QUANTICOL

A Quantitative Approach to Management and Design of Collective and Adaptive Behaviours

quanticol
http://www.quanticol.eu

D4.2

CAS-SCEL semantics and implementation

Revision: 1.0

Author(s): Vincenzo Ciancia (ISTI), Rocco De Nicola (IMT), Jane Hillston (UEDIN), Diego Latella (ISTI), Michele Loreti (IMT), Mieke Massink (ISTI)

Due date of deliverable: Month 30 (September 2015)
Actual submission date: Sep 30, 2015
Nature: R. Dissemination level: PU

Funding Scheme: Small or medium scale focused research project (STREP)
Topic: ICT-2011 9.10: FET-Proactive 'Fundamentals of Collective Adaptive Systems' (FOCAS)
Project number: 600708

Coordinator: Jane Hillston (UEDIN)
e-mail: `Jane.Hillston@ed.ac.uk`
Fax: +44 131 651 1426

Part. no.	Participant organisation name	Acronym	Country
1 (Coord.)	University of Edinburgh	UEDIN	UK
2	Consiglio Nazionale delle Ricerche – Istituto di Scienza e Tecnologie della Informazione "A. Faedo"	CNR	Italy
3	Ludwig-Maximilians-Universität München	LMU	Germany
4	Ecole Polytechnique Fédérale de Lausanne	EPFL	Switzerland
5	IMT Lucca	IMT	Italy
6	University of Southampton	SOTON	UK
7	Institut National de Recherche en Informatique et en Automatique	INRIA	France

COOPERATION

Executive summary

At the end of the first year of the QUANTICOL project, we identified the *linguistic primitives* and the *interaction patterns* (such as broadcast communication or anonymous interaction) that are needed to model the QUANTICOL case studies and, more generally, in Collective Adaptive Systems (CAS) design.

Starting from these *primitives* and *interaction patterns*, in the second reporting period, we have worked on the definition of syntax and semantics of the CAS-SCEL language that we named CARMA (*Collective Adaptive Resource-sharing Markovian Agents*). CARMA is a language specifically developed for the specification and analysis of CAS, with the specific objective to support quantitive evaluation and verification. CARMA combines the lessons learnt from other stochastic process algebras such as PEPA [21], EMPA [2], MTIPP [20] and MoDEST [3], with those learnt from languages specifically designed to model CAS, such as SCEL [13], the AbC calculus [1], PALOMA [14], and the Attributed Pi calculus [22], which feature attribute-based communication and explicit representation of locations.

To support simulation of CARMA models a prototype simulator has been also developed. This simulator, which has been implemented in Java, can be used to perform stochastic simulation and can be used as the basis for implementing other analysis techniques. An Eclipse plug-in for supporting specification and analysis of CAS in CARMA has also been developed. Thanks to this plug-in, CARMA systems can be specified by means of an appropriate high-level language. The high level specification is mapped to a process algebra to enable qualitative and quantitive analysis of CAS during system development by following specific design workflows and analysis pathways. In this deliverable we will describe the so-called CARMA *Specification Language* while in D5.2 an overview of the CARMA Eclipse Plug-in, together with an example of use, is provided.

In this document, we first introduce the syntax of CARMA and its operational semantics; then we describe the CARMA tools and show how the formalism can be used to support the quantitative analysis of a simple scenario. Then, we provide a brief illustration of how CARMA could be extended in order to provide a flexible and structured mechanism for defining common spatial aspects of CAS. The document ends with a description of the direction of future work and the relationship with the work done in the other workpackages of the project.

Contents

1 Introduction

In the first year deliverable [4] we reported on the progress made towards the definition of a formal language for CAS. In particular, the deliverable reported about the *design principles* and the identification of *primitives* and *interaction patterns* that are needed in CAS design. The focus was on identifying abstractions and linguistic primitives for collective adaptation, locality representation, knowledge handling, and system interaction and aggregation. To identify these abstractions and linguistic primitives, we relied on various formalisms that QUANTICOL partners had previously developed and experimented with them to model simple CAS. At the end of this work a general consensus was reached in the project that, to be effective, any language for CAS should provide:

- Separation of knowledge and behaviour,
- Control over abstraction levels,
- Bottom-up design,
- Mechanisms to take into account the environment,
- Support for both global and local views, and
- Automatic derivation of the underlying mathematical model.

In the present deliverable, we report on the work done for the definition of a language, developed specifically to support the specification and analysis of CAS, with the particular objective of supporting quantitive evaluation and verification. We named this language CARMA, Collective Adaptive Resource-sharing Markovian Agents. CARMA combines the lessons we learnt from other stochastic process algebras such as PEPA [21], EMPA [2], MTIPP [20] and MoDEST [3], with those learnt from languages specifically designed to model CAS, such as SCEL [13], the AbC calculus [1], PALOMA [14], and the Attributed Pi calculus [22], which feature attribute-based communication and explicit representation of locations.

Compared with SCEL [13], a formal language developed within the ASCENS EU project [33], the representation of knowledge in CARMA is more abstract, and not designed for detailed reasoning during the evolution of the model. This reflects the different objectives of the languages. Whilst SCEL was designed to support programming of autonomic computing systems, the primary focus of CARMA is quantitative analysis. In stochastic process algebras such as PEPA, MTIPP and EMPA, data is typically abstracted away, and the influence of data on behaviour is captured only stochastically. When data are important to differentiate behaviours, they must be implicitly encoded in the state of components. In the context of CAS we want to support attribute-based communication to reflect the flexible and dynamic interactions that occur in such systems; thus we cannot entirely abstract from data. For this reason CARMA offers a reasonable compromise between expressiveness and tractability.

Another key feature of CARMA is the inclusion of an explicit environment in which components interact. In PALOMA [14] there was a simple form of environment, called the *perception function*, but this proved to be cumbersome to use, and there was no way of modelling the influence of the different components on it. In CARMA, in contrast, the environment not only modulates the rates and the probabilities related to the interactions between components, but it can also evolve at runtime, by taking into account feedbacks from the collective.

The attributed pi calculus [22] is an extension of the pi calculus [26] that supports attribute-based communication, and was designed primarily with biological applications in mind. As with the languages discussed above, processes may have attributes and these are used to select partners for interaction, but communication is strictly synchronisation-based and binary. The language is equipped with both a deterministic and a Markovian semantics, and in the Markovian case the rates may depend on the values of the attributes involved. The possible attribute values are defined by a language $\mathscr{L}$, and the definition of the attributed pi calculus is parameterised by $\mathscr{L}$. The language $\mathscr{L}$ is also used to model the possible rates and the constraints that can be applied to attributes, offering thus the possibility to capture diverse behaviours within the framework when rates and probabilities of interaction are all dependent only on local behaviour and knowledge.

The rest of the report is organised as follows. In Section 2 we introduce the main features of CARMA and its syntax, while in Section A we describe its operational semantics. In Section 3 a number of tools developed to support CARMA specification and analysis are presented, together with a simple example showing the use of the proposed framework. Finally, in Section 4 we show how CARMA constructs can be used to model the space where CASs operate.

2 CARMA: Collective adaptive resource-sharing Markovian agents

CARMA is a new stochastic process algebra for the representation of systems developed according to the CAS paradigm [6]. The language offers a rich set of communication primitives, and permits exploiting attributes, captured in a store associated with each component, to enable attribute-based communication. For most CAS systems we anticipate that one of the attributes could be the location of the agent. Thus it is straightforward to model those systems in which, for example, there is a limited scope of communication or there is the restriction to only interact with components that are co-located, or where there is spatial heterogeneity in the behaviour of agents.

The rich set of communication primitives is one of the distinctive features of CARMA. Specifically, CARMA supports both unicast and broadcast communication, and permits locally synchronous, but globally asynchronous communication. This richness is important to take into account the spatially distributed nature of CAS, where agents may have only local awareness of the system, yet the design objectives and adaptation goals are often expressed in terms of global behaviour. Representing these patterns of communication in classical process algebras or traditional stochastic process algebras would be difficult, and would require the introduction of additional model components to represent buffers, queues and other communication structures.

Another key feature of CARMA is its distinct treatment of the *environment*. It should be stressed that although this is an entity explicitly introduced within our models, it is intended to represent something more pervasive and diffusive of the real system, which is abstracted within the modelling to be an entity which exercises influence and imposes constraints on the different agents in the system. For example, in a model of a smart transport system, the environment may have responsibility for determining the rate at which entities (buses, bikes, taxis etc) move through the city. However this should be recognised as an abstraction of the presence of other vehicles causing congestion which may impede the progress of the focus entities to a greater or lesser extent at different times of the day. The presence of an environment in the model does not imply the existence of centralised control in the system. The role of the environment is also related to the spatially distributed nature of CAS — we expect that the location *where* an agent is will have an effect on *what* an agent can do.

2.1 A gentle introduction to CARMA

To simplify the presentation, and to help the reader to appreciate the CARMA features, we will consider a simple running scenario; namely a *Smart Taxi System* used to coordinate the activities of a group of taxis in a city. In our scenario we assume that the city is subdivided in patches forming a grid. Two kinds of agents populate the system: *taxis* and *users*. Each taxi can either stay in a patch, waiting for user requests, or move to another patch. Users randomly arrive at different patches with a rate that depends on the specific time of day. After arrival, a user waits for a taxi and then moves to another patch.

The *smart taxi scenario* well represents typical scenarios that can be modelled with CARMA. Indeed, a CARMA system consists of a *collective* (N) operating in an *environment* ($\mathscr{E}$). The collective consists of a set of components and models the behavioural part of a system; it is used to describe a set of interacting *agents* that cooperate to achieve a given set of tasks. The environment models all those aspects which are intrinsic to the context where the agents under consideration are operating. The environment also mediates agent interactions.

Example 1. Smart Taxis - step 1/6
In our running example the collective N will be used to model the behaviour of *taxis* and *users*, while the environment will be used to model the city context where these agents operate. □

We let SYS be the set of CARMA *systems* S defined by the following syntax:

$$S ::= N \ \mathbf{in} \ \mathscr{E}$$

where N is a collective and $\mathscr{E}$ is an environment. The latter provides the global state of the system and governs the interactions in the collective.

We let COL be the set of collectives N which are generated by the following grammar:

$$N \ ::= \ C \ \big| \ N \parallel N$$

A collective N is either a *component* C or the parallel composition of two collectives ($N \parallel N$).

The precise syntax of components is:

$$C ::= \mathbf{0} \ \big| \ (P, \gamma)$$

where we let COMP be the set of components C generated by the previous grammar.

A component C can be either the *inactive component*, which is denoted by $\mathbf{0}$, or a term of the form (P, γ), where P is a *process* and γ is a *store*. A term (P, γ) models an *agent* operating in the system under consideration: the process P represents the agent's behaviour whereas the store γ models its *knowledge*. A store is a function which maps *attribute names* to *basic values*. We let:

- ATTR be the set of *attribute names* a, a', $a_1, \ldots$, b, b', $b_1, \ldots$;
- VAL be the set of *basic values* v, v', $v_1, \ldots$;
- Γ be the set of *stores* $\gamma, \gamma_1, \gamma', \ldots$, i.e. functions from ATTR to VAL.

Example 2. Smart Taxis - step 2/6.
To model our *smart taxi system* in CARMA we need two kinds of components, one for each of the two groups of agents involved in the system, i.e. *taxis* and *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.

The local store of components associated with taxis contains the following attributes:

- *loc*: identifies current taxi location;
- *occupied*: ranging in $\{0, 1\}$, describes if a taxi is free ($occupied = 0$) or engaged ($occupied = 1$);
- *dest*: if occupied, this attribute indicates the destination of a taxi journey.

Similarly, the local store of components associated with users contains the following attributes:

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

□

The behaviour of a component is specified via a process P. We let PROC be the set of processes P, Q,... defined by the following grammar:

$$\begin{array}{rcll|rcl}
P, Q & ::= & \mathbf{nil} & & act & ::= & \alpha^{\star}[\pi]\langle \overrightarrow{e} \rangle \sigma \\
& | & \mathbf{kill} & & & | & \alpha[\pi]\langle \overrightarrow{e} \rangle \sigma \\
& | & act.P & & & | & \alpha^{\star}[\pi](\overrightarrow{x})\sigma \\
& | & P + Q & & & | & \alpha[\pi](\overrightarrow{x})\sigma \\
& | & P \mid Q & & & & \\
& | & [\pi]P & & e & ::= & a \mid \mathsf{my}.a \mid x \mid v \mid \mathsf{now} \mid \cdots \\
& | & A & (A \triangleq P) & \pi & ::= & \top \mid \bot \mid e_1 \bowtie e_2 \mid \neg\pi \mid \pi \wedge \pi \mid \cdots
\end{array}$$

In CARMA processes can perform four types of actions: *broadcast output* ($\alpha^\star[\pi]\langle\overrightarrow{e}\rangle\sigma$), *broadcast input* ($\alpha^\star[\pi](\overrightarrow{x})\sigma$), *output* ($\alpha[\pi]\langle\overrightarrow{e}\rangle\sigma$), and *input* ($\alpha[\pi](\overrightarrow{x})\sigma$), where:

- α is an *action type* in the set of action type ACTTYPE;
- π is a *predicate*;
- x is a *variable* in the set of variables VAR;
- e is an expression in the set of expressions EXP[1];
- $\overrightarrow{\cdot}$ indicates a sequence of elements;
- σ is an *update*, i.e. a function from Γ to $Dist(\Gamma)$ in the set of *updates* Σ; where $Dist(\Gamma)$ is the set of probability distributions over Γ.

The admissible communication partners of each of these actions are identified by the predicate π. This is a predicate on *attribute names*. Note that, in a component (P,γ) the store γ regulates the behaviour of P. Primarily, γ is used to evaluate the predicate associated with an action in order to filter the possible synchronisations involving process P. In addition, γ is also used as one of the parameters for computing the actual rate of actions performed by P. The process P can change γ immediately after the execution of an action. This change is brought about by the *update* σ. The update is a function that when given a store γ returns a probability distribution over Γ which expresses the possible evolutions of the store after the action execution.

The *broadcast output* $\alpha^\star[\pi]\langle\overrightarrow{e}\rangle\sigma$ models the execution of an action α that spreads the values resulting from the evaluation of expressions $\overrightarrow{e}$ in the local store γ. This message can be potentially received by any process located at components whose store satisfies predicate π. This predicate may contain references to attribute names that have to be evaluated under the local store. These references are prefixed by the special name my. For instance, if loc is the attribute used to store the position of a component, action

$$\alpha^\star[\mathsf{distance}(\mathsf{my.loc},\mathsf{loc}) \le L]\langle\overrightarrow{v}\rangle\sigma$$

potentially involves all the components located at a distance that is less than or equal to a given threshold L. The *broadcast output* is non-blocking. The action is executed even if no process is able to receive the values which are sent. Immediately after the execution of an action, the update σ is used to compute the (possible) *effects* of the performed action on the store of the hosting component where the output is performed.

To receive a broadcast message, a process executes a *broadcast input* of the form $\alpha^\star[\pi](\overrightarrow{x})\sigma$. This action is used to receive a tuple of values $\overrightarrow{v}$ sent with an action α from a component whose store satisfies the predicate $\pi[\overrightarrow{v}/\overrightarrow{x}]$. The transmitted values can be part of the predicate π. For instance, $\alpha^\star[x > 5](x)\sigma$ can be used to receive a value that is greater than 5.

The other two kinds of action, namely *output* and *input*, are similar. However, differently from broadcasts described above, these actions realise a *point-to-point* interaction. The *output* operation is blocking, in contrast with the non-blocking broadcast output.

Choice and parallel composition are the usual process algebra operators. Processes can be guarded so that $[\pi]P$ behaves as the process P if the predicate π is satisfied. Finally, process **kill** is used to *destroy* a component. We assume that this term always occurs under the scope of an action prefix.

Example 3. Smart Taxis - step 3/6
We are now ready to describe the behaviour of *users* and *taxis*. The behaviour of a user is modelled via the process defined below:

$$W \stackrel{\triangle}{=} \mathsf{call}^\star[\top]\langle\mathsf{my.loc}\rangle.W + \mathsf{take}[\mathsf{loc} == \mathsf{my.loc}]\langle\mathsf{my.dest}\rangle.\mathbf{kill}$$

[1]The precise syntax of expressions e has been deliberately omitted. We only assume that expressions are built using the appropriate combinations of *values*, *attributes* (sometime prefixed with my), variables and the special term now. The latter is used to refer to current time unit.

This process can either *call* or *take* a taxi. To call a taxi W executes a *broadcast output* over call to all taxis. A *unicast output* over take is executed to take a taxi. This action is used to send user destination (my.dest) to a taxi that shares the same location as the user. To identify the target of this action, predicate (loc == my.loc) is used. The latter is satisfied only by those components that have attribute loc equal to my.loc. Here prefix my is used to refer to actual values of local attributes. After this action, the user disappears (he/she enters the taxi).

The behaviour of a taxi is described via processes F and G defined below:

$$F \stackrel{\triangle}{=} \mathsf{take}[\top](d)\{\mathsf{dest} \leftarrow d, \mathsf{occupied} \leftarrow 1\}.G + \mathsf{call}^{\star}[\mathsf{this.loc} \neq \mathsf{loc}](d)\{\mathsf{dest} \leftarrow d\}.G$$

$$G \stackrel{\triangle}{=} \mathsf{move}^{\star}[\bot]\langle\circ\rangle\{\mathsf{loc} \leftarrow \mathsf{dest}, \mathsf{dest} \leftarrow \bot, \mathsf{occupied} \leftarrow 0\}.F$$

A *taxi* component *executes* process F when it is *available* and it can either take a user in the current location or receive a *call* from another patch. In the first case an input via take is performed and the user destination is received. In the second case, the location where a user is waiting for a taxi is received. In both the cases, the received location is used, after action execution, to update taxi destination (dest $\leftarrow d$). However, after take input, the taxi also records that it is occupied (occupied $\leftarrow 1$).

When G is executed, a *taxi* just executes a *spontaneous* broadcast over action move. We refer to this action as *spontaneous* since predicate $\bot$ is used and no component can receive this message. Execution of action move models taxi movements. After the movement the taxi position is updated and the taxi is ready to take users in the new location.

To model the arrival of new users, the following process is used:

$$A \stackrel{\triangle}{=} \mathsf{arrival}^{\star}[\bot]\langle\circ\rangle\{\}.A$$

Process A only performs the spontaneous action arrival and it is executed in a separated component where attribute loc indicates the location where users arrive. The precise role of this process will be clear in a few paragraphs when the environment will be described. □

CARMA collectives operate in an environment $\mathscr{E}$. This environment is used to model the intrinsic rules that govern, for instance, the physical context where our system is situated. An environment consists of two elements: a *global store* γ_g, that models the overall state of the system, and an *evolution rule* ρ. The latter is a function which, depending on the *current time*, on the global store and on the current state of the collective (i.e., on the configurations of each component in the collective) returns a tuple of functions $\varepsilon = \langle \mu_p, \mu_r, \mu_u \rangle$ known as the *evaluation context* where $\textsc{Act} = \textsc{ActType} \cup \{\alpha^{\star} | \alpha \in \textsc{ActType}\}$ and:

- $\mu_p : \Gamma \times \Gamma \times \textsc{Act} \to [0,1]$, $\mu_p(\gamma_s, \gamma_r, \alpha)$ expresses the probability that a component with store γ_r can receive a message from a component with store γ_s when α is executed;
- $\mu_r : \Gamma \times \textsc{Act} \to \mathbb{R}_{\geq 0}$, $\mu_r(\gamma, \alpha)$ computes the execution rate of action α executed at a component with store γ;
- $\mu_u : \Gamma \times \textsc{Act} \to \Sigma \times \textsc{Col}$, $\mu_u(\gamma, \alpha)$ determines the updates on the environment (global store and collective) induced by the execution of action α at a component with store γ.

These functions regulate system behaviour and are determined by an *evolution rule* ρ depending on the current time, the global store and the actual state of the components in the system. For instance, the probability to receive a given message may depend on the *concentration* of components in a given state. Similarly, the actual rate of an action may be a function of the number of components whose store satisfies a given property.

Function μ_p, which takes as parameters the local stores of the two interacting components, i.e. the sender and the receiver, and the action used to interact, returns the probability to receive a message.

Example 4. Smart Taxis - step 4/6
In our *Taxi Scenario*, function μ_p can have the following form:

$$\mu_p(\gamma_s,\gamma_r,\alpha) = \begin{cases} \frac{1}{\#\{F@\gamma_s(\mathsf{loc})\}} & \alpha = \mathsf{take} \wedge \gamma_s(\mathsf{loc}) = \gamma_r(\mathsf{loc}) \\ 0 & \alpha = \mathsf{take} \wedge \gamma_s(\mathsf{loc}) \neq \gamma_r(\mathsf{loc}) \\ 1 - p_{lost} & \alpha = \mathsf{call}^\star \\ 1 & \text{otherwise} \end{cases}$$

where γ_s is the store of the sender, γ_r is the store of the receiver while we let $\#\{F@\gamma_s\}$ denote the number of available taxis at patch $\gamma_s(\mathsf{loc})$. The above function states when a user sends a request, each taxi in the same location receives the request with a probability that depends on the number of taxis in the patch while taxis that are in a different patch cannot receive the request. Moreover, a $\mathsf{call}^\star$ can be missed with a probability p_{lost}. All the other interactions occur with probability 1. □

Function μ_r computes the rate of a unicast/broadcast output. This function takes as parameter the local store of the component performing the action and the action on which interaction is based. Note that the environment can disable the execution of a given action. This happens when the function μ_r (resp. μ_p) returns the value 0.

Example 5. Smart Taxis - step 5/6
In our example μ_r can be defined as follow:

$$\mu_r(\gamma_s,\alpha) = \begin{cases} \lambda_r & \alpha = \mathsf{take} \\ \lambda_c & \alpha = \mathsf{call}^\star \\ mtime(\mathsf{now}, \mathsf{sender.loc}, \mathsf{sender.dest}) & \alpha = \mathsf{move}^\star \\ atime(\mathsf{now}, \mathsf{sender.loc}) & \alpha = \mathsf{arrival}^\star \\ 0 & \text{otherwise} \end{cases}$$

We say that actions take and $\mathsf{call}^\star$ are executed at a constant rate; the rate of a taxi movement is a function of actual time (now) and of starting location and final destination. Rate of user arrivals depends on current time now and on location loc. □

Finally, the function μ_u is used to update the global store and to install a new collective in the system. The function μ_u takes as parameters the store of the component performing the action together with the action type and returns a pair (σ, N). Within this pair, σ identifies the update on the global store whereas N is a new collective installed in the system. This function is particularly useful for modelling the arrival of new agents into a system.

Example 6. Smart Taxis - step 6/6
In our scenario function update is used to model the arrival of new users and it is defined as follows:

$$\mu_u(\gamma_s,\alpha) = \begin{cases} \{\}, (W, \{\mathsf{loc} = \gamma_s(\mathsf{loc}), \mathsf{dest} = destLoc(\mathsf{now}, \gamma_s(\mathsf{loc}))\}) & \alpha = \mathsf{arrival}^\star \\ \{\}, 0 & \text{otherwise} \end{cases}$$

When action $\mathsf{arrival}^\star$ is performed a component associated with a new user is created in the same location as the sender (see Example 3). The destination of the new user will be determined by function *destLoc* that takes current system time and starting location and returns a probability distribution on locations. □

2.2 The role of *environment* in CARMA

As we already stressed before, a key feature of CARMA is its distinctive treatment of the *environment* that is used to represent the effect of external entities on the single components. For example, in the Smart Taxi System modelled in the previous subsection, the environment determines the rate at which taxis may move through the city, an abstraction of the presence of other vehicles causing congestion which may impede the progress of the taxi to a greater to lesser extent at different times of the day.

This view of the environment coincides with the view taken by many researchers within the situated multi-agent community e.g. [31]. Specifically, in [32] Weyns *et al.* argue the importance of having a distinct environment within every multi-agent system. Whilst they are viewing such systems from the perspective of software engineers, many of their arguments are as valid when it comes to modelling a multi-agent or collective adaptive system. Thus our work can be viewed as broadly fitting within the same framework, albeit with a higher level of abstraction. Just as in the construction of a system, in the construction of a model distinguishing clearly between the responsibilities of the agents and of the environment provides separation of concerns and assists in the management of inevitably complex systems.

In [32] the authors provide the following definition: "*The environment is a first-class abstraction that proves the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources.*" This is the role that the environment plays within CARMA models through the evolution rules. However, in contrast to the framework of Weyns *et al.*, the environment in a CARMA model is not an active entity in the same sense as the agents are active entities. In our case, the environment is constrained to work *through* the agents, by influencing their dynamic behaviour or by inducing changes in the number and types of agents making up the system. Despite these differences, that are primarily arising due to the more abstract approach needed when modelling rather than implementing a CAS, we consider the CARMA conceptual model to be in line with the Weyns *et al.* framework. It is also noteworthy that in CARMA the environment is formally specified and integrated into the operational semantics of the language.

In [28], Saunier *et al.* advocate the use of an *active environment* to mediate the interactions between agents; such an active environment is aware of the current context for each agent. The environment in CARMA also supports this view, as the evolution rules in the environment take into account the state of all the potentially participating components to determine both the rate and the probability of communications being successful, thus achieving a multicast communication not based on the address of the receiving agents, as suggested by Saunier *et al.* This is what we term "attribute-based communication" in CARMA. Moreover, when the application calls for a centralised information portal, the global store in CARMA can represent it. The higher level of abstraction offered by CARMA means that many implementation issues are ignored. However, the CARMA environment could be viewed as capturing the EASI (Environment for Active Support of Interaction) environment of Saunier *et al.* [28], although in CARMA the *filter* is more closely associated with the actions. However, just as in EASI, filters (predicates) may be specified separately by the sender, the receiver and the environment. Our predicates are, however, more strict and "overhearing" type interactions must be anticipated by the modeller, since the effect is taken to be the conjunction of the sender, receiver and environment predicates, thus removing the need for policies to arbitrate between conflicting filters.

The role of the environment is related to the spatially distributed nature of CAS — we expect that *where* an agent is will have an effect on *what* an agent can do. Thus we do find similar features in modelling languages targeted at other domains where locations do influence the possible behaviour of agents. For example, several formalisms developed in the context of biological processes, especially intracellular processes, capture the spatial arrangement of elements in the system because this can have a profound effect on the behaviours that can be observed. In this context, the most important aspect of the spatial arrangement is often hierarchical and logical, and is not concerned with the actual physical placement of elements; this is sometimes termed *multi-level* modelling. Moreover the concept of levels may also refer to organisational levels, as well as physical levels so that the relationship between levels might be characterised as *consists of* or *contains* [5]. Here we are particularly concerned with modelling techniques that aim to faithfully represent the stochastic dynamic behaviour of systems, allowing properties such as performance, availability and dependability to be assessed.

One example of such a language is the ML-Rules formalism developed by Maus *et al* [25]. Here the focus is on hierarchical nesting of biological entities and the underlying semantics is given in terms of a population CTMC, intended for analysis by simulation. Entities can be created "on demand" but this must be programmed by the modeller within the underlying simulation engine, and is not supported at the level of the modelling language itself. Rules are used to define the possible reactions in the system and the state updates which result from them; rules are applied by pattern-matching. As in CARMA, agents are equipped with attributes and these may be used to filter the rules which may be applied, although there is no explicit naming of attributes making it difficult for other agents to access current values. Moreover it is the modeller's responsibility to ensure that attributes are used consistently across different rules. Explicit function calls are performed by agents to determine execution parameters such as Markovian rates or probabilities, thus assuming that agents have direct access to a global knowledge. In contrast, in CARMA it is assumed that access to global knowledge is restricted to the environment. Allowing agents to directly access global knowledge makes it more difficult to consider the same set of agents in a different context, because there is not a clear separation between agents and their environment. The multi-level aspect of ML-Rules is used to capture a form of vertical causation, where the application of a rule at one level triggers an update within another level of the model.

The upward and downward causation are also key features of the ML-DEVS formalism, presented in [29], although in a slightly more restricted form since here agents can only trigger changes in the levels immediately above or below them, whereas in ML-Rules changes impact across arbitrary levels in the hierarchy. ML-DEVS is a modular, hierarchical formalism which is intended to represent a reactive system which interacts asynchronously with its “environment". However the formalism does not support the notion of a distinguished environment, as in CARMA, but rather considers the environment of an agent to consists of the agents at the same level (with which it may form horizontal couplings) and those in the adjacent levels (with which it may form vertical couplings).

BioSpace [9] is a novel process calculus designed for modelling the physical arrangement of biological molecules in applications such as the formation of polymers and the interactions between microbes and biomaterials [10]. Individual agents in the calculus represent the biological entities, but operate in a type environment against which the legitimacy of their actions can be checked: all involved actions and entities have a type, and type consistency is checked before the model evolves. BioSpace^L extends BioSpace by allowing explicit placement of entities, and giving the modeller the power to program location updates. This reflects the key role that location plays in the considered biological applications, but the physical environment is represented rather implicitly.

In the formalism presented in [5], a multi-level approach is introduced which is based on organisational rather than spatial structures. Each level consists of a number of agents whose behaviour may depend on agents at a lower level. In this *system of systems*, agents are represented by automata and automata are organised in tree-like structures in such a way that agents at one level are constituted from their child automata. For example, in a biological setting, the top level might be *tissue* which may alternate between healthy and diseased states; this may be made up by *cells* and the state of the cells within the tissue will influence its health or otherwise; the behaviour and state of cells will depend on the biochemical networks within them, themselves made up of proteins in various states of abundance. Again there are notions of horizontal and vertical couplings, and agents higher in the tree provide the environment to those that are below, in a hierarchical manner.

Similarly the ambient calculus [7], and its biological dialect, bio-ambients [27], capture the behaviour of elements within a system, with respect to a hierarchical arrangement of physical or logical space. As elements move into or out of domains, their behaviour may change because they change their context of operation and communication is limited to be local.

In contrast to these multi-level models, in CARMA we restrict to two levels. The behaviour of the entities within the system are captured by the collective, and this is placed in the context of an environment, which is distinct from any entity and which has the power to constrain the behaviour of the entities through the evolution rule. This reflects our treatment of location in terms of physical space, rather than the hierarchical arrangement of space commonly used in biological modelling where the emphasis is on the containment relationship. It is worth noting that BioSpace^L, which has similar focus of the physical rather than logical representation of space, also takes a two level approach with the entities and the environment. Within the collective there is no hierarchy, although a single component may have behaviour resulting from the composition of multiple process automata.

A two level approach is also found in quantitative formalisms such as PEPA nets [19], Spatial PEPA [15] and STOKLAIM [12]. These languages were motivated by mobile computing and therefore share CAS's aim of capturing systems with behaviour distributed over physical space, where current location or position of an entity, and in particular co-location, may influence the actions that can be undertaken. In PEPA nets and Spatial PEPA a graphical representation of the physical locations is used either as a Petri net, or as a hyper-graph, and the physical structure is taken to be static. In STOKLAIM, in contrast, the processes within the system may explicitly control the physical structure of the global network. Despite some success in modelling mobile computing scenarios of the time, these languages are not equipped to represent large populations of entities with similar behaviour, thus they are not well-suited to capture the collective nature of CAS. This large scale nature of CAS systems makes it essential to support scalable analysis techniques, thus CARMA has been designed anticipating both a discrete and a continuous semantics in the style of [30].

2.3 CARMA semantics

Operational semantics of CARMA specifications is defined in three stages:

1. the transition relation $\xrightharpoonup{\cdot}_{\cdot,\cdot}$ describes the behaviour of a single component.
2. the transition relation $\xrightarrow{\cdot}_{\cdot,\cdot}$ builds on $\xrightharpoonup{\cdot}_{\cdot,\cdot}$ to describe the behaviour of collectives.
3. the transition relation $\xmapsto{\cdot}$ describes how CARMA systems evolve.

Formal details of these three relations can be found in Appendix A.

All relations are defined in the FuTS style [11]. Using this approach, a transition relation is described using a triple of the form $(N, \ell, \mathscr{N})$. The first element of this triple is either a component, or a collective, or a system. The second element is a transition label. The third element is a function associating each component, collective, or system with a non-negative number. A non-zero value represents the rate of the exponential distribution characterising the time needed for the execution of the action represented by ℓ. The zero value is associated with unreachable terms. We use the FuTS style semantics because it makes explicit an underlying (time-inhomogeneous) Action Labelled Markov Chain, which can be simulated with standard algorithms [18] but is nevertheless more compact than Plotkin-style semantics, as the functional form allows different possible outcomes to be treated within a single rule. A complete description of FuTS and their use can be found in [11].

3 CARMA implementation

To support simulation of CARMA models, a prototype simulator has been developed. This simulator, which has been implemented in Java, can be used to perform stochastic simulation and will be the basis for the implementation of other analysis techniques. An Eclipse plug-in for supporting specification and analysis of CAS in CARMA has also been developed. In this plug-in, CARMA systems are specified by using an appropriate high-level language for designers of CAS, named the CARMA *Specification Language*. This is mapped to the process algebra, and hence will enable qualitative and quantitive analysis of CAS during system development by enabling a design workflow and analysis pathway. The intention of this high-level language is not to add to the expressiveness of CARMA, which we believe to be well-suited to capturing the behaviour of CAS, but rather to ease the task of modelling for users who are unfamiliar with process algebra and similar formal notations. Both the simulator and the Eclipse plug-in are available at `https://quanticol.sourceforge.net/`.

In the rest of this section, we first describe the CARMA *Specification Language*, then we show how the *Smart Taxi Scenario* considered in Section 2 can be modelled, simulated and analysed in the provided plug-in. An overview of the CARMA Eclipse Plug-in, together with an example of use, is provided in Deliverable D5.2.

3.1 CARMA Specification Language

In this section we present the language that supports design of CAS in CARMA. To describe the main features of this language, following the same approach used in Section 2, we will use the *Smart Taxi Scenario*.

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

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

Data types. Three basic types are natively supported in our specification language. These are: `bool`, for booleans, `int`, for integers, and `real`, for real values. However, to model complex structures, like for instance the one introduced in Section 4, it is often useful to introduce custom types. In a CARMA specification two kind of custom types can be declared: *enumerations* and *records*.

Like in many other programming languages, an *enumeration* is a data type consisting of a set of *named values*. The enumerator names are identifiers that behave as constants in the language. An attribute (or variable) that has been declared as having an enumerated type can be assigned any of the enumerators as value. In other words, an enumerated type has values that are different from each other, and that can be compared and assigned, but which are not specified by the programmer as having any particular concrete representation. The syntax to declare a new *enumeration* is:

enum *name* = $elem_1, \ldots, elem_n$;

where *name* is the name of the declared enumeration while $elem_i$ are its value names. Enumeration names start with a capitalised letter while the enumeration values are composed by only capitalised letters.

Example 7. Enumerations can be used to define predefined set of values that can be used in the specification. For instance one can introduce an enumeration to identify the possible four directions of movement:

```
enum Direction = NORTH, SOUTH, EAST, WEST;
```

To declare aggregated data structures, a CAS designer can use *records*. A record consists of a sequence of a set of typed fields:

record *name* = [$type_1$ $field_1, \ldots,$ $type_n$ $field_n$];

Each field has a type $type_i$ and a name $field_i$: $type_i$ can be either a built-in type or one of the new declared type in the specification; $field_i$ can be any valid identifier.

Example 8. In the Smart Taxi scenario considered in Section 2, we can use a record to model the position of users and taxis:

```
record Position = [ int x, int y];
```

A record can be created by assigning a value to each field, within square brackets:

[$field_1$=$expression_1, \ldots,$ $field_n$=$expression_n$]

Example 9. In the Smart Taxi scenario, the instantiation of a location referring to the patch located at $(0,0)$ has the following form:

```
[ x=0 , y=0 ]
```

Given a variable (or attribute) having a record type, each field can be accessed using the *dot* notation:

variable . $field_i$

Constants and Functions. A CARMA specification can also contain *constants* and *functions* declarations having the following syntax:

const *name* = *expression*;

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

where the body of an expression consists of standard statements in a high-level programming language. The type of a constant is not declared but inferred directly from the assigned expression.

Example 10. A constant can be used to represent the *size* of the grid:

```
const SIZE = 3;
```

Moreover, functions can be used to perform complex computations that cannot be done in a single expression:

```
fun Position DestLoc(real time, Position g){
    Position q := [ x := 0 , y := 0 ];
    if( g.x == 1 && g.y == 1){
      q := Roving();
    } else {
      q := [ x := 1 , y := 1 ];
    }
    return q;
}
```

Function DestLoc defined above is used to compute the destination of a user located at position g. If g is the central patch, i.e. the one with coordinates $(1,1)$, one of the patches in the border is selected (computed by function Roving that is not reported here); otherwise, g is a patch on the border and the position $(1,1)$ is returned.

Components prototype. A *component prototype* provides the general structure of a component that can be later instantiated in a CARMA system. Each prototype is parameterised with a set of typed parameters and defines: the store; the component's behaviour and the initial configuration. The syntax of a *component prototype* is:

```
component name( type_1 arg_1 ,..., type_n arg_n ) {
    store { ···
        attr_kind aname_i := expression_i ; ···
    }
    behaviour { ···
        proc_i = pdef_i ; ···
    }
    init { P_1 |···| P_w }
}
```

Each component prototype has a possibly empty list of arguments. Each argument arg_i has a type $type_i$ that can be one of the built-in types (`bool`, `int` and `real`), a custom type (an enumeration or record), or the type `process` that indicates a component behaviour. These arguments can be used in the body of the component. The latter consists of three (optional) blocks: `store`, `behaviour` and `init`.

The block `store` defines the list of attributes (and their initial values) exposed by a component. Each attribute definition consists of an attribute kind *attr_kind* (that can be either `attrib` or `const`), a *name* and an expression identifying the initial attribute value. When an attribute is declared as `const`, it cannot be changed. The actual type of an attribute is not declared but inferred from the expression providing its initialisation value.

The block `behaviour` is used to define the processes that are specific to the considered components and consists of a sequence of definitions of the form

$$proc_i = pdef_i ;$$

where $proc_i$ is the process name while $pdef_i$ is its definition having the following syntax[2]:

[2]All the operators are right associative and presented in the order of priority.

$$
\begin{array}{lcl}
\mathit{pdef} & ::= & \mathit{pdef}+\mathit{pdef} \\
& | & \texttt{[}\ \mathit{expr}\ \texttt{]}\ \mathit{pdef} \\
& | & \mathit{act\_name}\texttt{[}\ \mathit{expr}\ \texttt{]<}\ \mathit{expr}_1,\ldots,\ \mathit{expr}_n\texttt{>\{}\ \mathit{aname}_1\ \texttt{:=}\ \mathit{expr}'_1,\ldots,\mathit{aname}_k\ \texttt{:=}\ \mathit{expr}'_k\ \ldots\ \texttt{\}.}\mathit{proc} \\
& | & \mathit{act\_name}\texttt{*[}\ \mathit{expr}\ \texttt{]<}\ \mathit{expr}_1,\ldots,\ \mathit{expr}_n\texttt{>\{}\ \mathit{aname}_1\ \texttt{:=}\ \mathit{expr}'_1,\ldots,\mathit{aname}_k\ \texttt{:=}\ \mathit{expr}'_k\ \ldots\ \texttt{\}.}\mathit{proc} \\
& | & \mathit{act\_name}\texttt{[}\ \mathit{expr}\ \texttt{](}\ \mathit{var}_1,\ldots,\ \mathit{var}_n\texttt{)\{}\ \mathit{aname}_1\ \texttt{:=}\ \mathit{expr}'_1,\ldots,\mathit{aname}_k\ \texttt{:=}\ \mathit{expr}'_k\ \ldots\ \texttt{\}.}\mathit{proc} \\
& | & \mathit{act\_name}\texttt{*[}\ \mathit{expr}\ \texttt{](}\ \mathit{var}_1,\ldots,\ \mathit{var}_n\texttt{)\{}\ \mathit{aname}_1\ \texttt{:=}\ \mathit{expr}'_1,\ldots,\mathit{aname}_k\ \texttt{:=}\ \mathit{expr}'_k\ \ldots\ \texttt{\}.}\mathit{proc}
\end{array}
$$

Finally, block `init` is used to specify the initial behaviour of a component. It consists of a sequence of terms P_i separated by the symbol `|`. Each P_i can be a process defined in the block `behaviour`, `kill` or `nil`.

Example 11. The prototypes for User, Taxi and Arrival components, already described in Example 2, can be defined as follow:

```
component User(Position loc, Position dest){
  store{
    attrib loc := loc;
    attrib dest := dest;
  }
  behaviour{
    Wait =
      call*[true]<loc>.Wait
      +
      take[loc == my.loc]<my.dest>.kill;
    }
  init{ Wait }
}

component Taxi( Position loc ){
  store{
    attrib loc := loc;
    attrib dest := [ x:=-1 , y:=-1];
    attrib occupancy := false;
  }
  behaviour {
    F =
      take[true]( pos ){dest := pos, occupancy := true}.G
      +
      call*[my.loc != loc](pos){dest := [ pos ] }.G;
    G =
      move*[false]<>{loc := dest, dest := [x:=-1,y:=-1], occupancy := false}.F;
  }
  init{ F }
}

component Arrival(int a, int b){
  store{
    attrib loc := [ x:=a , y:= b];
  }
  behaviour{
    A = arrival*[false]<>.A;
  }
  init { A }
}
```

System definitions. A system definition consists of two blocks, `collective` and `environment`, that are used to declare the collective in the system and its environment, respectively:

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

Above, *inist_stmt* indicates a sequence of commands that are used to instantiate components. The basic command to create a new component is:

```
new name( expr1 ,... ,exprn )
```

where *name* is the name of a component prototype. However, in a system a large number of collectives can occur. For this reason, our specification language provides specific constructs for the instantiation of multiple copies of a component. A first construct is the *range operator*. This operator is of the form:

```
[ expr1 : expr2 : expr3 ]
```

and can be used as an argument of type integer. It is equivalent to a sequence of integer values starting from $expr_1$, ending at $expr_2$. The element $expr_3$ (that is optional) indicates the step between two elements in the sequence. When $expr_3$ is omitted, value 1 is assumed. The *range operator* can be used where an integer parameter is expected. This is equivalent to having multiple copies of the same instantiation command where each element in the sequence replaces the command.

For instance, assuming `SIZE` to be the constant identifying the size of the grid of the city, the instantiation of the taxi components can be modelled as:

```
new Taxi(0:SIZE−1,0:SIZE−1);
```

The command above is equivalent to:

```
new Taxi(0,0);
        ⋮
new Taxi(SIZE−1,0);
new Taxi(0,1);
        ⋮
new Taxi(SIZE−1,SIZE−1);
```

Two other commands are used to control components instantiation. These are:

```
for ( var_name = expr1 ; expr2 ; expr3 ) {
    inist_stmt
}

if ( expr ) {
    inist_stmt
} else {
    inist_stmt
}
```

The former is used to iterate an instantiation block for a given number of times while the latter can be used to differentiate the instantiation depending on a given condition.

Syntax of an block `environment` is the following:

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

The block `store` defines the *global store* and has the same syntax as the similar block already considered in the component prototypes. Block `prob` is used to compute the probability to receive a message. Syntax of `prob` is the following:

```
prob { ···
    [guard_i] act_i : expr_i; ···
    default : expr;
}
```

In the above, each $guard_i$ is a boolean expression over the stores of the two interacting components, i.e. the sender and the receiver, and while act_i denotes the action used to interact. In $guard_i$ attributes of sender and receiver are referred to using `sender.a` and `receiver.a`, while the values published in the global store are referenced by using `global.a`. This probability value may depend on the number of components in a given state. To compute this value, expressions of the following form can be used:

```
#{ Π | expr }
```

This expression denotes the number of components in the system satisfying boolean expression *expr* where a process of the form Π is executed. In turn, Π is a pattern of the following form:

$$\Pi \;::=\; * \;\Big|\; *[\ proc\ ] \;\Big|\; comp[\ *\ ] \;\Big|\; comp[\ proc\ ]$$

Example 12. One can use `#{ Taxi[F] | my.loc == sender.loc }` to count the number of available taxis at patch `sender.loc`. This expression can be used as follows:

```
prob{
  [true] take : Takeprob(real(#{Taxi[F] | my.loc == sender.loc }));
  [true] call* : 1-P_LOST;
  default : 1.0;
}
```

Above, we say that each taxi receives a user request with a probability that depends on the number of taxis in the patch. Moreover, call* can be missed with a probability p_{lost}. All the other interactions occur with probability 1.

Block `rate` is similar and it is used to compute the rate of an unicast/broadcast output. This represents a function taking as parameter the local store of the component performing the action and the action type used. Note that the environment can disable the execution of a given action. This happens when evaluation of block `rate` (resp. `prob`) is 0. Syntax of `rate` is the following:

```
rate { ···
    [guard_i] act_i : expr_i; ···
    default : expr;
}
```

Differently from `prob`, in `rate` guards $guard_i$ are evaluated by considering only the attributes defined in the store of the component performing the action, referenced as `sender.a`, or in the global store, accessed via `global.a`.

Example 13. In our example `rate` can be defined as follow:

```
rate{
  [true] take : R_T;
  [true] call* : R_C;
  [true] move* : Mrate(now,sender.loc,sender.dest);
  [true] arrival* : Arate(now,sender.loc);
  default : 0.0;
}
```

We say that actions `take` and `call*` are executed at a constant rate; the rate of a taxi movement is a function of actual time (`now`) and of starting location and final destination. Rate of user arrivals depends on current time `now` and on location `loc`.

Finally, the block `update` is used to update the global store and to install a new collective in the system. Syntax of `update` is:

```
update { ···
    [guard_i] act_i : attr_updt_i ; inst_cmd_i ; ···
}
```

As for `rate`, guards in the `update` block are evaluated on the store of the component performing the action and on the global store. However, the result is a sequence of attribute assignments followed by an instantiation command (above considered in the collective instatiation). If none of the guards are satisfied, or the performed action is not listed, the global store is not changed and no new collective is instantiated. In both cases, the collective generating the transition remains in operation. This function is particularly useful for modelling the arrival of new agents into a system.

Example 14. In our scenario block `update` is used to model the arrival of new users and it is defined as follows:

```
update {
  [true] arrival* : new User(sender.loc,DestLoc(now,sender.loc), Wait);
}
```

When action `arrival*` is performed a component associated with a new user is created in the same location as the sender (see Example 3). The destination of the new user will be determined by function `DestLoc` that takes current system time and starting location and returns a probability distribution on locations.

Measure definitions. To extract observations from a model, a CARMA specification also contains a set of *measures*. Each measure is defined as:

```
measure m_name[ var_1=range_1 , ..., var_n=range_n ] = expr;
```

Expression *expr* can be used to count, by using expressions of the form `#{ Π | expr }` already described above, or to compute statistics about attribute values of components operating in the system: `min{ expr | guard }`, `max{ expr | guard }` and `avg{ expr | guard }`. These expressions are used to compute the minimum/maximum/average value of expression *expr* evaluated in the store of all the components satisfying boolean expression *guard*, respectively.

Example 15. In our scenario, we could be interested in measuring the number of waiting users at a given location. These measures can be declared as:

```
measure WaitingUser[ i := 0:SIZE-1 , j := 0:SIZE-1 ]
                   = #{User[Wait] | my.loc.x == i && my.loc.y == j };
```

3.2 The Smart Taxi System: Simulation and Analysis

In this section we present the Smart Taxi System in its entirety and demonstrate the quantitative analysis which can be undertaken on a CARMA model. One of the main advantages of the fact that we structure a CARMA system specification in two parts – a collective and an environment – is that we can evaluate the same collective in different enclosing environments.

We now consider a scenario with a grid of 3×3 patches, a set of locations (i, j) where $0 \leq i, j \leq 2$, and instantiate the environment of the *smart taxi system* with respect to two different specifications for the environment:

Scenario 1: Users arrive in all the patches at the same rate;

Scenario 2: At the beginning users arrive with a higher probability to the patches at the border of the grid; subsequently, users arrive with higher probability in the centre of the grid.

In both the scenarios users in the border will use the taxi to go to the *centre*, while users from the centre will use the taxi to go to any other location (the destination is probabilistically selected). In both scenarios, we assume that the movement rate is constant and is proportional to the number of patches to be traversed to reach the destination, and collectives have the following structure:

```
    collective{
      for( i ; i<K ; i+1 ) {
          new Taxi(0:SIZE,0:SIZE,3,3,0,F);
        }
  new Arrival(0:SIZE,0:SIZE);
    }
```

Above we consider `K=5` taxis in each location and `SIZE=3`. The environment for **Scenario 1** is:

```
environment{

  prob{
    [true] take :
      Takeprob( real( #{Taxi[F] | my.loc == sender.loc } ) );
      [true] call* : 1.0 - P_LOST;
      default : 1.0;
  }

        rate{
              [true] take : R_T;
      [true] call* : R_C;
      [true] move* : Mrate(sender.loc,sender.dest);
      [true] arrival* : A_RATE*(1.0/real(SIZE*SIZE));
      default : 0.0;
        }

        update{
            [true] arrival* : new User(sender.loc ,DestLoc(now,sender.loc), Wait);
        }

    }
```

where function `DestLoc` is the one defined in Example 10 while functions `Mrate` and `Takeprob` are defined below:

```
fun real Mrate(Position l1, Position l2){
  real t := real( abs(l1.x - l2.x) + abs(l1.y - l2.y) );
  real r := 0.0;
  if(t > 1.0){
    r := R_STEP / t;
  } else{
    r := R_STEP;
  }
  return r;
}

fun real Takeprob( int taxisAtLoc ){
  real x_:= 0.0;
  if (taxisAtLoc == 0){
    x_ := 0.0;
  }
  else{
    x_ := 1.0/real(taxisAtLoc);
  }
  return x_;
}
```

In the above `R_STEP` is the rate of the movements from one location to an adjacent one while `A_RATE` is the arrival rate of users in the system. The complete CARMA specification of our system can be found in Appendix B. The results of the simulation of the CARMA model are instead reported in Figure 1. On the left we can observe the average number of users that are waiting for a taxi in the location $(1,1)$ and in one location in the border of the grid, namely $(0,0)$[3]. On the right is the proportion of free taxis that are waiting for a user at location $(1,1)$ and

[3]Due to the symmetry of the considered model, any other location in the border presents similar results.

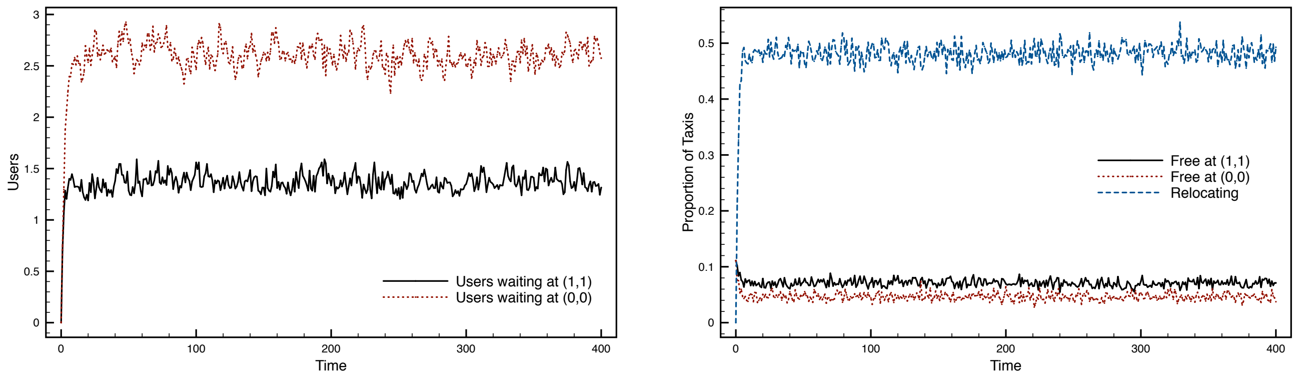


Figure 1: Smart Taxi System: Scenario 1 — *single simulation run*

$(0,0)$, respectively, and the fraction of taxis that are moving from one patch to another without a user (these are the taxis that are relocating after a *call* has been received). The remaining taxis (not shown) are engaged by users.

We can notice that, on average and after an initial startup period, around 2.5 users are waiting for a taxi in the location in the periphery of the grid while only 1.5 users are waiting for a taxi in location $(1,1)$. This is due to the fact that in **Scenario 1** a larger fraction of users are delivered to location $(1,1)$, that is the central patch. For this reason, a larger fraction of taxis will soon be available to collect users at the centre whereas to collect a user from the border, a taxi has to change its location. This aspect is also witnessed by the fact that, in this scenario, a large fraction of taxis (around 50%) are continually moving between the different patches.

The simulation of **Scenario 2** is reported in Figure 2. The environment for this scenario is exactly the same as considered for the previous one except for the computation of user arrival rate. This is computed via function *Arate* that takes into account the current time (parameter now) to model the fact that the arrival of new users depends on current time, just as we might expect traffic patterns within a city to vary according to the time of day. We assume that from time 0 to time 200, $3/4$ of users *arrive* on the border while only $1/4$ request a taxi in the city centre. After time 200 these values are switched. Environment for **Scenario 2** is the following:

```
environment {
  prob{
    [true] take :  Takeprob(  real( #{Taxi[F] | my.loc.x == sender.loc } )  );
    [true] call* : 1.0 − P_LOST;
    default : 1.0;
  }
  rate {
    [true] take : R_T;
    [true] call* : R_C;
    [true] move* : Mrate(sender.loc,sender.dest);
    [true] arrival* : Arate(now,sender.loc);
    default : 0.0;
  }
  update {
    [true] arrival* : new User(sender.loc,DestLoc(now,sender.loc), Wait);
  }
}
```

while definition of function *Arate* is the following:

```
fun real Arate(real time, Position l1){
  real r := 0.0;
  if ((l1.x == 1)&&(l1.y==1)) {
    if(time < 20)  r := R_A / 4.0;
    else r := 3.0 * R_A / 4.0;
  } else {
```

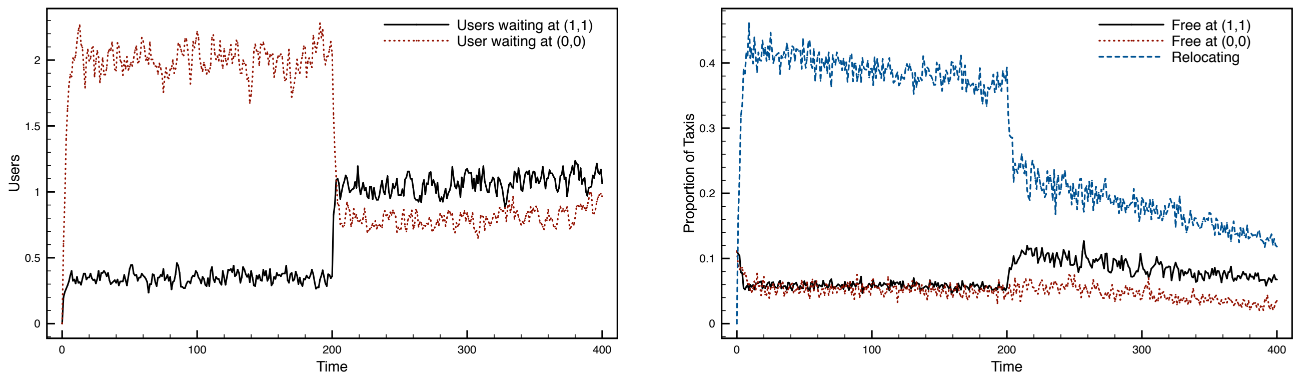


Figure 2: Smart Taxi System: Scenario 2 — *single simulation run*

```
        if(time < 200) r := 3.0 * R_A / 4.0;
        else r := R_A / 4.0;
    }
    return r;
}
```

We can notice that the results obtained from time 0 to time 200 are similar to the ones already presented for the first scenario. However, after time 200, as expected, the number of users waiting for a taxi in the border decreases below 1 whilst the average waiting for a taxi in the centre increases to just over 1. Since after time 200 a large proportion of users request a taxi in the centre, the fraction of taxis that change their location without a user decreases from 40% to 20%.

In both the scenarios one can observe that even if only a small number of users are waiting for a taxi, a significant fraction of taxis are continually moving from one patch to another without users (i.e. in a free state). This is mainly due to the fact that the action used to *call* a taxi is a broadcast output. As a consequence we have that even if only a single user needs a taxi at a given location, all the *free* taxis can change their position to satisfy this request. To study this aspect in more detail, we consider now a variant of components *Taxi* and *User* where action *call* is no longer a broadcast output, but it is instead a unicast output. The CARMA representation of the variants of these two components is reported below:

```
component User(Position loc, Position dest){
    store{
        attrib loc := loc;
        attrib dest := dest;
    }
    behaviour{
        Wait =
            call[true]<loc>.Wait
            +
            take[loc == my.loc]<my.dest>.kill;
        }
    init{ Wait }
}
component Taxi( Position loc ){
    store{
        attrib loc := loc;
        attrib dest := [ x:=-1 , y:=-1];
        attrib occupancy := false;
    }
    behaviour {
        F =
            take[true]( pos ){dest := pos, occupancy := true}.G
            +
            call[my.loc != loc](pos) {dest := [ pos ] }.G;
```

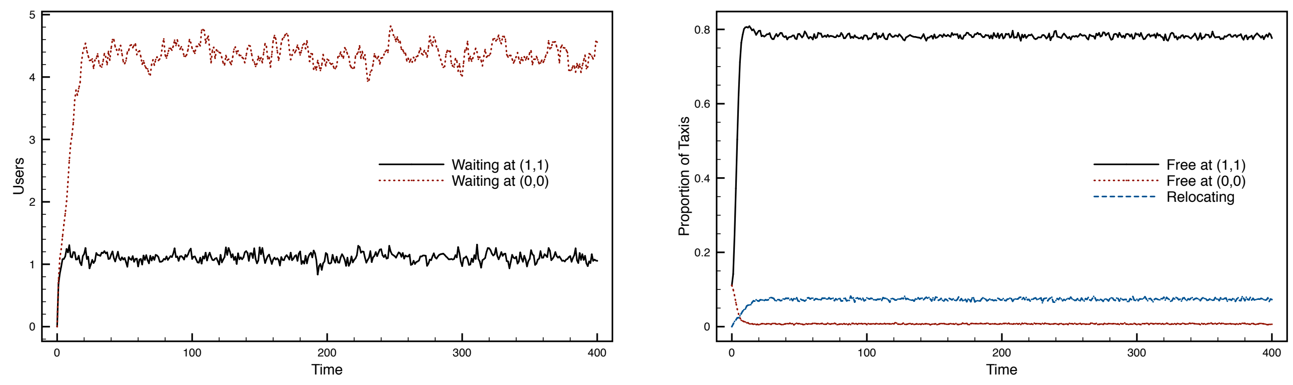


Figure 3: Smart Taxi System: Scenario 1 (modified specification) — *single simulation run*

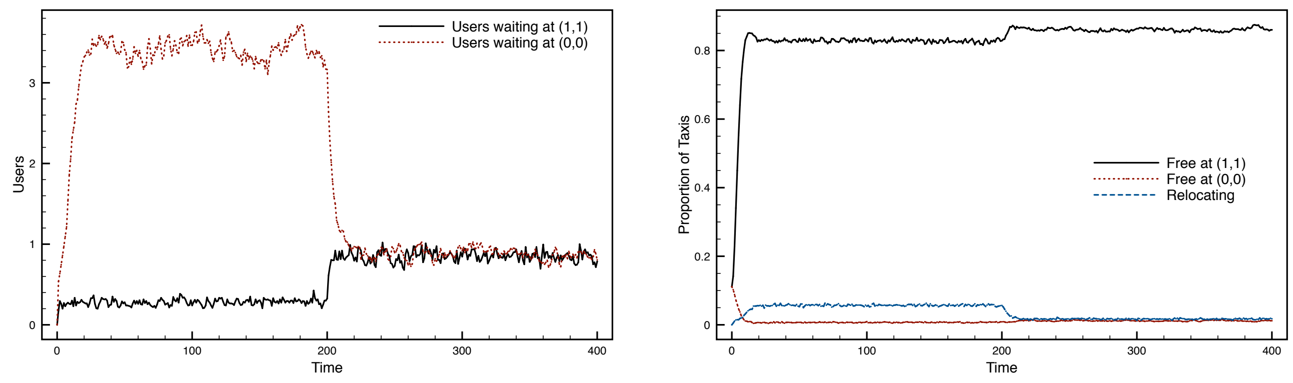


Figure 4: Smart Taxi System: Scenario 2 (modified specification) — *single simulation run*

```
        G =
          move*[false]<>{loc := dest, dest := [x:=-1,y:=-1], occupancy := false}.F;
      }
      init{ F }
    }
```

The results of simulating the two scenarios with the modified specifications are reported in Figure 3 and Figure 4. In both cases we can observe that the number of users waiting for a taxi in patches located in the border of the grid doubles, whilst almost all taxis will wait for users' calls in the centre location (around 80%). This means that after an initial startup period all the taxis will be always staying in the central location and the patch arrangement of the city is, in fact, no longer used in the model.

4 Towards integrating space in CARMA

In this section we briefly discuss a proposal for representing spatial information in CARMA component attributes. As previously discussed in [16], *Closure Spaces* can be used to provide a unifying framework for modelling and reasoning about space at a conceptual/theoretical level. They can be represented as Abstract Data Types (ADTs) and can be incorporated in programming/modelling languages to provide users with a controlled way for defining/using specific spaces and space inter-relationships (e.g. via suitable packages).

Additional information about space representation in QUANTICOL can be found in [16, 24]. More details on the extension of CARMA proposed here are presented in [8], where an example of a detailed specification of an adaptive bike-sharing system is also provided, using the extension.

In the next section, we recall the main notions concerning Closure Spaces, then, in Sect. 4.2, we show, by means of small examples, how they can be embedded in CARMA as Abstract Data Types (ADTs) and how the definition of ADTs as combination of different space ADTs can be useful in CARMA models. This is quite important when dealing with models of complex systems made of components representing objects of different nature with different interaction features, which may imply *different views* of space within the same model.

4.1 Closure Spaces as a Foundational Framework for Space

At a foundational level, *Closure Spaces* can be used as an underlying unifying framework for modelling and reasoning about space [17]. A closure space is a pair $(X,\mathscr{C})$, where X is a set (of points) and $\mathscr{C} : \mathscr{P}(X) \to \mathscr{P}(X)$ is a *closure operator*, i.e. a total function such that, for all $A,B \subseteq X$ the following three *Closure Axioms* hold: $\mathscr{C}(\emptyset) = \emptyset$; $A \subseteq \mathscr{C}(A)$; $\mathscr{C}(A \cup B) = \mathscr{C}(A) \cup \mathscr{C}(B)$.

The importance of Closure Spaces at the foundational level stems mainly from the fact that they enjoy basic properties which are essential for supporting rigorous reasoning about space, e.g.:

Generality: Most common mathematical models of space, including Topological Spaces –thus Euclidean Spaces– Distance Spaces, Pseudo-metric and Metric Spaces are special cases of Closure Spaces; but also discrete spaces like, e.g. Patches or Graphs, and their labelled variants, are Closure Spaces;

Minimality: In their general form, Closure Spaces are characterised by just *one* operation, namely a closure operator;

Simplicity: A closure operator is one satisfying the three simple and intuitive axioms recalled above.

In the following we shall briefly discuss the classical mathematical models of space mentioned before, from the perspective of Closure Spaces.

Topological and Continuous Metric Spaces

According to classical Topology, a topological space is a pair (X,O) of a set X (of points) and a collection $O \subseteq \mathscr{P}(X)$ of subsets of X called *open sets*, such that $\emptyset, X \in O$, and subject to closure under arbitrary unions and finite intersections. For $A \subseteq X$, $\mathscr{I}(A)$, the *interior* of A, is the largest open set contained in A, that is, the union of all open sets contained in A. It turns out that Topological Spaces are Closure Spaces for which the additional *idempotence* axiom holds as well: $\mathscr{C}(A) = \mathscr{C}(\mathscr{C}(A))$.

Common continuous spaces are the continuous line $(\mathbb{R},\mathscr{C}^1)$, plane $(\mathbb{R}^2,\mathscr{C}^2)$, and the continuous 3-D space $(\mathbb{R}^3,\mathscr{C}^3)$ with classical open and closed sets, and the classical definition of closure, i.e. $\mathscr{C}^n(A) = \overline{\mathscr{I}(\overline{A})}$ where $\overline{A}$ denotes the complement of set A in $\mathbb{R}^n$ and $\mathscr{I}(A)$ is the *interior* of A in $\mathbb{R}^n$.

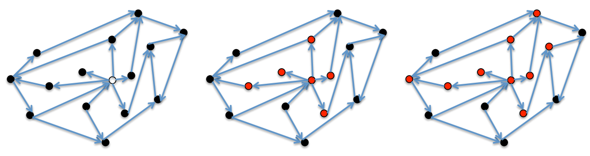
The above spaces can be enriched with a *distance* function, typically a function from the set of pairs to non-negative real numbers, which, depending on the satisfaction of additional specific properties, such as coincidence, symmetry and triangle inequality, can be a *metric* function. In this way, also Metric Spaces (and their variants) are specialisations of Closure Spaces.

Quasi-discrete Closure Spaces and Graphs

An interesing feature of Closure Spaces is that, given a set X and any binary relation $R \subseteq X \times X$, the function $\mathscr{C}_R$ defined as $\mathscr{C}_R(A) = A \cup \{x \in X \mid \exists a \in A.aRx\}$ satisfies the Closure Axioms. Such spaces coincide with so-called Quasi-discrete Closure Spaces (QDCSs, see [17] for details). As a consequence, *any (directed) graph* (X,E), with X the set of vertices and $E \subseteq X \times X$ the set of edges, is the (quasi-discrete) closure space $(X,\mathscr{C}_E)$ (see Fig. 5), possibly enriched with the following additional operations[4]: $\mathbf{Post}(x) = \mathscr{C}_E(\{x\}) \setminus \{x\}$ and $\mathbf{Pre}(x) = \mathscr{C}_{E^{-1}}(\{x\}) \setminus \{x\}$.

We close this section with a brief description of how (edge)-labelled graphs can be easily expressed using Closure Spaces. There are several ways for defining a graph with edges labelled by labels drawn from a set

[4]Here we use a strong version of **Post**, i.e. one in which $x \notin \mathbf{Post}(x)$ even in the case in which xEx. Similarly for **Pre**.



(a) (b) (c)

Figure 5: A sample graph as QDCS: the closure of the singleton containing the white point in (a) is shown in red in (b), while (c) shows the closure of the latter set.

L. A common notation uses a pair (X,E) where $E \subseteq X \times L \times X$. Note that E can also be represented as an L-indexed family of binary relations, that is, $E = \{E_\ell\}_{\ell \in L}$, where for each ℓ, we have $E_\ell \subseteq X \times X$. For instance, with reference to the graph of Fig. 5 and, for the sake presentation, showing labels as different colours of the edges they are associated with, we consider the labelled graph of Fig. 6. For each $\ell \in L$, relation E_ℓ induces

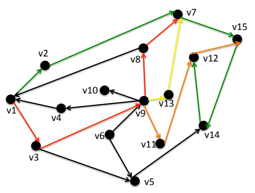
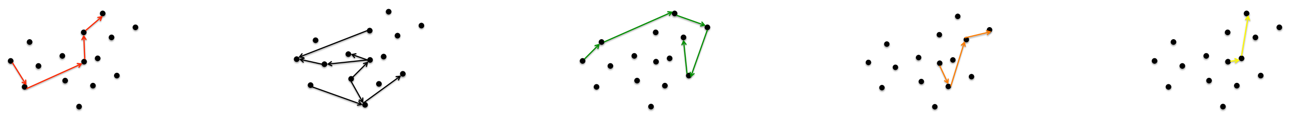


Figure 6: An edge-labelled graph; labels are represented as different colours.

in the standard way the closure $\mathscr{C}_{E_\ell}$ and thus characterises the closure space $(X,\mathscr{C}_{E_\ell})$. Fig. 7 shows the closure spaces which are relevant for the graph of Fig. 6.



(a) (b) (c) (d) (e)

Figure 7: The closure spaces $(V,\mathscr{C}_{R_{Red}})$ (a), $(V,\mathscr{C}_{R_{Black}})$ (b), $(V,\mathscr{C}_{R_{Green}})$ (c), $(V,\mathscr{C}_{R_{Orange}})$ (d), and $(V,\mathscr{C}_{R_{Yellow}})$ (e) are generated by edge relations R_{Red}, R_{Black}, R_{Green}, R_{Orange}, and R_{Yellow}, for the labelled graph of Fig.6.

We observe that the latter indeed *coincides* with the *family of closure spaces* shown in Fig. 7. In fact, we call a *join* of closure spaces any finite family $\{(X,\mathscr{C}_i)\}_{i=1}^k$ of closure spaces sharing the same set X and we represent any labelled graph $(X,\{E_\ell\}_{\ell \in L})$ as the join $\{(X,\mathscr{C}_{E_\ell})\}_{\ell \in L}$.

Finally, also QDCSs can be enriched with a suitable notion of distance, based for instance on edge weights, in the case of graphs labelled with weights.

4.2 Closure Spaces as an Abstract Data Type

In the context of language design and use, ADTs have proven to be an important abstraction mechanism for encapsulation, promoting modularity and facilitating reuse. These features can be especially appreciated in *polymorphic* ADTs, typically in the context of Functional Programming Languages as well as Object Oriented ones. When compared to data structures, ADTs, and in particular polymorphic ADTs, provide *controlled* ways for accessing and updating data as well as *independence* from implementation.

Closure Spaces lend themselves naturally as suitable ADTs for the values of space attributes of CARMA components. In particular, such ADTs can be seen as suitable instantiations of the polymorphic class[5] **CS**$(*)$ of

[5] In this Deliverable we freely use terminology from Object Oriented Languages and Polymorphism in Abstract Data Types. Also, when we say *set* X, it can often be read as *type* X or, more specifically, the *carrier of type* X.

Closure Spaces, where, for each set X, $\mathbf{CS}(X)$ is the class of all closure spaces with X as the set of points (we say "based on X"). The elements of $\mathbf{CS}(X)$ are specific closure spaces; each such space $(X,\mathscr{C})$ is characterised by a *specific* closure operator. At the language implementation level, such ADTs can be offered as suitable *packages* or appropriate language constructs can be provided for polymorphic ADTs definition, instantiation, refinement and enrichment.

In the following, we will show a few examples of how CARMA could be extended with an ADT management kernel in order to provide its users with constructs for the definition and use of ADTs relevant for space-related component attributes and operations.

Continuous Spaces as Refinement of $\mathbf{CS}(*)$

The class $\mathbf{TS}(*)$ of Topological Spaces is the refinement of $\mathbf{CS}(*)$ obtained by requiring idempotence of closure. We have already mentioned the continuous plane. It is natural to represent in CARMA elements in $\mathbf{TS}(\mathbb{R}\times\mathbb{R})$ as:

```
record Point = [ real x, real y];
```

while metric function dist can be defined as:

```
fun real dist( Point p1 , Point p2 ) {
    return sqrt( pow(p1.x−p2.x,2) + pow(p1.y−p2.y,2) );
}
```

In addition, a CARMA component attribute coord could be declared as

```
attrib coord := [ x := expr1 , y := expr2 ];
```

where the coordinates of the component's current position in the Plane are stored.

Under these assumptions, we have for example that the effect of the broadcast action

$$\alpha^{\star}[k \leq \mathsf{dist}(\mathsf{coord},\mathsf{my.coord}) \leq K]\langle a\rangle$$

by (a process of) a component is to send value a to all those components which are willing/able to receive it and which are positioned in the annulus between the circle of radius k and the circle of radius K, centred in the component hosting the process.

It is worth pointing out that, although in the above example the standard CARMA syntax has been used, the fact that the bi-dimensional plane is modelled as $\mathbf{TS}(\mathbb{R}\times\mathbb{R})$ allows for reasoning about space features of the relevant CARMA model in the general framework of closure spaces. In addition, considering $\mathbf{TS}(\mathbb{R}\times\mathbb{R})$ as an ADT, automatic correctness checking tools can be used.

Graphs as Refinement of $\mathbf{CS}(*)$

We have already shown that (labelled) graphs can be easily considered QDCSs, generated by the graph edge relation.

Here we briefly discuss how the CARMA specification language could be extended in order to provide support for the definition of QDCSs. To that purpose, we assume here that other aggregated types are provided: *sets* and *relations*. Moreover, specifically *datatypes* are also introduced to represent QDCSs and to combine multiple QDCSs.

Type `set<t>` is used to identify sets of elements of type `t`. Moreover, if $expr_1,\ldots,expr_k$ are expressions of type `t` the expression

$\{\ expr_1\ ,\ldots,expr_k\ \}$

represents an instance of type `set<t>`. Moreover, if `s` has type `set<t>` and *expr* has type `t`, boolean expression *expr* `in s` is used to check if *expr* belongs to `s`. We also assume to use (binary) functions `union`, `intersection` and `diff` to compute union, intersection and difference of two sets, while *choose* is used to randomly select an element into a set. Standard Java-like iterators can be used to iterate over the elements of a set.

Type `rel<t1,t2>` denotes a binary relation associating elements of type `t1` with elements of type `t2`. Such a mapping is defined by using an expression of the form:

```
{ <$expr_1^1$,$expr_1^2$> ,..., <$expr_k^1$,$expr_k^2$> }
```

where $expr_i^1$ are expressions of type `t1` while $expr_i^2$ are expressions of type `t2`. Starting from *sets* and *binary relations* we could extend CARMA by considering the type `QDCS<t,rel<t,t>>`, where `t` is the datatype of the points in the considered QDCS:

```
const aqdcs := QDCS( { v1,...,vk } , { ...,<vi1,vi2>,... } );
```

above, v_i are values of type `t`, while `{ ...,<`v_i^1`,`v_i^2`>,... }` is an expression of type `rel<t,t>` representing the edges of the graph. Each instance of a QDCS will provide the method `closure` that computes the closure of a set of points of type `t`.

Moreover, using method `closure`, method `post` will implement function `post(v)` which computes the set of elements of type `t` that are associated with `v` in `{ ...,<`v_i^1`,`v_i^1`>,... }`.

The CARMA specification language could be also extended by considering the type constructor `JoinIn` that, received a sequence of pairs builds the collection of multiple QDCSs. With reference to the example of Fig.6 and Fig. 7 we could have something like:

```
const V := { v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12,v13,v14,v15 };
enum Labels = Red, Black, Green, Orange, Yellow;
const R_Red = QDCS( V , { <v1,v3>,<v3,v9>,<v9,v8>,<v8,v7> } );
...
const R_Yellow = QDCS( V , { <v9,v13>,<v13,v7> } );
const Color_Lab_Graph = JoinIn( { < Red , R_Red> , ... , <Yellow,R_Yellow> } );
```

Methods `closure` and `post` on the result of a `JoinIn` are obtained as simple extensions of corresponding operations on QDCSs. So, for example, `Color_Lab_Graph.post( v1 , Red )` will return the set $\{v3\}$.

Combined Spaces as combinations of ADTs

In this section we briefly discuss how one can use *different* models of space within the *same* CARMA model. As an example, consider a smart urban transportation system. A model for such a system would be composed of several different kinds of components, including agents—e.g. buses, taxis or bikers—and service-devices used by agents. Agents move in the city via streets; devices communicate via radio-waves with service-stations located in street-crossings. In such a model, it would be desirable to use a notion of space which *combines both* a *discrete* model of space and the *continuous* Euclidean Plane, as in Fig. 8.

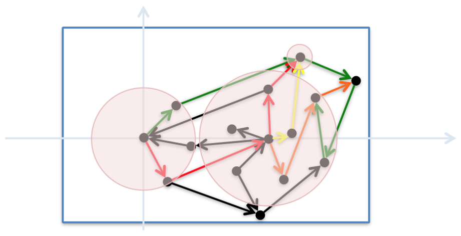


Figure 8: Discrete and Continuous space combined

The discrete space model would be used for the map of the city, with reference to the agents moving around by the streets. For that purpose, in the model, we use the Color_Lab_Graph defined above, where different colours identify different streets and the vertices identify street-crossings or street-begin/-end points. For instance, assume attribute `posn` is declared as having type `V`, and assume its current value for the component modelling a given agent is v1. Then, the agent would move to v3 as a result of the assignment

```
my.posn:= choose( Color_Lab_Graph.post( v1 , Red ) )
```

Note that the graph could be enriched with additional labels, each representing the length of the street segment it would be associated with, or the average time it takes for a pedestrian or a biker (or both) to cover the street-segment[6].

The Euclidean Plane would instead be useful for modelling the behaviour of the devices. One could in fact imagine them running algorithms which depend on their Euclidean distance from the stations, since such devices communicate using radio-signals. For that purpose, in the model, one should also include the Euclidean Plane defined, as above, on the basis of $\mathbf{TS}(\mathbb{R} \times \mathbb{R})$. It could then be useful to define a conversion function `pos2coord` such that, for every point `v` in `V`, `pos2coord(v)` returns the corresponding point in the Euclidean_Plane, in the form of an element of `Point`.

The following CARMA broadcast can be used by a service-station in order to send value b to all those components which are not further than r from the station:

```
β*[dist(pos2coord(posn),pos2coord(my.posn))<= r]<b>
```

In Fig. 8 the areas of interest for three different stations (with different r) are shown in pink.

5 Concluding remarks

In this document we have introduced CARMA, a novel modelling language which aims to represent collectives of agents working in a specified environment and support the analysis of quantitative aspects of their behaviour such as performance, availability and dependability. CARMA is a stochastic process algebra-based language combining several innovative features such as the separation of behaviour and knowledge, locally synchronous and globally asynchronous communication, attribute-defined interaction and a distinct environment which can be changed independently of the agents. We have demonstrated the use of CARMA on a smart taxi system example, showing the ease with which the same system can be studied under different contexts or environments.

Together with the modelling language presented as a stochastic process algebra, we have also introduced a high level language (named the CARMA Specification Language) that can be used as a front-end to support the design of CARMA models and to support quantitative analyses that, currently, are performed via simulation.

Finally, we have presented a possible approach for extending CARMA in order to provide a flexible and structured mechanism for defining common spatial aspects of CAS.

Relations with other WPs. The work presented in this deliverable mainly relates with WP5. Indeed, the development of the CARMA Eclipse plugin, and in particular the definition of the CARMA Specification Language, has been done in strong collaboration with WP5. The proposal for representing spatial information in CARMA via component attributes has be done by relying on the results presented in [16].

Work plan for the next reporting period.

- We envisage providing CARMA with a fluid semantics and in general the exploitation of the specification and analysis techniques developed in WP1 to provide alternative semantic account of CARMA models (see internal report IR1.1 for more details). In this direction we refer also here to [23] where the process language ODELINDA has been proposed which provides an *asynchronous*, tuple-based, interaction paradigm for CAS. The language is equipped both with an individual-based Markovian semantics and with a population-based Markovian semantics. The latter forms the basis for a continuous, fluid-flow, semantics definition, in a way similar to [14].

- We will also use static analysis to identify suitable models for the particular analysis techniques mentioned in the previous point. Indeed, some of these techniques could be applicable only to models with certain characteristics.

[6]Here, by street-segment we intend the portion of a street between two vertices, e.g. two crossings.

- We will continue the development of the CARMA Eclipse plug-in and its evaluation. First, we plan to integrate new tools and features like, for instance, the one implementing the analysis techniques described in the previous point, or the possibility to integrate alternative simulators developed by other research groups. This activity will be done in collaboration with WP5.
- We plan to use the CARMA specification language to model and analyse more and more challenging case studies. Also this activity will be done in strong collaboration with WP5.
- WP4 and WP5 will also collaborate on the development of specific design workflows and analysis pathways. This pathways will involve all the analytical tools (e.g. the statistical model checkers) developed in WP3.

References

[1] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. A calculus for attribute-based communication. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1840–1845, 2015.

[2] Marco Bernardo and Roberto Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 202(1-2):1–54, 1998.

[3] Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MODEST: A compositional modeling formalism for hard and softly timed systems. *IEEE Trans. Software Eng.*, 32(10):812–830, 2006.

[4] Luca Bortolussi, Rocco De Nicola, Cheng Feng, Vashti Galpin, Jane Hillston, Diego Latella, Michele Loreti, Mieke Massink, and Valerio Senni. CAS-SCEL language design. Deliverable D4.1, QUANTICOL Project, 2014.

[5] Luca Bortolussi, Jane Hillston, and Mirco Tribastone. Fluid performability analysis of nested automata models. *Electr. Notes Theor. Comput. Sci.*, 310:27–47, 2015.

[6] Luca Bortolussi, Rocco De Nicola, Vasti Galpin, Stephen Gilmore, Jane Hillston, Diego Latella, Michele Loreti, and Mieke Massink. CARMA: Collective adaptive resource-sharing markovian agents. In *Proc. of the Workshop on Quantitative Analysis of Programming Languages 2015*, 2015. to appear.

[7] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[8] Vincenzo Ciancia, Diego Latella, and Mieke Massink. On Space in CARMA. Quanticol Technical Report TR-QC-01-2015, QUANTICOL, 2015.

[9] Adriana Compagnoni, Paola Giannini, Catherine Kim, Matthew Milideo, and Vishakha Sharma. A calculus of located entities. In *Proc. of DCM 2013*. ArXiv, 2014.

[10] Adriana Compagnoni, Vishakha Sharma, Yifei Bao, Matthew Libera, Svetlana Suhkishvili, Philippe Bidinger, Livio Bioglio, and Eduardo Bonelli. Biospace: A modeling and simulation language for bacteria-materials interactions. *ENTCS*, 293:35–49, 2013.

[11] Rocco De Nicola, Diego Latella, Michele Loreti, and Mieke Massink. A uniform definition of stochastic process calculi. *ACM Comput. Surv.*, 46(1):5, 2013.

[12] Rocco De Nicola, Diego Latella, and Massink Massink. Formal modeling and quantitative analysis of klaim-based mobile systems. In *Proc. of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, pages 428–435. ACM, 2005.

[13] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7, 2014.

[14] Cheng Feng and Jane Hillston. PALOMA: A process algebra for located markovian agents. In *Quantitative Evaluation of Systems - 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings*, volume 8657 of *LNCS*, pages 265–280. Springer, 2014.

[15] Vashti Galpin. Modelling network performance with a spatial stochastic process algebra. In *Proc. of International Conference on Advanced Information Networking and Applications*, pages 41–49. IEEE, 2009.

[16] Vashti Galpin, Luca Bortolussi, Vincenzo Ciancia, Allan Clark, Rocco De Nicola, Cheng Feng, Stephen Gilmore, Nicolas Gast, Jane Hillston, Alberto Lluch-Lafuente, Michele Loreti, Mieke Massink, Laura Nenzi, Daniël Reijsbergen, Valerio Senni, Francesco Tiezzi, Mirco Tribastone, and Max Tschaikowski. A preliminary investigation of capturing spatial information for cas. Deliverable D2.1, QUANTICOL Project, 2014.

[17] Antony Galton. A generalized topological view of motion in discrete space. *Theoretical Computer Science*, 305((1-3)):111–134, 2003.

[18] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403 – 434, 1976.

[19] Stephen Gilmore, Jane Hillston, Leila Kloul, and Marina Ribaudo. PEPA nets: a structured performance modelling formalism. *Performance Evaluation*, 54:79–104, 2003.

[20] Holger Hermanns and Michael Rettelbach. Syntax, Semantics, Equivalences and Axioms for MTIPP. In U. Herzog and M. Rettelbach, editors, *Proc. of 2nd Process Algebra and Performance Modelling Workshop*, 1994.

[21] Jane Hillston. *A Compositional Approach to Performance Modelling*. CUP, 1995.

[22] Mathias John, Cédric Lhoussaine, Joachim Niehren, and Adelinde M. Uhrmacher. The attributed Pi calculus. In *Proc. of Computational Methods in Systems Biology*, volume 5307 of *LNBI*, pages 83–102, 2008.

[23] Diego Latella, Michele Loreti, and Mieke Massink. Investigating fluid-flow semantics of asynchronous tuple-based process languages for collective adaptive systems. In Tom Holvoet and Mirko Viroli, editors, *Prof. of COORDINATION 2015*, volume 9037 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015.

[24] Mieke Massink, Luca Bortolussi, Vincenzo Ciancia, Jane Hillston, Alberto Lluch-Lafuente, Diego Latella, Michele Loreti, Daniël Reijsbergen, and Andrea Vandin. Foundations of scalable verification for stochastic logics. Deliverable D3.1, QUANTICOL Project, 2014.

[25] Carsten Maus, Stefan Rybacki, and Adelinde M. Uhrmacher. Rule-based multi-level modelling of cell biological systems. *BMC Systems Biology*, 5:166, 2011.

[26] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[27] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Y. Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

[28] Julien Saunier, Flavien Balbo, and Suzanne Pinson. A formal model of communication and context awareness in multiagent systems. *Journal of Logic, Language and Information*, 23(2):219–247, 2014.

[29] Alexander Steiniger, Frank Krüger, and Adelinde M. Uhrmacher. Modeling agents and their environment in Multi-Level-DEVS. In *Proc. of the 2012 Winter Simulation Conference*, Berlin, Germany, 2012. IEEE.

[30] Mirco Tribastone, Stephen Gilmore, and Jane Hillston. Scalable differential analysis of process algebra models. *IEEE Transactions on Software Engineering*, 38(1):205–219, 2012.

[31] Danny Weyns and Tom Holvoet. A formal model for situated multi-agent systems. *Fundam. Inform.*, 63(2-3):125–158, 2004.

[32] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.

[33] Martin Wirsing, Hölzl Mathhias, Kock Nora, and Philip Mayer. *Software Engineering for Collective Autonomic Systems: The ASCENS Approach*. Springer, 2015.

A CARMA semantics

A.1 Operational semantics of components

We use the transition relation $\rightharpoondown_{\varepsilon,t} \subseteq \textsc{Comp} \times \textsc{Lab} \times [\textsc{Comp} \to \mathbb{R}_{\geq 0}]$ to define the behaviour of a single component under evaluation context ε at time t. In this relation $[\textsc{Comp} \to \mathbb{R}_{\geq 0}]$ denotes the set of functions from $\textsc{Comp}$ to $\mathbb{R}_{\geq 0}$ and $\textsc{Lab}$ is the set of transition labels ℓ which are generated by the following grammar, where π is defined in Section 2.1:

ℓ	$::=$	$\alpha^\star[\pi]\langle\overrightarrow{v}\rangle,\gamma$	Broadcast output
	$\mid$	$\alpha^\star[\pi](\overrightarrow{v}),\gamma$	Broadcast input
	$\mid$	$\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma$	Unicast Output
	$\mid$	$\alpha[\pi](\overrightarrow{v}),\gamma$	Unicast Input
	$\mid$	$\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]$	Unicast Synchronization
	$\mid$	$\mathscr{R}[\alpha^\star[\pi](\overrightarrow{v}),\gamma]$	Broadcast Input Refusal

The first four labels are associated with the four CARMA input-output actions and contain a reference to the action which is performed (α or $\alpha^\star$), the store of the component where the action is executed (γ), and the value which is transmitted or received. The transition label $\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]$ is associated with *unicast synchronisation*. Label $\mathscr{R}[\alpha^\star[\pi](\overrightarrow{v}),\gamma]$ denotes the case where a component is not able to receive a broadcast output. This arises at the level of the single component either because the associated message has been lost, or because no process is willing to receive that message. We will observe later in this section that the use of $\mathscr{R}[\alpha^\star[\pi](\overrightarrow{v}),\gamma]$ labels are crucial to appropriately handle *dynamic process operators*, namely *choice* and *guard*.

The transition relation $\rightharpoondown_{\varepsilon,t}$, as formally defined in Table 1 and Table 2, is parameterised with respect to an *evaluation context* ε and a time $t \in \mathbb{R}_{\geq 0}$. The former is used to compute the actual rate of process actions and to compute the probability to receive messages, while the latter is the time when the transition is executed and used in the expression evaluation.

The process **nil** denotes the process that cannot perform any action. The transitions associated to this process at the level of components can be derived via rules **Nil** and **Nil-F1**. These rules respectively state that the inactive process cannot perform any action, and always refuses any broadcast input. Note that, the fact that a component $(\textbf{nil},\gamma)$ does not perform any transition is derived from the fact that any label that is not a *broadcast input refusal* leads to function $\emptyset$ (rule **Nil**). Indeed, $\emptyset$ denotes the 0 constant function. Conversely, **Nil-F1** states that $(\textbf{nil},\gamma)$ can always perform a transition labelled $\mathscr{R}[\alpha^\star[\pi](\overrightarrow{v}),\gamma]$ leading to $[(\textbf{nil},\gamma) \mapsto 1]$, where $[C \mapsto v]$ denotes the function mapping the component C to $v \in \mathbb{R}_{\geq 0}$ and all the other components to 0.

The behaviour of a *broadcast output* $(\alpha^\star[\pi_1]\langle\overrightarrow{e}\rangle\sigma.P,\gamma)$ is described by rules **B-Out**, **B-Out-F1** and **B-Out-F2**. Rule **B-Out** states that a broadcast output $\alpha^\star[\pi]\langle\overrightarrow{e}\rangle\sigma$ sends message $[\![\overrightarrow{e}]\!]^t_\gamma$[7] to all components that satisfy $\pi' = \pi[\mathsf{my} \leftarrow \gamma]$. The latter identifies the predicate obtained from π by replacing each attribute $\mathsf{my}.a$ with $\gamma(\mathsf{a})$. The action rate is determined by the evaluation context $\varepsilon = \langle \mu_p, \mu_r, \mu_u \rangle$ and, in particular, by the function μ_r. This function, given a store γ and the kind of action performed, in this case $\alpha^\star$, returns a value in $\mathbb{R}_{\geq 0}$. If this value is greater than 0, it denotes the execution rate of the action. However, the evaluation context can disable the execution of some actions. This happens when $\mu_r(\gamma,\alpha^\star) = 0$. The possible next local stores after the execution of an action are determined by the update σ. This takes the store γ and yields a probability distribution $\mathbf{p} = \sigma(\gamma) \in \mathit{Dist}(\Gamma)$. In rule **B-Out**, and in the rest of the paper, the following notations are used:

- let $P \in \textsc{Proc}$ and $\mathbf{p} \in \mathit{Dist}(\Gamma)$, $(P,\mathbf{p})$ is a probability distribution in $\mathit{Dist}(\textsc{Comp})$ such that:

$$(P,\mathbf{p})(C) = \begin{cases} 1 & P \equiv Q|\textbf{kill} \ \wedge \ C \equiv \mathbf{0} \\ \mathbf{p}(\gamma) & C \equiv (P,\gamma) \ \wedge \ P \not\equiv Q|\textbf{kill} \\ 0 & \text{otherwise} \end{cases}$$

[7]We let $[\![\cdot]\!]^t_\gamma$ denote the evaluation function of an expression/predicate with respect to the store γ and time t.

$$\frac{\ell \neq \mathscr{R}[\alpha^\star[\pi](\overrightarrow{v}),\gamma]}{(\mathbf{nil},\gamma) \xrightarrow{\ell}_{\varepsilon,t} \emptyset}\ \textbf{Nil} \qquad \frac{}{(\mathbf{nil},\gamma) \xrightarrow{\mathscr{R}[\alpha^\star[\pi](\overrightarrow{v}),\gamma]}_{\varepsilon,t} [(\mathbf{nil},\gamma)\mapsto 1]}\ \textbf{Nil-F1}$$

$$\frac{\pi[\mathsf{my}\leftarrow\gamma]=\pi' \quad [\![\overrightarrow{e}]\!]^t_\gamma = \overrightarrow{v} \quad \mathbf{p}=\sigma(\gamma) \quad \varepsilon=\langle\mu_p,\mu_r,\mu_u\rangle}{(\alpha^\star[\pi]\langle\overrightarrow{e}\rangle\sigma.P,\gamma) \xrightarrow{\alpha^\star[\pi']\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mu_r(\gamma,\alpha^\star)\cdot(P,\mathbf{p})}\ \textbf{B-Out}$$

$$\frac{}{(\alpha^\star[\pi_1]\langle\overrightarrow{e}\rangle\sigma.P,\gamma) \xrightarrow{\mathscr{R}[\beta^\star[\pi_2](\overrightarrow{v}),\gamma]}_{\varepsilon,t} [(\alpha^\star[\pi_1]\langle\overrightarrow{e}\rangle\sigma.P,\gamma)\mapsto 1]}\ \textbf{B-Out-F1}$$

$$\frac{\pi[\mathsf{my}\leftarrow\gamma]=\pi' \quad [\![\overrightarrow{e}]\!]^t_\gamma = \overrightarrow{v} \quad \ell\neq\alpha^\star[\pi']\langle\overrightarrow{v}\rangle,\gamma \quad \ell\neq\mathscr{R}[\beta^\star[\pi'](\overrightarrow{v}_1),\gamma]}{(\alpha^\star[\pi]\langle\overrightarrow{e}\rangle\sigma.P,\gamma) \xrightarrow{\ell}_{\varepsilon,t} \emptyset}\ \textbf{B-Out-F2}$$

$$\frac{\pi_2[\overrightarrow{v}/\overrightarrow{x}][\mathsf{my}\leftarrow\gamma_2]=\pi_2' \quad \gamma_1\models\pi_2' \quad \gamma_2\models\pi_1 \quad \mathbf{p}=\sigma[\overrightarrow{v}/\overrightarrow{x}](\gamma_2) \quad \varepsilon=\langle\mu_p,\mu_r,\mu_u\rangle}{(\alpha^\star[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\alpha^\star[\pi_1](\overrightarrow{v}),\gamma_1}_{\varepsilon,t} \mu_p(\gamma_1,\gamma_2,\alpha^\star)\cdot(P[\overrightarrow{v}/\overrightarrow{x}],\mathbf{p})}\ \textbf{B-In}$$

$$\frac{\pi_2[\overrightarrow{v}/\overrightarrow{x}][\mathsf{my}\leftarrow\gamma_2]=\pi_2' \quad \gamma_1\models\pi_2' \quad \gamma_2\models\pi_1 \quad \varepsilon=\langle\mu_p,\mu_r,\mu_u\rangle}{(\alpha^\star[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\mathscr{R}[\alpha^\star[\pi_1](\overrightarrow{v}),\gamma_1]}_{\varepsilon,t} [(\alpha^\star[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2)\mapsto 1-\mu_p(\gamma_1,\gamma_2,\alpha^\star)]}\ \textbf{B-In-F1}$$

$$\frac{\pi_2[\overrightarrow{v}/\overrightarrow{x}][\mathsf{my}\leftarrow\gamma_2]=\pi_2' \quad (\gamma_1\not\models\pi_2' \text{ or } \gamma_2\not\models\pi_1)}{(\alpha^\star[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\alpha^\star[\pi_1](\overrightarrow{v}),\gamma_1}_{\varepsilon,t} \emptyset}\ \textbf{B-In-F2}$$

$$\frac{\ell\neq\alpha^\star[\pi_1](\overrightarrow{v}),\gamma_1 \quad \ell\neq\mathscr{R}[\alpha^\star[\pi_1](\overrightarrow{v}),\gamma_1]}{(\alpha^\star[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\ell}_{\varepsilon,t} \emptyset}\ \textbf{B-In-F3}$$

$$\frac{\alpha\neq\beta}{(\alpha^\star[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\mathscr{R}[\beta^\star[\pi_1](\overrightarrow{v}),\gamma_1]}_{\varepsilon,t} [(\alpha^\star[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2)\mapsto 1]}\ \textbf{B-In-F4}$$

Table 1: Operational semantics of components (Part 1)

- let $\mathbf{c} \in Dist(\textsc{Comp})$ and $r \in \mathbb{R}_{\geq 0}$, $r \cdot \mathbf{c}$ denotes the function $\mathscr{C} : \textsc{Comp} \to \mathbb{R}_{\geq 0}$ such that: $\mathscr{C}(C) = r \cdot \mathbf{c}(C)$

Note that, after the execution of an action a component can be destroyed. This happens when the continuation process after the action prefixing contains the term **kill**. For instance, by applying rule **B-Out** we have that: $(\alpha^{\star}[\pi_1]\langle v\rangle\sigma.(\mathbf{kill}|Q),\gamma) \xrightarrow{\alpha^{\star}[\pi_1]\langle v\rangle,\gamma}_{\varepsilon,t} [\mathbf{0} \mapsto r]$.

Rule **B-Out-F1** states that a *broadcast output* always refuses any *broadcast input*, while **B-Out-F2** states that a *broadcast output* can be only involved in labels of the form $\alpha^{\star}[\pi]\langle\overrightarrow{v}\rangle,\gamma$ or $\mathscr{R}[\beta^{\star}[\pi_2](\overrightarrow{v}),\gamma]$.

Transitions related to a broadcast input are labelled with $\alpha^{\star}[\pi_1](\overrightarrow{v}),\gamma_1$. There, γ_1 is the store of the component executing the output, α is the action performed, π_1 is the predicate that identifies the target components, while $\overrightarrow{v}$ is the sequence of transmitted values. Rule **B-In** states that a component $(\alpha^{\star}[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2)$ can perform a transition with this label when its store γ_2 satisfies the target predicate, i.e. $\gamma_2 \models \pi_1$, and the component executing the action satisfies the predicate $\pi_2[\overrightarrow{v}/\overrightarrow{x}]$. The evaluation context $\varepsilon = \langle\mu_p,\mu_r,\mu_u\rangle$ can influence the possibility to perform this action. This transition can be performed with probability $\mu_p(\gamma_1,\gamma_2,\alpha^{\star})$.

Rule **B-In-F1** models the fact that even if a component can potentially receive a broadcast message, the message can get lost according to a given probability regulated by the evaluation context, namely $1-\mu_p(\gamma_1,\gamma_2,\alpha^{\star})$. Rule **B-In-F2** models the fact that if a component is not in the set of possible receivers ($\gamma_2 \not\models \pi_1$) or the sender does not satisfy the expected requirements ($\gamma_1 \not\models \pi_2'$) then the component cannot receive a broadcast message. Finally, rules **B-In-F3** and **B-In-F4** model the fact that $(\alpha^{\star}[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2)$ can only perform a broadcast input on action α and that it always refuses input on any other action type $\beta \neq \alpha$, respectively.

The behaviour of *unicast output* and *unicast input* is defined by the first six rules of Table 2. These rules are similar to the ones already presented for broadcast output and broadcast input. The only difference is that both unicast output (**Out-F1**) and unicast input (**In-F1**) always refuse any broadcast input with probability 1.

The other rules of Table 2 describe the behaviour of other process operators, namely *choice* $P+Q$, *parallel composition* $P|Q$, *guard* and *recursion*. The term $P+Q$ identifies a process that can behave either as P or as Q. The rule **Plus** states that the components that are reachable by $(P+Q,\gamma)$, via a transition that is not a *broadcast input refusal*, are the ones that can be reached either by (P,γ) or by (Q,γ). In this rule we use $\mathscr{C}_1 \oplus \mathscr{C}_2$ to denote the function that maps each term C to $\mathscr{C}_1(C)+\mathscr{C}_2(C)$, for any $\mathscr{C}_1,\mathscr{C}_2 \in [\textsc{Comp} \to \mathbb{R}_{\geq 0}]$. At the same time, process $P+Q$ *refuses* a broadcast input when both the process P and Q do that. This is modelled by **Plus-F1**, where, for each $\mathscr{C}_1 : \textsc{Comp} \to \mathbb{R}_{\geq 0}$ and $\mathscr{C}_2 : \textsc{Comp} \to \mathbb{R}_{\geq 0}$, $\mathscr{C}_1 + \mathscr{C}_2$ denotes the function that maps each term of the form $(P+Q,\gamma)$ to $\mathscr{C}_1((P,\gamma)) \cdot \mathscr{C}_2((Q,\gamma))$, while any other component is mapped to 0. Note that, differently from rule **Plus**, when rule **Plus-F1** is applied operator $+$ is not removed after the transition. This models the fact that when a broadcast message is refused the choice is not resolved.

In $P|Q$ the two composed processes interleave for all the transition labels except for broadcast input refusal (**Par**). For this label the two processes synchronise (**Par-F1**). This models the fact that a message is lost when both processes refuse to receive it. In the rules the following notations are used:

- for each component C and process Q we let:

$$C|Q = \begin{cases} \mathbf{0} & C \equiv \mathbf{0} \\ (P|Q,\gamma) & C \equiv (P,\gamma) \end{cases}$$

  $Q|C$ is symmetrically defined.

- for each $\mathscr{C} : \textsc{Comp} \to \mathbb{R}_{\geq 0}$ and process Q, $\mathscr{C}|Q$ (resp. $Q|\mathscr{C}$) denotes the function that maps each term of the form $C|Q$ (resp. $Q|C$) to $\mathscr{C}(C)$, while the others are mapped to 0;

- for each $\mathscr{C}_1 : \textsc{Comp} \to \mathbb{R}_{\geq 0}$ and $\mathscr{C}_2 : \textsc{Comp} \to \mathbb{R}_{\geq 0}$, $\mathscr{C}_1|\mathscr{C}_2$ denotes the function that maps each term of the form $(P|Q,\gamma)$ to $\mathscr{C}_1((P,\gamma)) \cdot \mathscr{C}_2((Q,\gamma))$, while the others are mapped to 0.

Rule **Rec** is standard. The behaviour of $([\pi]P,\gamma)$ is regulated by rules **Guard**, **Guard-F1**, **Guard-F2** and **Guard-F3**. The first two rules state that $([\pi]P,\gamma)$ behaves exactly like (P,γ) when γ satisfies predicate π. However, in the first case the *guard* is removed when a transition is performed. In contrast, the *guard* still remains active after the transition when a broadcast input is refused. This is similar to what we consider for

$$\frac{[\![\pi]\!]_{\gamma}=\pi' \quad [\![\overrightarrow{e}]\!]^{t}_{\gamma}=\overrightarrow{v} \quad \mathbf{p}=\sigma(\gamma) \quad \varepsilon=\langle\mu_p,\mu_r,\mu_u\rangle}{(\alpha[\pi]\langle\overrightarrow{e}\rangle\sigma.P,\gamma) \xrightarrow{\alpha[\pi']\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mu_r(\gamma,\alpha)\cdot(P,\mathbf{p})}\ \textbf{Out}$$

$$\frac{}{(\alpha[\pi_1]\langle\overrightarrow{e}\rangle\sigma.P,\gamma_1) \xrightarrow{\mathscr{R}[\beta^{\star}[\pi_2](\overrightarrow{v}),\gamma_2]}_{\varepsilon,t} [(\alpha[\pi_1]\langle\overrightarrow{e}\rangle\sigma.P,\gamma_1)\mapsto 1]}\ \textbf{Out-F1}$$

$$\frac{[\![\pi]\!]_{\gamma}=\pi' \quad [\![\overrightarrow{e}]\!]^{t}_{\gamma}=\overrightarrow{v} \quad \ell\neq\alpha[\pi']\langle\overrightarrow{v}\rangle,\gamma \quad \ell\neq\mathscr{R}[\alpha^{\star}[\pi'](\overrightarrow{v}),\gamma]}{(\alpha[\pi]\langle\overrightarrow{e}\rangle\sigma.P,\gamma) \xrightarrow{\ell}_{\varepsilon,t} \emptyset}\ \textbf{Out-F2}$$

$$\frac{[\![\pi_2[\overrightarrow{v}/\overrightarrow{x}]]\!]_{\gamma_2}=\pi_2' \quad \gamma_1\models\pi_2' \quad \gamma_2\models\pi_1 \quad \mathbf{p}=\sigma[\overrightarrow{v}/\overrightarrow{x}](\gamma_2) \quad \varepsilon=\langle\mu_p,\mu_r,\mu_u\rangle}{(\alpha[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\alpha[\pi_1](\overrightarrow{v}),\gamma_1}_{\varepsilon,t} \mu_p(\gamma_1,\gamma_2,\alpha)\cdot(P[\overrightarrow{v}/\overrightarrow{x}],\mathbf{p})}\ \textbf{In}$$

$$\frac{}{(\alpha[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\mathscr{R}[\beta^{\star}[\pi_1](\overrightarrow{v}),\gamma_1]}_{\varepsilon,t} [(\alpha[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2)\mapsto 1]}\ \textbf{In-F1}$$

$$\frac{[\![\pi_2[\overrightarrow{v}/\overrightarrow{x}]]\!]_{\gamma_2}=\pi_2' \quad (\gamma_1\not\models\pi_2' \text{ or } \gamma_2\not\models\pi_1)}{(\alpha[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\alpha[\pi_1](\overrightarrow{v}),\gamma_1}_{\varepsilon,t} \emptyset}\ \textbf{In-F2} \quad \frac{\ell\neq\alpha[\pi_1](\overrightarrow{v}),\gamma_1 \quad \ell\neq\mathscr{R}[\beta^{\star}[\pi_1](\overrightarrow{v}),\gamma_1]}{(\alpha[\pi_2](\overrightarrow{x})\sigma.P,\gamma_2) \xrightarrow{\ell}_{\varepsilon,t} \emptyset}\ \textbf{In-F3}$$

$$\frac{(P,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}_1 \quad (Q,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}_2 \quad \ell\neq\mathscr{R}[\alpha^{\star}[\pi'](\overrightarrow{v}),\gamma]}{(P+Q,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}_1\oplus\mathscr{C}_2}\ \textbf{Plus}$$

$$\frac{(P,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi'](\overrightarrow{v}),\gamma]}_{\varepsilon,t}\mathscr{C}_1 \quad (Q,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi'](\overrightarrow{v}),\gamma]}_{\varepsilon,t}\mathscr{C}_2}{(P+Q,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi'](\overrightarrow{v}),\gamma]}_{\varepsilon,t}\mathscr{C}_1+\mathscr{C}_2}\ \textbf{Plus-F1}$$

$$\frac{(P,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}_1 \quad (Q,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}_2 \quad \ell\neq\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}{(P|Q,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}_1|Q\oplus P|\mathscr{C}_2}\ \textbf{Par}$$

$$\frac{(P,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}_{\varepsilon,t}\mathscr{C}_1 \quad (Q,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}_{\varepsilon,t}\mathscr{C}_2}{(P|Q,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}_{\varepsilon,t}\mathscr{C}_1|\mathscr{C}_2}\ \textbf{Par-F1} \quad \frac{A\stackrel{\triangle}{=}P \quad (P,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}}{(A,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}}\ \textbf{Rec}$$

$$\frac{\gamma\models\pi \quad (P,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C} \quad \ell\neq\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}{([\pi]P,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\mathscr{C}}\ \textbf{Guard} \quad \frac{\gamma\models\pi \quad (P,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}_{\varepsilon,t}\mathscr{C}}{([\pi]P,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}_{\varepsilon,t}[\pi]\mathscr{C}}\ \textbf{Guard-F1}$$

$$\frac{\gamma\not\models\pi \quad \ell\neq\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}{([\pi]P,\gamma)\xrightarrow{\ell}_{\varepsilon,t}\emptyset}\ \textbf{Guard-F2} \quad \frac{\gamma\not\models\pi}{([\pi]P,\gamma)\xrightarrow{\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}_{\varepsilon,t}[([\pi]P,\gamma)\mapsto 1]}\ \textbf{Guard-F3}$$

Table 2: Operational semantics of components (Part 2)

$$\frac{}{\mathbf{0} \xrightarrow{\ell}_{\varepsilon,t} \emptyset}\ \textbf{Zero} \qquad \frac{(P,\gamma) \xrightharpoonup{\alpha^{\star}[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_1 \quad (P,\gamma) \xrightharpoonup{\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}_{\varepsilon,t} \mathscr{N}_2}{(P,\gamma) \xrightarrow{\alpha^{\star}[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \frac{\mathscr{N}_1 \oplus \mathscr{N}_2}{\oplus\mathscr{N}_1 + \oplus\mathscr{N}_2}}\ \textbf{Comp-B-In}$$

$$\frac{(P,\gamma) \xrightharpoonup{\ell}_{\varepsilon,t} \mathscr{N} \quad \ell \neq \mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]}{(P,\gamma) \xrightarrow{\ell}_{\varepsilon,t} \mathscr{N}}\ \textbf{Comp} \qquad \frac{N_1 \xrightarrow{\alpha^{\star}[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_1 \quad N_2 \xrightarrow{\alpha^{\star}[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_2}{N_1 \parallel N_2 \xrightarrow{\alpha^{\star}[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_1 \parallel \mathscr{N}_2}\ \textbf{B-In-Sync}$$

$$\frac{N_1 \xrightarrow{\alpha^{\star}[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_1^o \quad N_1 \xrightarrow{\alpha^{\star}[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_1^i \quad N_2 \xrightarrow{\alpha^{\star}[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_2^o \quad N_2 \xrightarrow{\alpha^{\star}[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_2^i}{N_1 \parallel N_2 \xrightarrow{\alpha^{\star}[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} (\mathscr{N}_1^o \parallel \mathscr{N}_2^i) \oplus (\mathscr{N}_1^i \parallel \mathscr{N}_2^o)}\ \textbf{B-Sync}$$

$$\frac{N_1 \xrightarrow{\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_1 \quad N_2 \xrightarrow{\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_2}{N_1 \parallel N_2 \xrightarrow{\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_1 \parallel N_2 \oplus N_1 \parallel \mathscr{N}_2}\ \textbf{Out-Sync} \qquad \frac{N_1 \xrightarrow{\alpha[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_1 \quad N_2 \xrightarrow{\alpha[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_2}{N_1 \parallel N_2 \xrightarrow{\alpha[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_1 \parallel N_2 \oplus N_1 \parallel \mathscr{N}_2}\ \textbf{In-Sync}$$

$$\frac{\begin{array}{c} N_1 \xrightarrow{\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]}_{\varepsilon,t} \mathscr{N}_1^s \quad N_1 \xrightarrow{\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_1^o \quad N_1 \xrightarrow{\alpha[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_1^i \\ N_2 \xrightarrow{\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]}_{\varepsilon,t} \mathscr{N}_2^s \quad N_2 \xrightarrow{\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_2^o \quad N_2 \xrightarrow{\alpha[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_2^i \end{array}}{N_1 \parallel N_2 \xrightarrow{\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]}_{\varepsilon,t} \frac{(\mathscr{N}_1^s \parallel N_2)\cdot\oplus\mathscr{N}_1^i}{\oplus\mathscr{N}_1^i + \oplus\mathscr{N}_2^i} \oplus \frac{(N_1 \parallel \mathscr{N}_2^s)\cdot\oplus\mathscr{N}_2^i}{\oplus\mathscr{N}_1^i + \oplus\mathscr{N}_2^i} \oplus \frac{(\mathscr{N}_1^o \parallel \mathscr{N}_2^i)}{\oplus\mathscr{N}_1^i + \oplus\mathscr{N}_2^i} \oplus \frac{(\mathscr{N}_1^i \parallel \mathscr{N}_2^o)}{\oplus\mathscr{N}_1^i + \oplus\mathscr{N}_2^i}}\ \textbf{Sync}$$

Table 3: Operational semantics of collective

the rule **Plus-F1** and models the fact that broadcast input refusals do not remove *dynamic operators*. In rule **Guard-F1** we let $[\pi]\mathscr{C}$ denote the function that maps each term of the form $([\pi]P,\gamma)$ to $\mathscr{C}((P,\gamma)))$ and any other term to 0, for each $\mathscr{C} : \textsc{Comp} \to \mathbb{R}_{\geq 0}$. Rules **Guard-F2** and **Guard-F3** state that no component can be reached from $([\pi]P,\gamma)$ and all the broadcast messages are refused when γ does not satisfy predicate π.

A.2 Operational semantics of collective

The operational semantics of a *collective* is defined via the transition relation $\rightarrow_{\varepsilon,t} \subseteq \textsc{Col} \times \textsc{Lab} \times [\textsc{Col} \to \mathbb{R}_{\geq 0}]$. This relation is formally defined in Table 3. We use a straightforward adaptation of the notations introduced in the previous section.

Rules **Zero**, **Comp-B-In** and **Comp** describe the behaviour of the single component at the level of collective. Rule **Zero** is similar to rule **Nil** of Table 1 and states that inactive component $\mathbf{0}$ cannot perform any action. Rule **Comp-B-In** states that the result of a *broadcast input* of a component at the level of *collective* is obtained by combining (summing) the transition at the level of *components* labelled $\alpha^{\star}[\pi](\overrightarrow{v}),\gamma$ with the one labelled $\mathscr{R}[\alpha^{\star}[\pi](\overrightarrow{v}),\gamma]$. This value is then renormalised to obtain a probability distribution. There we use $\oplus\mathscr{N}$ to denote $\sum_{N\in\textsc{Col}} \mathscr{N}(N)$. The renormalisation guarantees a reasonable computation of *broadcast output* synchronisation rates (see comments on rule **B-Sync** below). Note that each component can always perform a *broadcast input* at the level of collective. However, we are not able to observe if the message has been received or not. Moreover, thanks to renormalisation, if $C \xrightarrow{\alpha^{\star}[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}$ then $\oplus\mathscr{N} = 1$, i.e. $\mathscr{N}$ is a probability distribution over $\textsc{Col}$. Rule **Comp** simply states that for the single component $C \neq \mathbf{0}$ all the transition labels that are not a *broadcast input*, the relation $\xrightarrow{\ell}_{\varepsilon,t}$ coincides with the relation $\xrightharpoonup{\ell}_{\varepsilon,t}$.

Rules **B-In-Sync** and **B-Sync** describe broadcast synchronisation. The former states that two collectives N_1 and N_2 that operate in parallel synchronise while performing a broadcast input. This models the fact that the input can be potentially received by both of the collectives. In this rule we let $\mathscr{N}_1 \parallel \mathscr{N}_2$ denote the function associating the value $\mathscr{N}_1(N_1)\cdot\mathscr{N}_2(N_2)$ with each term of the form $N_1 \parallel N_2$ and 0 with all the other terms. We can observe that if $N \xrightarrow{\alpha^\star[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}$ then, as we have already observed for rule **Comp-B-In**, $\oplus\mathscr{N} = 1$ and $\mathscr{N}$ is in fact a probability distribution over COL.

Rule **B-Sync** models the synchronisation consequent of a *broadcast output* performed at the level of a collective. For each $\mathscr{N}_1 : \text{COL} \to \mathbb{R}_{\geq 0}$ and $\mathscr{N}_2 : \text{COL} \to \mathbb{R}_{\geq 0}$, $\mathscr{N}_1 \oplus \mathscr{N}_2$ denotes the function that maps each term N to $\mathscr{N}_1(N) + \mathscr{N}_2(N)$.

At the level of collective a transition labelled $\alpha^\star[\pi]\langle\overrightarrow{v}\rangle,\gamma$ identifies the execution of a broadcast output. When a collective of the form $N_1 \parallel N_2$ is considered, the result of these kinds of transitions must be computed (in the FUTS style) by considering:

- the broadcast output emitted from N_1, obtained by the transition $N_1 \xrightarrow{\alpha^\star[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_1^o$
- the broadcast input received by N_1, obtained by the transition $N_1 \xrightarrow{\alpha^\star[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_1^i$
- the broadcast output emitted from N_2, obtained by the transition $N_2 \xrightarrow{\alpha^\star[\pi]\langle\overrightarrow{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N}_2^o$
- the broadcast input received by N_2, obtained by the transition $N_2 \xrightarrow{\alpha^\star[\pi](\overrightarrow{v}),\gamma}_{\varepsilon,t} \mathscr{N}_2^i$

Note that the first synchronises with the last to obtain $\mathscr{N}_1^o \parallel \mathscr{N}_2^i$, while the second synchronises with the third to obtain $\mathscr{N}_1^i \parallel \mathscr{N}_2^o$. The result of such synchronisations are summed to model the *race condition* between the broadcast outputs performed within N_1 and N_2 respectively. We have to remark that above $\mathscr{N}_1^o$ (resp. $\mathscr{N}_2^o$) is $\emptyset$ when N_1 (resp. N_2) is not able to perform any broadcast output. Moreover, the label of a broadcast synchronisation is again a *broadcast output*. This allows further synchronisations in a derivation. Finally, it is easy to see that the total rate of a broadcast synchronisation is equal to the total rate of *broadcast outputs*. This means that the number of receivers does not affect the rate of a broadcast that is only determined by the number of senders.

Rules **Out-Sync**, **In-Sync** and **Sync** control the unicast synchronisation. Rule **Out-Sync** states that a collective of the form $N_1 \parallel N_2$ performs a *unicast output* if this is performed either in N_1 or in N_2. This is rendered in the operational semantics as an interleaving rule, where for each $\mathscr{N} : \text{COL} \to \mathbb{R}_{\geq 0}$, $\mathscr{N} \parallel N_2$ denotes the function associating $\mathscr{N}(N_1)$ with each collective of the form $N_1 \parallel N_2$ and 0 with all other collectives. Rule **In-Sync** is similar to **Out-Sync**. However, it considers *unicast input*.

Finally, rule **Sync** regulates the *unicast synchronisations* and generates transitions with labels of the form $\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]$. This is the result of a synchronisation between transitions labelled $\alpha[\pi](\overrightarrow{v}),\gamma$, i.e. an input, and $\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma$, i.e. an output.

In rule **Sync**, $\mathscr{N}_k^s$, $\mathscr{N}_k^o$ and $\mathscr{N}_k^i$ denote the result of synchronisation ($\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]$), unicast output ($\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma$) and unicast input ($\alpha[\pi](\overrightarrow{v}),\gamma$) within N_k ($k = 1,2$), respectively. The result of a transition labelled $\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]$ is therefore obtained by combining:

- the synchronisations in N_1 with N_2: $\mathscr{N}_1^s \parallel N_2$;
- the synchronisations in N_2 with N_1: $N_1 \parallel \mathscr{N}_2^s$;
- the output performed by N_1 with the input performed by N_2: $\mathscr{N}_1^o \parallel \mathscr{N}_2^i$;
- the input performed by N_1 with the output performed by N_2: $\mathscr{N}_1^i \parallel \mathscr{N}_2^o$.

To guarantee a correct computation of synchronisation rates, the first two addendi are renormalised by considering inputs performed in N_2 and N_1 respectively. This, on one hand, guarantees that the total rate of synchronisation $\tau[\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma]$ does not exceed the *output capacity*, i.e. the total rate of $\alpha[\pi]\langle\overrightarrow{v}\rangle,\gamma$ in N_1 and N_2. On the other hand, since synchronisation rates are renormalised during the derivation, it also ensures that parallel composition is associative [11].

$$\frac{\rho(t,\gamma_g,N)=\varepsilon=\langle\mu_r,\mu_p,\mu_u\rangle \quad N \xrightarrow{\alpha^\star[\pi]\langle\vec{v}\rangle,\gamma}_{\varepsilon,t} \mathscr{N} \quad \mu_u(\gamma_g,\alpha^\star)=(\sigma,N')}{N \textbf{ in } (\gamma_g,\rho) \xmapsto{\alpha^\star[\pi]\langle\vec{v}\rangle,\gamma}_t \mathscr{N} \parallel N' \textbf{ in } (\sigma(\gamma_g),\rho)}\ \textbf{Sys-B}$$

$$\frac{\rho(t,\gamma_g,N)=\varepsilon=\langle\mu_r,\mu_p,\mu_u\rangle \quad N \xrightarrow{\tau[\alpha[\pi]\langle\vec{v}\rangle,\gamma]}_{\varepsilon,t} \mathscr{N} \quad \mu_u(\gamma_g,\alpha)=(\sigma,N')}{N \textbf{ in } (\gamma_g,\rho) \xmapsto{\tau[\alpha[\pi]\langle\vec{v}\rangle,\gamma]}_t \mathscr{N} \parallel N' \textbf{ in } (\sigma(\gamma_g),\rho)}\ \textbf{Sys}$$

Table 4: Operational Semantics of Systems.

A.3 Operational semantics of systems

The operational semantics of systems is defined via the transition relation $\mapsto_t \subseteq \textsc{Sys} \times \textsc{Lab} \times [\textsc{Sys} \to \mathbb{R}_{\geq 0}]$ that is formally defined in Table 4. Only synchronisations are considered at the level of systems.

The first rule is **Sys-B**. This rule states that a system of the form N **in** (γ_g,ρ) can perform a *broadcast output* when the collective N, under the environment evaluation $\varepsilon=\langle\mu_r,\mu_p,\mu_u\rangle=\rho(\gamma_g,N)$, can evolve at the level of collective with the label $\alpha^\star[\pi]\langle\vec{v}\rangle,\gamma$ to $\mathscr{N}$. After the transition, the global store is updated and a new collective can be created according to function μ_u. In rule **Sys-B** the following notations are used. For each collective N_2, $\mathscr{N}:\textsc{Col}\to\mathbb{R}_{\geq 0}$, $\mathscr{S}:\textsc{Sys}\to\mathbb{R}_{\geq 0}$ and $\mathbf{p}\in \mathit{Dist}(\Gamma)$ we let $\mathscr{N}$ **in** $(\mathbf{p},\rho)$ denote the function mapping each system N **in** (γ,ρ) to $\mathscr{N}(N)\cdot\mathbf{p}(\gamma)$. The second rule is **Sys** that is similar to **Sys-B** and regulates unicast synchronisations.

B The CARMA model of the Smart Taxi System

```
const SIZE = 3;
const K = 5;

const R_T = 12.0;
const R_C = 6.0;
const R_A = 1.0;
const R_STEP = 1.0;

const P_LOST = 0.2;

record Position = [ int x , int y ];

fun Position Roving(){
    int pos_x := U(0,1,2);
  int pos_y := 0;
  if (pos_x == 1) {
    pos_y := U(0,2);
  } else {
    pos_y := U(0,1,2);
  }
    return [ x := pos_x , y:= pos_y ];
}

fun Position DestLoc(real time, Position g){
  Position q := [ x := 0 , y := 0 ];
    if( g.x == 1 && g.y == 1){
      q := Roving();
   } else{
      q := [ x := 1 , y := 1 ];
    }
    return q;
```

```
}

fun real Mrate( Position l1, Position l2){
  real t := real( abs(l1.x − l2.x) + abs(l1.y − l2.y) );
  real r := 0.0;
  if(t > 1.0){
    r := R_STEP / t;
  } else{
    r := R_STEP;
  }
  return r;
}

fun real Arate(real time, Position l1){
  real r := 0.0;
  if ((l1.x == 1)&&(l1.y==1)) {
    if(time < 200)
      r := R_A / 4.0;
    else{
      r := 3.0 ∗ R_A / 4.0;
    }
  }
  else{
    if(time < 20)
      r := 3.0 ∗ R_A / 4.0;
    else{
      r := R_A / 4.0;
    }
  }
  return r;
}

fun real Takeprob( int taxisAtLoc ){
  real x_:= 0.0;
  if (taxisAtLoc == 0){
    x_ := 0.0;
  }
  else{
    x_ := 1.0/real(taxisAtLoc);
  }
  return x_;
}

component User(Position g, Position h, process Z){

    store{
        attrib loc := g;
        attrib dest := h;
    }

    behaviour{
        Wait = call∗[true]<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;
  }

    init{
        Z
    }
}

component Taxi(int a, int b, int c, int d, int e, process Z){
    store{
        attrib loc := [ x:= a , y:= b];
        attrib dest := [ x:=c , y:=d];
        attrib occupancy := e;
    }

    behaviour{
        F = take[true](posx,posy){dest := [x:=posx,y:=posy], occupancy := 1}.G +
          call*[(my.loc.x != posx)&&(my.loc.y !=posy)](posx,posy){dest := [ x:=posx,y
              :=posy] }.G;
        G = move*[false]<>{loc := dest, dest := [x:=3,y:=3], occupancy := 0}.F;
    }

    init{
        Z
    }
}

component Arrival(int a, int b){

  store{
    attrib loc := [ x:=a , y:= b];
  }

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

  init{
    A
  }
}

  measure WaitingUser[ i := 0:SIZE-1 , j := 0:SIZE-1 ] = #{User[Wait] | my.loc.x == i
       && my.loc.y == j };
  measure FreeTaxi[ i := 0:SIZE-1 , j := 0:SIZE-1 ] = #{Taxi[F] | my.loc.x == 0 && my
      .loc.y == 0 };
  measure All_User = #{User[*] | true };

system Scenario1{

    collective{
      for( i ; i<K ; i+1 ) {
          new Taxi(0:SIZE-1,0:SIZE-1,3,3,0,F);
        }
      new Arrival(0:SIZE-1,0:SIZE-1);
    }

    environment{


      prob{
              [true] take : Takeprob(#{Taxi[F] | my.loc == sender.loc });
              [true] call* : 1-P_LOST;
              default : 1.0;
      }

        rate{
          [true] take : R_T;
        [true] call* : R_C;
        [true] move* : Mrate(sender.loc,sender.dest);
        [true] arrival* : R_A * (1.0 / real( SIZE * SIZE )) ;
        default : 0.0;
        }

        update{
          [true] arrival* : new User(sender.loc,DestLoc(now,sender.loc), Wait);
        }
    }

}

system Scenario2{

    collective{
      for( i ; i<K ; i+1 ) {
          new Taxi(0:SIZE-1,0:SIZE-1,3,3,0,F);
        }
      new Arrival(0:SIZE-1,0:SIZE-1);
    }

    environment{

      prob{
              [true] take : Takeprob(#{Taxi[F] | my.loc == sender.loc });
              [true] call* : 1-P_LOST;
              default : 1.0;
      }

        rate{
          [true] take : R_T;
        [true] call* : R_C;
        [true] move* : Mrate(sender.loc,sender.dest);
        [true] arrival* : Arate(now,sender.loc);
        default : 0.0;
        }

        update{
          [true] arrival* : new User(sender.loc,DestLoc(now,sender.loc), Wait);
        }
    }

}
```