QUANTICOL

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

quantiCOL

http://www.quanticol.eu

TR-QC-1-2015

On Space in CARMA

Revision: 1.2; June 17, 2015

Author(s): Vincenzo Ciancia (CNR), Diego Latella (CNR), Mieke Massink (CNR)

Publication date: June 17, 2015

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

COOPERATION

Contents

1 Introduction

In these notes we provide some pragmatic considerations, based on notions of Abstract Data Types, on how space information can be incorporated in CARMA. We assume that spatial information is modelled in the language using component attributes. More specifically, we assume that space is modelled by an abstract data type to be used for the values of space attribute and in particular that each component has an attribute, posn that takes as value the point in space where the component is in the current global state—where the meaning of "point in space" depends on the notion of space which is used in a specific system model, i.e. the specific type(s) used; so it can be a point in a n-dimensional Euclidean Space, or a vertex in a graph, or an IP address, etc. Furthermore, depending on the specific notion of space there can be more than one component in the same point.

The specification of the space type to be used in a system model $N \,\mathbf{in}\, \mathcal{E}$ is assumed to be given in the environment $\mathcal{E}$ of the model. We refrain here from suggesting any specific language for defining the space type(s); we rather present some hints on the more conceptual framework which could be used for the definition of such a language.

2 Space as a type

We consider the (parametric) class[1] $\mathsf{CS}(*)$ of *Closure Spaces*, where, for each set X, $\mathsf{CS}(X)$ is the class of all closure spaces with X as set of points (we say "based on X"). The elements of $\mathsf{CS}(X)$ are specific closure spaces. Each such a space $(X, \mathcal{C})$ is characterized by a *specific* closure operator, that is a function $\mathcal{C} : 2^X \to 2^X$ satisfying the following axioms:

Definition 1 (Closure Axioms)**.**

C1: $\mathcal{C}(\emptyset) = \emptyset$*;*

C2: $A \subseteq \mathcal{C}(A)$*, for all* A*;*

C3: $\mathcal{C}(A \cup B) = \mathcal{C}(A) \cup \mathcal{C}(B)$*, for all* A, B

We refer to [5, 2, 3] for details on closure spaces. Here we only want to point out that the class $\mathsf{CS}(*)$ is sufficient for expressing the most common notions of space which can be used as basic types for (space) attribute values of CARMA components (see [4]). This can be done via suitable instantiations, enrichments and refinements of the class $\mathsf{CS}(*)$, as we will show later on in Sect 2.1, Sect. 2.2, Sect 2.3 and Sect. 3.

Let us consider the class $\mathsf{CS}(X)$ of all closure spaces based on X. A *join* in $\mathsf{CS}(X)$ is any finite family $\{(X, \mathcal{C}_i)\}_{i=1}^k$ of closure spaces in $\mathsf{CS}(X)$. Furthermore, let us define operator $\sqcup_{i=1}^k : (2^X \to 2^X)^k \to (2^X \to 2^X)$ with $(\sqcup_{i=1}^k \mathcal{C}_i)(A) = \bigcup_{i=1}^k \mathcal{C}_i(A)$ for all $A \subseteq X$.

Proposition 1. *For join* $\{(X, \mathcal{C}_i)\}_{i=1}^k$*,* $(X, (\sqcup_{i=1}^k \mathcal{C}_i))$ *is in* $\mathsf{CS}(X)$*.*

Proof.

C1: $(\sqcup_{i=1}^k \mathcal{C}_i)(\emptyset) = \bigcup_{i=1}^k \mathcal{C}_i(\emptyset) = \bigcup_{i=1}^k \emptyset = \emptyset$;

C2: choose $h \in \{1, \ldots, k\}$, recalling that $\mathcal{C}_h$ is a closure: $A \subseteq \mathcal{C}_h(A) \subseteq \bigcup_{i=1}^k \mathcal{C}_i(A) = (\sqcup_{i=1}^k \mathcal{C}_i)(A)$;

[1] In this note we freely use terminology from Object Oriented Languages and Parametric 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. We do not intend to suggest any specific choice for the theoretical framework to be used as underlying semantics for CARMA space attribute values, neither any specific syntactical construct. We leave these choices to the actual language design activity.

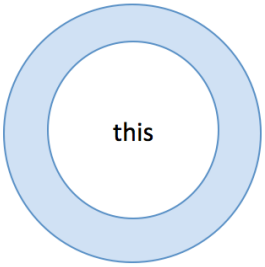

Figure 1: An annulus of components

C3: $(\sqcup_{i=1}^{k}\mathcal{C}_i)(A\cup B) = \bigcup_{i=1}^{k}\mathcal{C}_i(A\cup B) = \bigcup_{i=1}^{k}(\mathcal{C}_i(A)\cup\mathcal{C}_i(B)) = (\bigcup_{i=1}^{k}\mathcal{C}_i(A))\cup(\bigcup_{i=1}^{k}\mathcal{C}_i(B)) = (\sqcup_{i=1}^{k}\mathcal{C}_i)(A)\ \cup\ (\sqcup_{i=1}^{k}\mathcal{C}_i)(B)$

□

So, for all X, $\mathsf{CS}(X)$ is closed under join closure unions, and this allows us to define, the *merge* operator $\mathcal{M} : 2^{\mathsf{CS}(*)} \to \mathsf{CS}(*)$ with $\mathcal{M}(\{(X,\mathcal{C}_i)\}_{i=1}^{k}) = (X, \sqcup_{i=1}^{k}\mathcal{C}_i)$, for each join $\{(X,\mathcal{C}_i)\}_{i=1}^{k}$.

2.1 Continuous Space(s): refinement of CS(∗)

The mono-, bi-, three-, and n-dimensional (continuous) Euclidian Spaces can be seen as *topological spaces*. The (parametric) class $\mathsf{TS}(*)$ of *Topological Spaces* can be defined as a refinement of $\mathsf{CS}(*)$, by adding an additional axiom [5, 2]:

Definition 2. *$\mathsf{TS}(*)$ is the (parametric) class of* Topological Spaces*, defined as a (parametric) subclass of $\mathsf{CS}(*)$. In particular, for each set X, the class $\mathsf{TS}(X)$ is composed of those closure spaces $(X,\mathcal{C})$ in $\mathsf{CS}(X)$ such that the following additional fourth axiom holds:*

C4: $\mathcal{C}(\mathcal{C}(A)) = \mathcal{C}(A)$ for all A. □

Note that Proposition 1 does not hold for (non-trivial) topological spaces since

$(\sqcup_{i=1}^{k}\mathcal{C}_i)((\sqcup_{i=1}^{k}\mathcal{C}_i)(A)) = \bigcup_{i=1}^{k}\mathcal{C}_i(\bigcup_{i=1}^{k}\mathcal{C}_i(A)) = \bigcup_{i=1}^{k}(\mathcal{C}_i(\mathcal{C}_i(A))\bigcup_{j=1,j\neq i}^{k}\mathcal{C}_i(\mathcal{C}_j(A))) = \bigcup_{i=1}^{k}(\mathcal{C}_i(A))\bigcup_{j=1,j\neq i}^{k}\mathcal{C}_i(\mathcal{C}_j(A))\supset\bigcup_{i=1}^{k}(\mathcal{C}_i(A)) = (\sqcup_{i=1}^{k}\mathcal{C}_i)(A)$.

Common continuous spaces are $(\mathbb{R},\mathcal{C}^1)$ in $\mathsf{TS}(\mathbb{R})$, $(\mathbb{R}^2,\mathcal{C}^2)$ in $\mathsf{TS}(\mathbb{R}^2)$, and $(\mathbb{R}^3,\mathcal{C}^3)$ in $\mathsf{TS}(\mathbb{R}^3)$ with classical open and closed sets, and the classical definition of closure, i.e. $\mathcal{C}^n(A) = \overline{\mathcal{I}(\overline{A})}$ where $\overline{A}$ denotes the complement of set A in $\mathbb{R}^n$ and $\mathcal{I}(A)$ is the *interior* of A in $\mathbb{R}^n$, i.e. the union of all open sets in $\mathbb{R}^n$ which are included in A. Cartesian representation $(x_1,\ldots,x_n)$ is assumed for the elements of $\mathbb{R}^n$, which is extended to $\mathsf{posn} = (\mathsf{posn}_1,\ldots,\mathsf{posn}_n)$ in the obvious way.

A simple example of use of $(\mathbb{R}^2,\mathcal{C}^2)$ in $\mathsf{TS}(\mathbb{R}^2)$ in CARMA is the following action

$$\alpha^*[k \le (\mathsf{posn}_1 - \mathsf{this.posn}_1)^2 + (\mathsf{posn}_2 - \mathsf{this.posn}_2)^2 \le K]\langle v\rangle\{\}$$

by means of which a process performs a broadcast action α sending value v, to all those components which are willing/able to receive it and which are positioned in the annulus between the circle of radius $\sqrt{k}$ and the circle of radius $\sqrt{K}$, centred in the component hosting the process, i.e. in the light-blue area in Fig. 1.

Finally, one can further refine $(\mathbb{R}^n,\mathcal{C}^n)$ in $\mathsf{TS}(\mathbb{R}^n)$ using topological spaces based on the (hyper-)rectangles $[a_1,b_1]\times\ldots\times[a_n,b_n]$, and their open variants, for $a_1,b_1\ldots a_n,b_n\in\mathbb{R}$, and appropriate closure functions (e.g. the classical one).

2.2 Graphs and Quasi-discrete Spaces

An interesing feature of Closure Spaces is that, given a set X and any binary relation $R \subseteq X \times X$, the function $\mathcal{C}_R$ defined as $\mathcal{C}_R(A) = A \cup \{x \in X \mid \exists a \in A.a\, R\, x\}$ satifies axioms C1, C2, and C3 (but not necessarily C4, typically). So, for instance, *any graph on* X is an element of $\mathsf{CS}(X)$. Such spaces coincide with so-called Quasi-discrete Closure Spaces (see [5] for details). Notable examples of Quasi-discrete Closure Spaces are the mono-, bi-, three-, and n-dimensional infinite regular grids, i.e. those elements of $\mathsf{CS}(\mathbb{Z})$, $\mathsf{CS}(\mathbb{Z}^2)$, $\mathsf{CS}(\mathbb{Z}^3)$, $\mathsf{CS}(\mathbb{Z}^n)$ characterized by the closure $\mathcal{C}_A$ induced by the standard binary adjacency relation A recalled below[2]:

$$A = \{((x_1, \ldots, x_n), (y_1, \ldots, y_n)) \in \mathbb{Z}^n \times \mathbb{Z}^n \mid \bigvee_{i=1}^{n} \left(|x_i - y_i| = 1 \wedge_{j=1, j\neq i}^{n} x_j = y_j\right)\}$$

where $\mathbb{Z}$ is the set of integer numbers. As for the continuous case, one can consider finite grids $\mathsf{CS}(\{k_1, \ldots, m_1\} \times \ldots \times \{k_n, \ldots, m_n\})$ for $k_i < m_i \in \mathbb{Z}$. for $i = 1, \ldots, n$ and closure $\mathcal{C}_{F_A}$ induced by $F_A = A_{|_{\{k_1,\ldots,m_1\}\times\ldots\times\{k_n,\ldots,m_n\}}}$

2.2.1 Bi-dimensional Grid with directions as enrichment of $\mathsf{CS}(\mathbb{Z}^2)$

The bi-dimensional Grid with directions 2GridD is obtained by adding specific operations for the four cardinal directions:

Definition 3. *2GridD is the closure space* $(\mathbb{Z}^2, \mathcal{C}_A)$ *in* $\mathsf{CS}(\mathbb{Z}^2)$ *enriched with the following additional operations*

- **North**$(x_1, x_2) = (x_1, x_2 + 1)$
- **South**$(x_1, x_2) = (x_1, x_2 - 1)$
- **East**$(x_1, x_2) = (x_1 + 1, x_2)$
- **West**$(x_1, x_2) = (x_1 - 1, x_2)$ □

The following is a simple example of a CARMA process randomly[3] walking in the grid:

$Walk =$
walk$[\top]\langle\bullet\rangle\{$this.posn $\leftarrow$ **North**(this.posn)$\}.Walk +$
walk$[\top]\langle\bullet\rangle\{$this.posn $\leftarrow$ **South**(this.posn)$\}.Walk +$
walk$[\top]\langle\bullet\rangle\{$this.posn $\leftarrow$ **East**(this.posn)$\}.Walk +$
walk$[\top]\langle\bullet\rangle\{$this.posn $\leftarrow$ **West**(this.posn)$\}.Walk$

2.2.2 Three-dimensional Grid with directions as enrichment of $\mathsf{CS}(\mathbb{Z}^3)$

In a pretty similar way as for the bi-dimensional Grid with directions 2GridD we can define the three-dimensional Grid with directions 3GridD:

Definition 4. *3GridD is the closure space* $(\mathbb{Z}^3, \mathcal{C}_A)$ *in* $\mathsf{CS}(\mathbb{Z}^3)$ *enriched with the following additional operations*

- **North**$(x_1, x_2, x_3) = (x_1, x_2 + 1, x_3)$
- **South**$(x_1, x_2, x_3) = (x_1, x_2 - 1, x_3)$

[2] Again, the Cartesian representation $(x_1, \ldots, x_n)$ is used for the elements of $\mathbb{Z}^n$.

[3] We recall here that the precise rates and probabilities associated with specific actions are to be defined in the environment part of the relevant CARMA model specification.

- $\mathbf{East}(x_1, x_2, x_3) = (x_1 + 1, x_2, x_3)$
- $\mathbf{West}(x_1, x_2, x_3) = (x_1 - 1, x_2, x_3)$
- $\mathbf{Up}(x_1, x_2, x_3) = (x_1, x_2, x_3 + 1)$
- $\mathbf{Down}(x_1, x_2, x_3) = (x_1, x_2, x + 3 - 1)$ □

The following is a simple example of a CARMA process randomly walking&jumping in the grid:

$WalkAndJump =$
$\mathsf{walk}[\top]\langle\bullet\rangle\{\mathsf{this.posn} \leftarrow \mathbf{North}(\mathsf{this.posn})\}.WalkAndJump +$
$\mathsf{walk}[\top]\langle\bullet\rangle\{\mathsf{this.posn} \leftarrow \mathbf{South}(\mathsf{this.posn})\}.WalkAndJump +$
$\mathsf{walk}[\top]\langle\bullet\rangle\{\mathsf{this.posn} \leftarrow \mathbf{East}(\mathsf{this.posn})\}.WalkAndJump +$
$\mathsf{walk}[\top]\langle\bullet\rangle\{\mathsf{this.posn} \leftarrow \mathbf{West}(\mathsf{this.posn})\}.WalkAndJump +$
$\mathsf{walk}[\top]\langle\bullet\rangle\{\mathsf{this.posn} \leftarrow \mathbf{Up}(\mathsf{this.posn})\}.WalkAndJump +$
$\mathsf{walk}[\top]\langle\bullet\rangle\{\mathsf{this.posn} \leftarrow \mathbf{Down}(\mathsf{this.posn})\}.WalkAndJump$

2.2.3 Directed Graphs as enrichment of $\mathsf{CS}(V)$

Directed graphs are readily defined as follows:

Definition 5. *A Directed Graph (V, E), with V the set of vertices and $E \subseteq V \times V$ the set of edges, is the closure space $(V, \mathcal{C}_E)$ in $\mathsf{CS}(V)$ enriched with the following additional operations*[4]*:*

- $\mathbf{Post}(v) = \mathcal{C}_E(\{v\}) \setminus \{v\}$
- $\mathbf{Pre}(v) = \mathcal{C}_{E^{-1}}(\{v\}) \setminus \{v\}$

2.2.4 Edge Labelled Graphs as Closure Space join

Let (V, E) be a Directed Graph, with V the set of vertices and $E \subseteq V \times V$ the set of edges, L be a finite set of labels, and consider an edge labelling total function $\mathcal{L} : E \to L$ mapping each edge $e \in E$ to a label $\ell \in L$. So, the graph can be obtained via enrichment of $\mathsf{CS}(V)$, by adding edge-labelling operators $\mathbf{Elab}_E : E \to L$.

An alternative view follows. Let $\{E_\ell\}_{\ell \in L}$ be the standard kernel partition induced by $\mathcal{L}$, i.e. $\{E_\ell\}_{\ell \in L}$ where $E_\ell = \{e \in E \mid \mathcal{L}(e) = \ell\}$. For each $\ell \in L$, relation E_ℓ induces in the standard way[5] the closure $\mathcal{C}_{E_\ell}$—abbreviated by $\mathcal{C}_\ell$ in the sequel—and thus characterizes the closure space $(V, \mathcal{C}_\ell)$. Then, the $\mathcal{L}_E$-edge-labelled graph (V, E) is the join $\{(V, \mathcal{C}_\ell)\}_{\ell \in L}$. We enrich $\mathsf{CS}(V)$ with operation **Take** defined as follows for $v \in V$ and $L' \subseteq L$:

- $\mathbf{Take}(L', v) = (\sqcup_{\ell \in L'} \mathcal{C}_\ell)(\{v\}) \setminus \{v\}$

So $\mathbf{Take}(L', v)$ returns the set of those vertices $v' \neq v$ which are connected to v via an edge with label in L'. Equivalently, one can consider the merge $(X, \mathcal{C}_{L'}) = \mathcal{M}(\{(X, \mathcal{C}_\ell)\}_{\ell \in L'})$ and define $\mathbf{Take}(L', v)$ as $\mathbf{Take}(L', v) = \mathcal{C}_{L'}(\{v\}) \setminus \{v\}$. Note that $\mathcal{M}(\{(X, \mathcal{C}_\ell)\}_{\ell \in L})$ coincides with the *unlabelled* graph (V, E).

In the case in which labels are tuples of n elements, i.e. $L = \times_{j=1}^n L_j$, function $\mathcal{L}$ is the product $\times_{j=1}^n \mathcal{L}_j$ of functions $\mathcal{L}_j$ with $\mathcal{L}_j : E \to L_j$. For each j we can consider the partition $\{E_{\ell_j}\}_{\ell_j \in L_j}$ of E induced by $\mathcal{L}_j$. Clearly, for $\ell = \langle \ell_1, \ldots, \ell_j, \ldots, \ell_n \rangle \in L$ we have that $E_\ell \subseteq E_{\ell_j}$ for $j = 1, \ldots, n$. Each relation E_{ℓ_j} characterizes a closure $\mathcal{C}_{\ell_j}$ and related closure space with

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

[5] We recall that $\mathcal{C}_{E_\ell}(A) = A \cup \{v \in V \mid \exists a \in A. a\, E_\ell\, v\}$.

points in V. It is easy to see that, for $j = 1 \dots n$, for label $\overline{\ell} \in L_j$ the following holds, where, for $\ell \in L$, $\ell[j]$ denotes the j-th element of ℓ: $\mathcal{C}_{\overline{\ell}} = \sqcup_{\ell \in L : \ell[j] = \overline{\ell}} \mathcal{C}_\ell$. Funtion **Take** is readily extended to the case of tuple labels, for $v \in V$, $L = \times_{j=1}^{n} L_j$ and $L' \subseteq L_j$:

- $\mathbf{Take}(L', j, v) = (\sqcup_{\ell \in L'} \mathcal{C}_\ell)(\{v\}) \setminus \{v\}$ where $\mathcal{C}_\ell$ is the closure defined by relation E_ℓ, element of the partition $\{E_{\ell_j}\}_{\ell_j \in L_j}$ induced by $\mathcal{L}_j$.

In the sequel, we will use often the notation $\mathsf{QDCS}(*)$ instead of $\mathsf{CS}(*)$ when we want to enphasize that we are dealing with a Quasi-discrete closure space generated by a binary relation.

2.3 Metric and Distance Spaces

It is often useful to enrich the class $\mathsf{CS}(*)$, or its refinements, with a *metric* function:

Definition 6. *$\mathsf{MS}(*)$ is the (parametric) class of* Metric Spaces*, defined as the (parametric) subclass of $\mathsf{CS}(*)$ enriched with a function $\mathbf{d} : * \times * \to \mathbb{R}_{\geq 0} \cup \{+\infty\}$ such that:*

D1: $\mathbf{d}(x, y) = 0$ iff $x = y$;

D2: $\mathbf{d}(x, y) = \mathbf{d}(y, x)$;

D3: $\mathbf{d}(x, z) \leq \mathbf{d}(x, y) + \mathbf{d}(y, z)$

for all x, y, z. □

In some cases, requiring all of D1 to D3 to hold can be too restrictive. For instance, due to one way roads it may happen that di distance to be covered to go from x to y is different from that to be covered for going from y to x, i.e. $\mathbf{d}(x, y) \neq \mathbf{d}(y, x)$—this is always the case for directed graphs with non-symmetric edge relation—or one might be interested in approximating a distance by using just a finite set of values like 1 for "short distance", 2 for "medium distance", and 3 for "long distance", in which case D3 would not make any sense ("short" plus "short" is not "medium" ...). In all these cases it might be more useful to use a *distance* function instead of a metric function: a distance function is a function in $* \times * \to \mathbb{R}_{\geq 0}$ for which only D1 is required to hold. Finally, sometimes it might be useful to use a *pseudo-metric* which is a function satisfying

D1': $\mathbf{d}(x, x) = 0$

instead of D1, so that *different* points in space may be at distance 0. A classical example of a metric is the Euclidean distance in $(\mathbb{R}^2, \mathcal{C})$ with

$$\mathbf{d}((x_1, x_2), (y_2, y_2)) = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2}$$

A common example of use of $(\mathbb{R}^2, \mathcal{C})$ with Euclidean metric $\mathbf{d}$ in CARMA is the action

$$\alpha^*[\mathbf{d}(\mathsf{posn}, \mathsf{this.posn}) \leq d]\langle v \rangle \{\}$$

by means of which a process performs a *broadcast output* action α, sending value v, to all those components which are closer than d from the hosting component, and which are willing/able to receive it (actions also have *updates*, which in our examples are the empty set $\{\}$).

A typical distance function in directed graphs $(V, \mathcal{C}_E)$ is

$$\mathbf{d}(v, v') = \min\{n \mid \exists v_0 = v, \dots, v_n = v'.n > 0 \Longrightarrow \wedge_{j=0}^{n-1} v_j \, E \, v_{j+1}\} \text{ with } \min \emptyset = +\infty$$

that is, the length of the *minimal path* from v to v', if such a path exists (otherwise ∞).

3 An Example

In this section we consider a bike-sharing system. The system consists of a set of bike-stations, each located in a different place of a city and a set of bike-agents. An agent starts from a place where a bike-station is located and repeatedly chooses (at random) a place in the city where the destination station is located and—with the help of its (local) planner—heads to the chosen destination. The city is represented by a city-map where each street is composed of several segments, each segment being identified by the portion of the street along a block. Segments of the same street are connected by city places, typically corresponding to street crossings (but they can also correspond to the beginning or end of the street). Each bike-agent is provided with a local planner that, given the chosen destination and the current agent location, on the basis of a local strategy, and possibly previous experience, chooses the best street-segment to take next[6] so that the bike-agent moves to the next place, i.e. the other end-point of the selected street-segment. This procedure continues until the bike-agent reaches the destination, unless a relevant update is received from the destination station. In fact, each station repeatedly broadcasts the number of its free parking slots to all those bike-agents which are heading to the station's location and which are sufficiently close to such a place. Upon receiving the information on the free slots, a bike-agent (planner) decides whether to continue the trip to such a destination station (i.e. the risk to find no parking place when arriving is acceptable) or to head to one of a set of alternative stations. The planner acquires knowledge about these alternative stations by first broadcasting to all convenient stations a request about their state and then taking into consideration a limited number of answers from such convenient stations. Typically, a station is convenient if it is close to the original destination station and has more free slots than the latter.

In the following, we assume CARMA provides the user with the built in standard types like `Int`, `Bool`, etc. as well as standard type definition mechanisms like

- set notation { } and enumeration types (e.g. `{a,b,c,d}: EnumSet` or `{(a,b),(c,d)}: EnumRelation on ...`), set-theoretic operations, e.g. Cartesian product `(*,**,*** ...)`, test on singleton and subset `singleton`, `subset_of`, etc.
- parametric lists `[*]` (e.g. lists of pairs integer-boolean `[(Int,Bool)]`).

We assume a function `RANDOM(dist)` is provided by CARMA such that for each type `T`, `RANDOM(distr)(T)` returns an element of `T` randomly chosen according to ditribution `distr`. We furthermore assume the parametric type CS($*$) (and TS($*$)) of closure spaces (topological spaces) is provided to the user as well as the parametric type QDCS($*$) of quasi-discrete closure spaces; for each finite set (of points) `V`, QDCS(`V`) is the collection of all quasi-discrete closure spaces based on `V`. Each specific element of QDCS(`V`) is introduced by declaring its characterizing edge relation, `E` $\subseteq$ `V` $\times$ `V` according to the following schema:

```
V  = {v1,v2, ...,vn} : EnumSet;
E  = {(v1,v2), ..., (vh,vk)} : EnumRelation on V;
VE = (V,E) : QDCS(V);
```

We assume that by processing the above declarations CARMA provides the user with a closure function `Closure(V,E)` such that, for each subset `A` of `V`, `Closure(V,E)(A)` returns the closure of `A` according to the standard definition of closure based on binary relations (see Section 2.2). This easily extends to edge labelled graphs as joins, as in the following example:

```
V  = {v1,v2, ...,vn} : EnumSet;
L  = {l1,l2, ...,ln} : EnumSet;
```

[6] One way to think of the planner is a smartphone application.

```
E_l1  = {(v1,v2), ..., (vh,vk)} : EnumRelation on V;
E_l2  =  ...  : EnumRelation on V;
...
E_ln  =  ...  : EnumRelation on V;
VE = {(V,E_l1), (V,E_l2), ... (V,E_ln)} : Join in QDCS(V);
with
TAKE(label:L,v:Point)=
let c = case label of l1:: Closure(V,E_l1)({v})\{v} ...  ln::Closure(V,E_ln)({v})\{v}
in if singleton(c) then element(c) else error;
/* Note that for simplicity the first argument of TAKE is a label and not a set of labels.
```

Graphs with tuple-labels can also be dealt with, as in the following example:

```
V  = {v1,v2, ...,vn} : EnumSet;
L_1  = {l11,l12, ...,l1n_1} : EnumSet;
L_2  = {l21,l22, ...,l2n_2} : EnumSet;
...
L_h  = {lh1,lh2, ...,lhn_h} : EnumSet;
L = (L_1, ..., L_h);
E_l11  = {(v1,v2), ..., (vh,vk)} : EnumRelation on V;
E_l21  =  ...  : EnumRelation on V;
...
E_lhnh =  ...  : EnumRelation on V;
VE = {(V,E_l11), (V,E_l2), ... (V,E_lhn_h)} : Join in QDCS(V);
with TAKE ...
```

The CARMAformalization of the example is given below[7]. The relevant type definitions are given below. The `CityMap` is an edge-labelled graph with labels in `StreetNames`, i.e. a `Join in QDCS(Places)`.

```
/*Constants:

Places  = { ... } : EnumSet;

StationPlaces = { ... } : EnumSet subset_of Places;

StreetNames  = {nm_1,nm_2,...,nm_k} : EnumSet;

StreetSegms_1  = { ... } : EnumRelation on Places;

StreetSegms_2  = { ... } : EnumRelation on Places;
...
StreetSegms_k  = { ... } : EnumRelation on Places;

CityMap = {(Places,StreetSegms_1),
           (Places,StreetSegms_2),
             ...
           (Places,StreetSegms_k)} : Join in QDCS(Places);
with
TAKE(label:StreetNames,p:Places)=
let c = case label of
       nm_1:: Closure(Places,StreetSegms_1)({p})\{p}
       ...
       nm_k:: Closure(Places,StreetSegms_k)({p})\{p}
in if singleton(c) then element(c) else error;

min_slots : Int; /* used by process Planner  as threshold for considering a Station risky
```

[7]It is worth noting that the model we provide here is quite a detailed one; probably too much detailed for the model to be analyzed by automatic tools and maybe more appropriate for execution. On the other hand, in this report, we abstract from all those issues which are not relevant for the purpose of the report like all aspcts related to time and probability, e,g the rate function definition.

```
updatereq : Nil;  /* used by process Planner for requesting status reports from Stations
capacity : Int; /* The max number of slots of a Station
```

We assume that, by construction:

- for each `p` in `Places` and `nm_j,nm_h` in `StreetNames` `TAKE(nm_j,p)` has at most one element and, for `j` $\neq$ `h`, `TAKE(nm_j,p)` and `TAKE(nm_h,p)` do not share any element. So edges are actually functions rather than relations and there is no place p' which can be reached from another place p via two segments (with different names) departing from p and both ending in p';
- the `name` attribute uniquely identifies a component;
- different station components reside at different `Places`.

The CARMA code for the bike-agent component is given below. The store consists of the following attributes:

`name:ComponentName` the name of the component. We assume each component is uniquely identified by its name;

`posn:Places` the place where the component is currently located;

`count:Int` a counter for fixing the max number of status reports from convenient stations;

`alternatives:[(Places,Int)]` a list of status reports from convenient stations; each status report is a pair where the first component is the place of the station and the second is the number of free slots in the station (at the time the report has been sent).

The behaviour of the component is composed of two proceses, namely `Biker` and `Planner` running in parallel:

```
/* Bike-Agent Component:

((Biker | Planner),
 gamma_ba{name:ComponentName,posn:Places,count:Int,alternatives:[(Places,Int)]})
```

The `Biker` process is shown below; a short description follows:

```
Biker =
choose_destination*[False]<_>
        {destination <-- RANDOM(Uniform)(StationPlaces \ {this.posn})}.
BP_destination[name = this.name]<destination>{}.
BS_pick[posn = this.posn](_){}.
Travelling

Travelling =
(PB_move[name = this.name](nextstreet){posn <-- TAKE(nextstreet,posn)}.
   (([posn!=destination] Travelling)
     +
    ([posn=destination] BS_release[posn = this.posn](_){}.Biker)
   )
)
+
(SB_free_slots*[True](free_slots).
  BP_free_slots[name = this.name]<free_slots>.
  PB_alternative[name = this.name](alternative_destination)
                                {destination <-- alternative_destination}.
  BP_destination[name = this.name]<destination>{}
  Travelling
)
```

A `Biker` process can be in one of two states:

`Biker` This is the initial state and the state in which the bike-agent starts a (new) journey. The `Biker` selects the destination by means of *spontaneous action* `choose_destination` and uses function `RANDOM`; it then communicates the destination to its `Planner` (i.e. the `Planner` in the same component, identified by attribute `name`) by action `PB_destination`; then, the `Biker` interacts with the station in its current location, by picking a bike; state `Biker` is left and state `Travelling` is entered at this point. It is assumed that each bike-agent component is initially located in a `Place` where a station is located as well. Recall that input actions are blocking.

`Travelling` While in state `Travelling` the process can receive (from its `Planner`, via action `PB_move`) the name of the next street to take, or (from any station, via action `SB_free_slots`) the number of remaining free slots. As we will see, not all stations will send the number of their free slots to all bike-agents: each station will broadcast the information only to those bike-agents heading to that station. The number of free slots is sent to the local `Planner`—via action `BP_free_slots`—which returns a (possibly different) destination and the `Travelling` cycle restarts (the additional communication with the `Planner`, via action `BP_destination` is there for symmetry and readability). When receiving the next street name, the process takes such a street (recall that we assume unicity of outgoing street names) and checks whether the destination has been reached. If so, the bike is released via an appropriate interaction with the station in the current place and a new journey can start. Otherwise the journey continues.

The `Planner` process is shown below.

```
Planner =
 (BP_destination[name = this.name](BP_dest).
 PB_move[name = this.name]<BestStreet(MyMap,this.posn,BP_dest)>{}.
 Planner)
 +
 (BP_free_slots[name = this.name AND free_slots >= min_slots](free_slots).
 PB_alternative[name = this.name]<BP_dest>{}.
 Planner)
 +
 (BP_free_slots[name = this.name AND free_slots < min_slots](free_slots).
 PS_updates[convenient(posn,this.posn,BP_dest,st_free_slots,free_slots...)]<update_req>
                  {count <-- max_alternatives, alternatives <--[ ]}.
 (Updating | Manage_Alternative_Destination | Timeout)
 )

Updating =
timeout[name=this.name]<set>{}.Upd

Upd=
(SP_updates[convenient(posn,this.posn,BP_dest,st_free_slots,free_slots...) AND count>0]
       (posn,free_slots) {count <-- count-1, alternatives <-- [(posn,free_slots)|alternatives]}.
 Upd)
 +
 (SP_updates[convenient(posn,this.posn,BP_dest,st_free_slots, free_slots...) AND count=0]
       (posn,free_slots){}.
 UM_syn[name = this.name]<alternatives>.
 nil)
 +
 (timeout[name=this.name](_){}.
 UM_syn[name = this.name]<alternatives>.
 nil)
```

```
Manage_Alternative_Destination =
 UM_syn[name = this.name](alternatives).
 PB_alternative[name = this.name]
     <BestAlternativeDestination(this.posn,BP_dest,alternatives,...)>{}.
 Planner)
```

It is worth noting that the choice for the next street is made by means of function `BestStreet` applied to the current position, the current destination and the *local view* `MyMap` of the city-map. We do not deal with the details of `BestStreet` or `MyMap`. We only want to point out that, for instance, `MyMap` could be a sub-graph of `CityMap`. Another possibility could be that if `CityMap` is a graph labelled by *(street-name, segment-length)* then `MyMap` could be a graph labelled by *(street-name, segment-time)*, where *segment-time* would be the average time for the bike-agent to cover the street segment and its value could be learned/adjusted by the `Planner` on the basis of experience.

In case the number of free slots is risky, i.e. is less than constant `min_slots`, the `Planner` sends a request for an update on their status to all stations that are considered `convenient` to be taken into consideration as alternative destination. The update process consists of getting status information from (at most) `count` statations; termination is guarateed by means of a timer, the details of which are not shown in the specification. We do not consider the details of the predicate `convenient`; we only point out that a possible definition of the predicate could be one which implies:

`close(posn,this.posn):` the potential alternative station is close to the (the current position of the) bike-agent, so that it should not take too long for it to get there and consequently less likely to find it full upon arrival;

`close(posn,BP_dest):` the potential alternative station is close to the destination; obvious;

`st_free_slots > free_slots:` the number of free slots of the potential alternative station must be larger than those of the current destination station, otherwise it is not worth changing destination.

In the description above we have used predicate `close(p1,p2)` which presumes the existence of a notion of distance. We shall briefly discuss the issue at the end of this section. We first show the definition of the station component, which is straightforward.

```
/* Station Component:

((Bike_Mgt | (Promotion | StatusReport)),
     gamma_s{st_free_slots:Int, posn: Places})

Bike_Mgt =
(([st_free_slots < capacity]BS_pick[posn = this.posn]<_>{st_free_slots <-- st_free_slots+1}.
 Bike_Mgt
 )
 +
 ((st_free_slots > 0]BS_release[posn = this.posn]<_>{st_free_slots <-- st_free_slots-1}.
 Bike_Mgt
 )
)

Promotion =
SB_free_slots*[destination = this.posn AND close(posn,this.posn)]<this.st_free_slots>{}.
Promotion

StatusReport =
PS_updates[True](x){}.SP_updates[True]<(this.posn,this.st_free_slots)>.
StatusReport
```

We close this section with some considerations on distances and possible choices for their representation as well as related refinements of the models. We have been using predicate `close(p1,p2)` which intuitively calls for a notion of distance. One way for defining a distance function is to refine `CityMap` so that each edge label is no longer just a `StreetName` but a pair in `(StreetNames,Distance)` where `Distance` could be `Int` or `Float` and the intended meaning of a label `(nm_h,d)` labelling a street-segment (i.e. edge) is of course that the length of that particular segment of street named `nm_h` is `d`. A distance function `dist(p1,p2)` can then be defined directly on the graph as, e.g. the length of the shortest path from place `p1` to place `p2`. Note that in this case function `TAKE(label:StreetNames,j:Int,p:Places)` should be used—with `j` =1—instead of `TAKE(label:StreetNames,p:Places)`, as discussed at the end of Section 2.2.4.

An alternative option, when appropriate, could be to consider the `CityMap` as laying on the Euclidean bi-dimensional space, to be represented by `TS(Floating)` with standard Euclidean metric `E_dist` together with a mapping `cordinates: Places` $\rightarrow$ `(Floating,Floating)`, to be defined in the model, which associates each element of `Places` with a distinct pair of Cartesian coordinates `(x,y)` so that the distance between `p1` and `p2` in `Places` is given by `E_dist(coordinates(p1),coordinates(p2))`.

4 Conclusions

In these notes we provided some pragmatic considerations, based on notions of Abstract Data Types, on how space information can be conveniently incorporated in CARMA. It is proposed to model space as an abstract data type, to be used for the values of space attributes. A range of models of space have been presented, all defined as instantiations, refinements and enrichments of a generic class of Closure Spaces.

We close by mentioning that the approach is compatible with the Spatial Logic for Closure Spaces (SLCS) presented in [2] and could bring to very sophisticated addressing mechanisms based on properties of space. For instance, the following

$$\alpha^*[\mathsf{posn} : \phi_1 \mathcal{S} \phi_2]\langle v \rangle\{\}$$

by means of which a process performs a broadcast action α sending value v, to all those components whose position posn satisfies ϕ_1 and are *surrounded* by components satisfying ϕ_2 (and which are willing/able to receive it). For instance, ϕ_1 could be a spatial property satisfied only by *safe* points in space whereas ϕ_2 could characterize *dangerous* points.

Finally, it could be interesting to investigate on the possibility of extending the present proposal with the use of affine geometry [1].

References

[1] Luca Cardelli and Philippa Gardner. Processes in space. *Theor. Comput. Sci.*, 431:40–55, 2012.

[2] V. Ciancia, D. Latella, M. Loreti, and M. Massink. Specifying and Verifying Properties of Space. In J. Diaz, I. Lanese, and D. Sangiorgi, editors, *Theoretical Computer Science (TCS 2014)*, volume 8705 of *LNCS*, pages 222–235. Springer, 2014. ISBN: 978-3-662-44601-0 (print), 978-3-662-44602-7 (online), ISSN: 0302-9743, DOI: 10.1007/978-3-662-44602-7_18.

[3] V. Ciancia, D. Latella, M. Loreti, and M. Massink. Specifying and Verifying Properties of Space - Extendend version. Technical Report TR-QC-06-2014, QUANTICOL, 2014.

[4] V. Galpin, L. Bortolussi, Ciancia V., A. Clark, R. De Nicola, C. Feng, S. Gilmore, N. Gast, J. Hillston, A Lluch-Lafuente, M. Loreti, M. Massink, L. Nenzi, D. Reijsbergen, V. Senni, F. Tiezzi, M. Tribastone, and M. Tschaikowski. A preliminary investigation of capturing spatial information for CAS. Technical Report Deliverable D2.1, QUANTICOL, 2014.

[5] A. Galton. A generalized topological view of motion in discrete space. *Theor. Comput. Sci.*, 305(1-3):111–134, 2003.

A Some Properties of Closure Spaces

The following are some useful notions and results concerning Closure Spaces [5, 2].

Definition 7. *Let $(X, \mathcal{C})$ be a closure space, $x \in X$ and $A \subseteq X$. Then:*

- *the* interior *$\mathcal{I}(A)$ of A is the set $\overline{\mathcal{C}(\overline{A})}$;*
- *A is a neighbourhood of x iff $x \in \mathcal{I}(A)$;*
- *A is* closed *if $A = \mathcal{C}(A)$ and it is* open *if $A = \mathcal{I}(A)$.* □

Lemma 1. *Let $(X, \mathcal{C})$ be a closure space and $A, B \subseteq X$. Then:*

1. *A is open iff $\overline{A}$ is closed;*
2. *$A \subseteq B$ implies $\mathcal{C}(A) \subseteq \mathcal{C}(B)$ and $\mathcal{I}(A) \subseteq \mathcal{I}(B)$*
3. *Finite intersections and arbitrary unions of open sets are open.* □

Definition 8. *Let $(X, \mathcal{C})$ be a closure space and $A \subseteq X$. Then:*

- *the* boundary *of A is $\mathcal{B}(A) = \mathcal{C}(A) \setminus \mathcal{I}(A)$;*
- *the* interior boundary *of A is $\mathcal{B}^-(A) = A \setminus \mathcal{I}(A)$;*
- *the* closure boundary *of A is $\mathcal{B}^+(A) = \mathcal{C}(A) \setminus A$*

where $\setminus$ denotes set-theoretic difference.

Proposition 2. *The following equations hold:*

$$\mathcal{B}(A) = \mathcal{B}^+(A) \cup \mathcal{B}^-(A) \quad (1)$$
$$\mathcal{B}^+(A) = \cap \mathcal{B}^-(A) = \emptyset \quad (2)$$
$$\mathcal{B}(A) = \mathcal{B}(\overline{A}) \quad (3)$$
$$\mathcal{B}^+(A) = \mathcal{B}^-(\overline{A}) \quad (4)$$
$$\mathcal{B}^+(A) = \mathcal{B}(A) \cap \overline{A} \quad (5)$$
$$\mathcal{B}^-(A) = \mathcal{B}(A) \cap A \quad (6)$$
$$\mathcal{B}(A) = \mathcal{C}(A) \cap \mathcal{C}(\overline{A}) \quad (7)$$

□