene di Markov

https://it.wikipedia.org/wiki/Processo_stocastico
https://it.wikipedia.org/wiki/Processo_markoviano



Various formalisms have been designed for capturing such behaviour.

One of the motivation behind performance modeling is the system **capacity planning** (e.g. what strategy can I use to maintain supply-demand balance within a smart electricity grid).

The size and complexity of real systems makes the direct construction of discrete state models costly and error-prone.

For the last three decades there has been substantial interest in applying **formal modeling techniques** enhanced with information about timing and probability. From these high-level system descriptions the underlying mathematical model (*Continuous Time Markov Chain* (**CTMC**)) can be automatically generated.

Primary examples include: **Stochastic Petri Nets** and **Stochastic Process Algebras**.

Stocastic Process Algebras

Models are constructed from **components** which engage in **activities**. Activities have a **name** and a **rate**.

The rate defines an **exponential distribution** which means that the duration of an activity is a random variable.

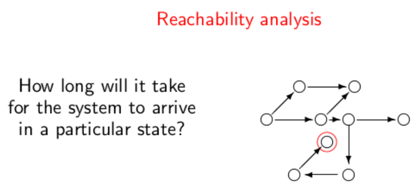
TODO: rivedi exponential distribution

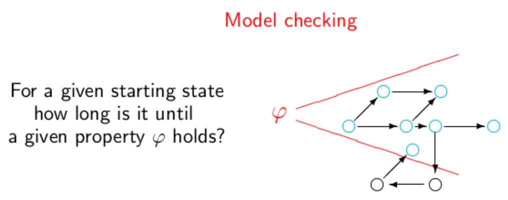
A small set of language constructs determine how the model will evolve.

The language is used to **generate a CTMC** for performance modeling (using the semantics).

$$\begin{tikzcd}
\text{SPA MODEL} \arrow[r, "\text{SOS rules}"] & \text{LABELLED TRANSITION SYSTEM} \arrow[r, "\text{state transition}", "\text{diagram}"'] & \text{CTMC } \mathbf{Q}
\end{tikzcd}$$

Qualitative verification can now be complemented by quantitative verification.

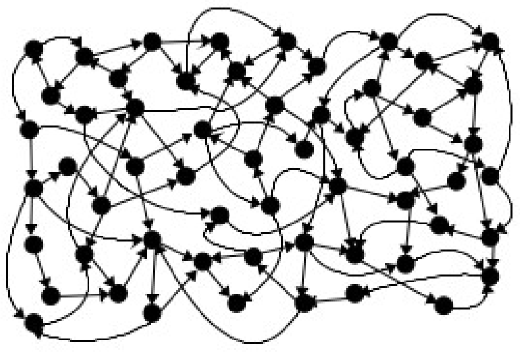

Reachability analysis
How long will it take for the system to arrive in a particular state?


Model checking
For a given starting state how long is it until a given property φ holds?
φ

Solving discrete state models

Using the SOS semantics a SPA model is mapped to a CTMC with global states determined by the local states of all the participating components.

When the size of the state space is not too large they are amenable to **numerical solution** (linear algebra) to determine a **steady state** or **transient probability distribution**.

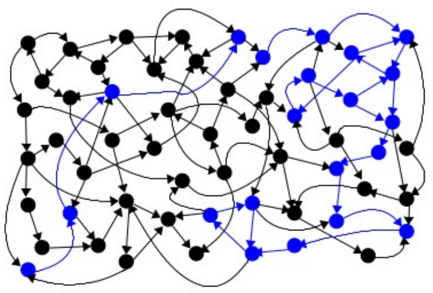


$$Q = \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,N} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,N} \\ \vdots & \vdots & & \vdots \\ q_{N,1} & q_{N,2} & \cdots & q_{N,N} \end{pmatrix}$$

$$\pi(t) = (\pi_1(t), \pi_2(t), \ldots, \pi_N(t))$$

$$\pi(\infty)Q = 0$$

Alternatively they may be studied using **stochastic simulation**. Each run generates a single trajectory through the state space. Many runs are needed in order to obtain average behaviours.



As the size of the state space becomes large it becomes infeasible to carry out numerical solution and extremely time-consuming to conduct stochastic simulation.

When the population sizes become large we can make a mean-field approximation, approximating the average trajectory of the CTMC by a set of ordinary differential equations.

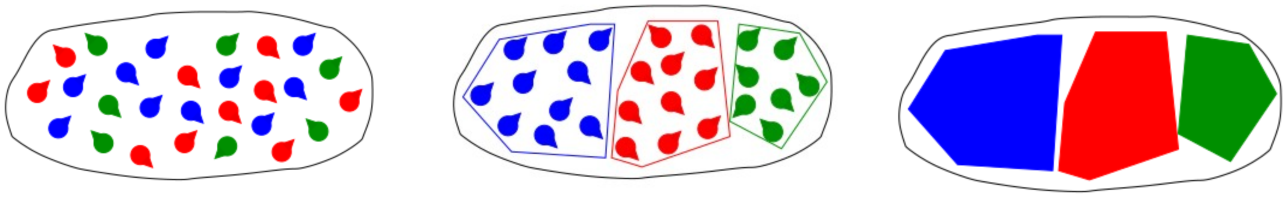
The Fluid Approximation Alternative

A key feature of collective systems is the existence of populations of entities who share certain characteristics. High-level modelling formalisms allow this repetition to be captured at the high-level rather than explicitly.

We can shift attention to the **populations** rather than the **individual** entities, and then consider the average behaviour within a population.

Ceasing to distinguish between instances of components we form an **aggregation** or **counting abstraction** to reduce the state space. We now disregard the identity of components.

Even better reductions can be achieved when we no longer regard the components as individuals.



A shift in perspective allows us to model the interactions between individual components but then only consider the system as a whole as an interaction of populations

To characterise the behaviour of a population we calculate the proportion of individuals within the population that are exhibiting certain behaviours rather than tracking individuals directly. Furthermore we make a continuous or fluid approximation of how the proportions vary over time.

Work over the last twenty years on stochastic process algebra provides a solid basic framework for modeling 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 (e.g. pherormone 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. But languages designed for other purposes typically contain too much detail to be used as the basis of quantitative modeling and analysis.

Modeling 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 modeling 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** (same place).

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.

CARMA

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

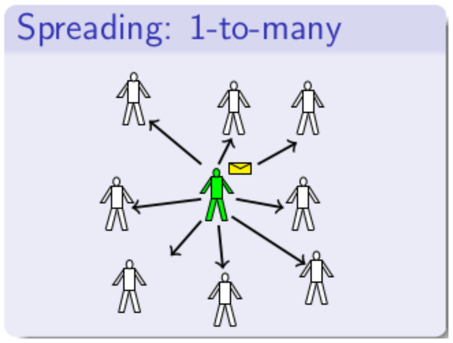
A key element of the QUANTICOL framework is the language, **CARMA** (Collective Adaptive Resource-sharing Markovian Agents), a new Stocastic Process Algebra, which handles:

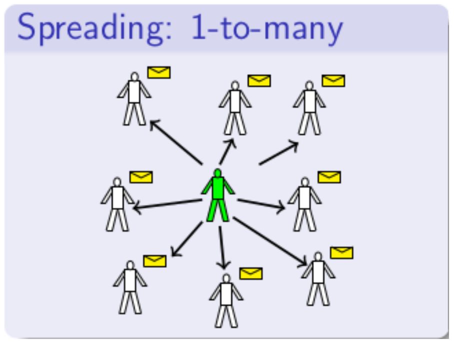
- The behaviours of agents and their interactions;
- The global knowledge of the system and that of its agents;
- The environment where agents operate:
  - taking into account open ended-ness and adaptation;
  - taking into account resources, locations and visibility/reachability issues.

Interaction patterns in CAS

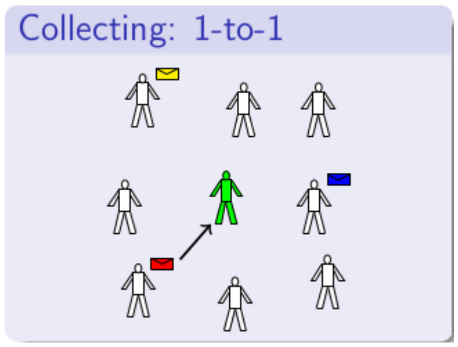
Typically, CAS exhibit two kinds of interaction pattern:

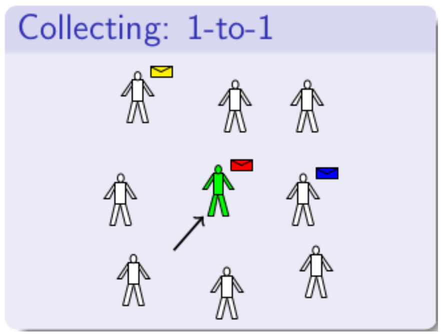
Spreading: one agent **spreads** relevant information to a **given group** of other agents





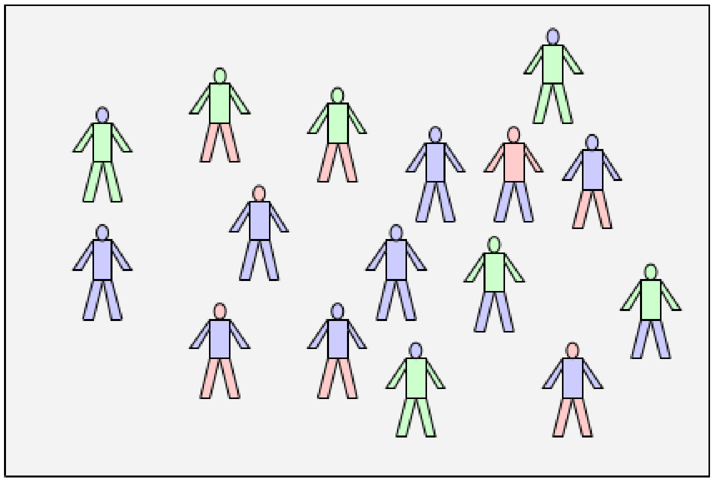
Collecting: one agent **changes its behaviour** according to data collected from **one agent** belonging to a **given group** of agents.





CARMA perspective:

- Collective: the set of agents (e.g. the single persons)
- Attributes: the agents characteristics (e.g. the persons dress)
- Environment: the context where agents are placed (e.g. a city)



Thus a CARMA system consists of a collective operating in an environment.

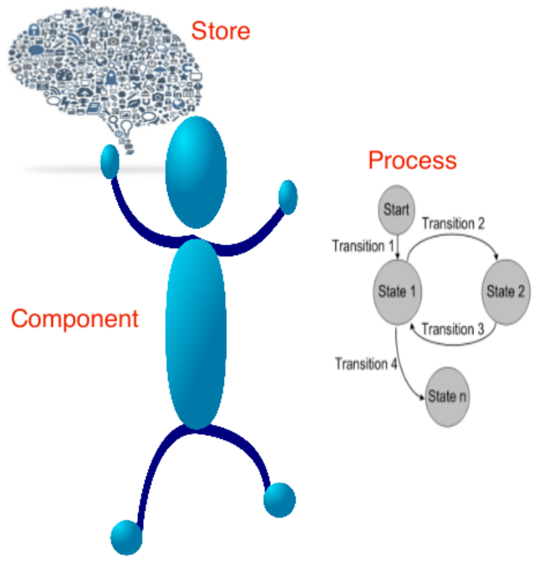
The **collective** (N) is composed by a set of components, i.e. the Markovian **agents** that compete and/or cooperate to achieve a set of given tasks. Models the behavioural part of a system.

The **environment** (E) models the rules intrinsic to the context where agents operate. Mediates and regulates agent interactions.

Modelling Agents

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**, modeling agent **knowledge**



The participants of an interaction are identified via **predicates**… the counterpart of a communication is selected according its **properties**.

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**.

Syntax

```
act ::=  α*[π]<e>σ  Broadcast output
      |  α*[π](x)σ  Broadcast input
      |  α[π]<e>σ   Unicast output
      |  α[π](x)σ   Unicast input
```

- α is an **action type**;
- π is a **predicate**;
- <e> is the **output** values vector
- (x) is the **input** values vector
- σ is the **effect** of the action on the store.

Example

```
Lecturer = teach*[awake = true]<fact>{} ...
         + coffee*[true]><.>{boring := false} ...

Student = teach*[boring = false](fact){know := know + fact} ...
        + coffee*[true](.){awake := true} ...
```

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**.

```
move*[π]<v>{x := x + U(−1, +1)}
```

Processes running in the same component can implicitly interact via the local store (shared memory); Updates are instantaneous.

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:

```
α*[(x > 0) ∧ (trust_level < 3)](x)σ.P
```

Example of interactions in CARMA

(Pattern matching can be ecncoded in carma)

Broadcast synchronization (???)

```
( stop*[bl < 5%]<v>$\sigma_1$.P , {role = “master ”}) ||
( stop*[role = “master ”](x)$\sigma_2$.$Q_1$ , {bl = 4%}) ||
( stop*[role = “super ”](x)$\sigma_3$.$Q_2$ , {bl = 2%}) ||
( stop*[T](x)$\sigma_4$.$Q_3$ , {bl = 2%})

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

Unicast synchronization (???)

```
( stop[bl < 5%]<•>$\sigma_1$.P , {role = “master ”}) ||
( stop[role = “master ”](x)$\sigma_2$.$Q_1$ , {bl = 4%}) ||
( stop[role = “super ”](x)$\sigma_3$.$Q_2$ , {bl = 2%}) ||
( stop[T](x)$\sigma_4$.$Q_3$ , {bl = 2%})

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

Modeling the environment

The environment is used to model the intrinsic rules that govern the **physical context**.

Interactions between components can be affected by the environment (e.g. a wall can inhibit wireless interactions or two components are too distant to interact).

The environment is represented as a pair (γ, ρ), where

- γ is a **global store** that models the overall state of the system;
- ρ is the evolution rule that regulates component interactions (receiving probabilities, action rates,. . .).

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.

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

- μ_p (γ_s , γ_r , α): the probability that a component with store γ_r can receive a broadcast message α from a component with store γ_s ;
- μ_w (γ_s , γ_r , α): 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 ;
- μ_r (γ_s , α) computes the execution rate of action α executed at a component with store γ_s ;
- μ_u (γ_s , α) determines the updates on the environment (global store and collective) induced by the execution of action α at a component with store γ_s .

Operational Semantics of Process Algebras

behaviour of a PA is classically represented via **transition relations**, e.g. α.P −α→ P.

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

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

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

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

However, in Stochastic Process Algebras (SPA), like CARMA, there is not any form of non-determinism...
… 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).

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 caracterising 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 synchronization.

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:

- the function C that describes the behaviour of a single component;
- the function N_ε builds on C to describe the behaviour of collectives;
- the function S_t that shows how CARMA systems evolve.

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

The behaviour of a single component is defined by a function

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

where Lab denotes the set of transition labels:

$$\begin{array}{rll}
\ell ::= & \alpha^*[\pi_s]\langle v\rangle,\gamma & \text{Broadcast output} \\
\mid & \alpha^*[\pi_s](v),\gamma & \text{Broadcast input} \\
\mid & \alpha[\pi_s]\langle v\rangle,\gamma & \text{Unicast output} \\
\mid & \alpha[\pi_s](v),\gamma & \text{Unicast input}
\end{array}$$

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

CARMA Specification Language (CaSL)

To facilitate the use of CARMA in the specification and analysis process of CAS, QUANTICOL developed:

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

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.

Basic data types

boolean	: `bool`
integer	: `int`
real values	: `real`

To convert from one type to the other explicit cast is required: int(e), float(e).

Collections

Sets	: list<type> name = [: exp_1 , … , exp_n :]
Arrays	: set<type> name = {: exp_1 , … , exp_n :}

To concatenate two lists: `list1 + list2`

Elements in a list can be accessed using the index: `list[i]`
Built-in functions `head()` and `tail()` can be used to access the first and last elements, respectively.

Union of sets:	`set1 || set2`
Intersection of sets:	`set1 && set2`
Difference:	`set1 - set2`

The built-in function `size(collection)` returns the number of elements in the collection.

Custom data types

Enumerations	: enum name = $elem_1$, … , $elem_n$
Records	: record name = [$type_1$ $field_1$, … , $type_n$ $field_n$]

Records are aggregated data structures (structs).
Record example:

Declaration:	`record pos = [ int x, int y ]`
Assignment:	`pos = [ x := 1, y := 3 ]`

Given a reference of record type each field can be accessed using the dot notation, i.e.
`pos.x = 4`

Expressions

In CaSL syntax of expressions includes:

- standard arithmetic and logical operations (+ , − , ...)
- common functions (log, sin, ...)
- conditional expression (exp_1 ? exp_2 : exp_3)
- special value now, indicating the current time
- literals (none, true, false, $(0\text{-}9)^+$, $(0\text{-}9)^*.(0\text{-}9)^+$)
- references

References

An expression can contain references to global or local attributes, constants, parameters or variables. Attributes references can be prefixed with the keywords *my, global, sender, receiver*, to specify the *store* used to evaluate an attribute.

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

Operators in expressions

- Arithmetic Operators: + , - , * , +
- Unary Operators: + , - , !
- Boolean 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 , ...

Constants and functions

A CARMA specification can also contain constants and functions declarations:

```
const name = expression ;
```

In a constant the type is inferred from the definition (implicit typing).

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

Built-in well known constants and functions: `true, false, E, PI, MAXINT, MININT, MAXREAL, MINREAL, abs(e), exp(e), pow(e1, e2), max(e1, e2), min(e1, e2), …`

Function statements

Variable declaration and initialization

```
type var = exp;
```

Variable value assignment

```
var = exp;
```

Variable return

```
return exp;
```

If-then-else

```
if (exp) { … } else { … }
```

Iterators

```
for (var = exp; exp2; exp3) { … }
for var in exp { … }
```

Example

```
const inst MAX_VAL = 100;

fun int maxVal( set<int> s )
{
    if (size(s) == 0) {
        return 0;
    }
    int res = MAX_VAL;
    for v in s {
        res = max(res, v);
    }
    return res;
}
```

Component prototypes

A component prototype defines the general structure of a component:

```
component name ( type1 arg1 , … , typen typen ) {
    store {
        attrib namei = expri; ...
    }
    behaviour {
        proci = pdefi; ...
    }
    init { P1 | … | Pw }
}
```

The block `behaviour` is used to define the components behaviour. It consists of a sequence of **process definitions**.

```
behaviour {
    proc1 = pdef1;
    …
    procn = pdefn;
}
```

Associates each **process name** with alternative actions.

Output Actions

```
[ guard ] act [ pred ] < x1 , … , xn > {
    exp1;
    …
    expm;
}
```

Input Actions

```
[ guard ] act [ pred ] ( x₁, … , xₙ ) {
    exp₁;
    …
    expₘ;
}
```

Example. Bike Sharing System.

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 and return it to any other station close to their planned destination.
- 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.

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

- parking stations;
- users.

Station description

Attributes

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

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 to get and return a bike are modeled via **unicast output** over `get` and `ret`.

The `get` is enabled when there are bikes available (`my.available > 0`), the `ret` is enabled when there are available slots (`my.available < my.capacity`).

Component

```
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 }
    }
```

User description

Attributes

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

Behaviours

Each user can be in three different states...

... P, denoting a pedastrian:

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

... a pedastrian 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;
```

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

Component

```
    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 }
    }
```

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.

Collective

Block collective can be used to define groups of components:

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

Example. Bike Sharing Station

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

Environment

The block environment is used to define the system environments

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

Block store defines the global store (global variables) while blocks weight, prob, rate and update define the **evolution rule** ρ.

Example. Bike Sharing Station

We use a global attribute to count the number of active users in the system.

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

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}) }
}
```

The block prob is used to define the probability that a component receives a broadcast message.

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

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; }
        }
    }
```

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

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

System

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

```
    system name {
        collective collective
        environment { … }
    }
```

Example. Bike Sharing Station

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

Measures

To extract observations from a model, a CaSL specification contains a set of measures.

```
    measure name (type₁ var₁, … , typeₙ varₙ) = expr;
```

Example. Bike Sharing Station

```
    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 };
```

Space models

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

`CaSL sources:` http://quanticol.github.io/CARMA/examples.html

CARMA 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, at a rate that depends on the specific time of day.
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.

Both kinds of component use the local store to publish the relevant data that will be used to represent the state of the agent.

Taxis store attributes:

loc: identifies current taxi location;
busy : ranging in {0, 1} describes if a taxi is busy;
dest: if occupied, this attribute indicates the destination of the taxi journey.

Users store attributes:

loc: identifies the user locations
dest: indicates the user destination

```
process User =
    Wait : call*[T]<my.loc.x, my.loc.y>.Wait
         + take[loc.x == my.loc.x ∧ loc.y == my.loc.y]<my.dest.x,my.dest.y>.kill
endprocess

process Taxi =
    F: call*[my.loc.x ≠ posx ∧ my.loc.y ≠ posy](posx, posy )
                 { dest := [x := posx, y:= posy] }.G
     + take[T](posx, posy)
                 { dest := [x := posx, y := posy], busy := 1 }.G
    G : move*[⊥]<o>
                 { loc := dest, dest := [x := 3, y := 3], busy := 0 }.F
endprocess
```

The arrivals process has a single attribute `loc`

```
process Arrivals =
      A : arrival*[⊥]<o>.A
endprocess
```

This process is executed in a separate component where attribute loc indicates the location where the users arrives. The precise role of this process will be clear when the environment is described.

The environment consists of a global store and an evolution rule, and provides the context in which the components operate.

In this model, the global store just contains constants related to the rates of the key actions in the system, take and call.

The evolution rule consist of three functions:

μ_p : determines the probability of an action, capturing how the current state of the system influences the communication within the system;
μ_r : defines the rates of actions in the system; again this may depend on the current state of the system, capturing adaptivity;
μ_u : allows the global store and/or the collective to be updated after an action, again capturing adaptivity.

APPENDIX

Process Algebra

In computer science, the **process calculi** (or **process algebras**) are a diverse family of related approaches for formally modeling concurrent systems. Process calculi provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes. They also provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalences between processes.

There are several features that all process calculi have in common:

- Representing interactions between independent processes as communication (message-passing), rather than as modification of shared variables.
- Describing processes and systems using a small collection of primitives, and operators for combining those primitives.
- Defining algebraic laws for the process operators, which allow process expressions to be manipulated using equational reasoning.

Parallel composition

Parallel composition of two processes *P* and *Q*, usually written *P|Q*, is the key primitive distinguishing the process calculi from sequential models of computation. Parallel composition allows computation in *P* and *Q* to proceed simultaneously and independently. But it also allows interaction, that is synchronization and flow of information from *P* to *Q* (or vice versa) on a channel shared by both. Crucially, an agent or process can be connected to more than one channel at a time.

Channels may be synchronous or asynchronous. In the case of a synchronous channel, the agent sending a message waits until another agent has received the message. Asynchronous channels do not require any such synchronization.

Communication

Interaction can be (but isn't always) a *directed* flow of information. That is, input and output can be distinguished as dual interaction primitives. Process calculi that make such distinctions typically define an input operator (*e.g.* $x(v)$) and an output operator (*e.g. x<y>)*, both of which name an interaction point (here *x*) that is used to synchronize with a dual interaction primitive.

Information should be exchanged, it will flow from the outputting to the inputting process. The output primitive will specify the data to be sent. In *x<y>*, this data is y. Similarly, if an input expects to receive data, one or more bound variables will act as place-holders to be substituted by data, when it arrives. In *x(v)*, *v* plays that role.

Sequential composition

Sometimes interactions must be temporally ordered. For example, it might be desirable to specify algorithms such as: *first receive some data on x and then send that data on y*. In process calculi, the sequentialisation operator is usually integrated with input or output, or both. For example, the process *x(v).P* will wait for an input on *x*. Only when this input has occurred will the process *P* be activated, with the received data through *x* substituted for identifier *v*.

Reduction semantics

The key operational reduction rule, containing the computational essence of process calculi, can be given solely in terms of parallel composition, sequentialization, input, and output. The details of this reduction vary among the calculi, but the essence remains roughly the same. The reduction rule is:

```
x<y>.P|x(v).Q → P|Q[y/v]
```

The interpretation to this reduction rule is:

- The process *x<y>.P* sends a message, here *y*, along the channel *x*. Dually, the process *x(v).Q* receives that message on channel *x*.
- Once the message has been sent, *x<y>.P* becomes the process *P*, while *x(v).Q* becomes the process *Q[y/v]*, which is *Q* with the place-holder *v* substituted by *y*, the data received on *x*.

The class of processes that *P* is allowed to range over as the continuation of the output operation substantially influences the properties of the calculus.

Choice

The agent behaves as A or B depending on which acts first

```
P + Q
```

Restriction

The set of labels M is hidden from outside agents

```
A \ M
```

Relabeling

In this agent label a_1 is renamed a_0

$A[a_1/a_0,...]$