QUANTICOL

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

http://www.quanticol.eu

D4.3

CaSL at work

Revision: 1.0

Author(s): Vashti Galpin (UEDIN), Anastasis Georgoulas (UEDIN), Stephen Gilmore (UEDIN), Jane Hillston (UEDIN), Diego Latella (CNR), Michele Loreti (IMT), Mieke Massink (CNR), Natalia Zoń (UEDIN)

Due date of deliverable: Month 48 (March 2017)
Actual submission date: March 30, 2017
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

This deliverable reports on the work completed in the final reporting period on the modelling language at the centre of the QUANTICOL framework, CARMA. A major focus of the period has been on making modelling with CARMA accessible to a wide audience of potential users interested in CAS, not just those already familiar with formal modelling with process algebras. To this end we have further developed the CARMA Specification Language (CaSL) and the software tools that support it; we have developed exemplar models, some of which are reported in this deliverable, and extended the suite of tools to offer a modeller different approaches to model analysis.

In essence, the CaSL language does not formally extend the expressiveness of CARMA, but it presents a more programmatic style of modelling, which will be usable by a wider set of people. Space plays a key role in many CAS and we have revisited the support that is offered to faithfully capture the spatial aspects within a model, resulting in improved syntax to assist the modeller and a graphical front end which can be used to automatically generate the spatial aspects of models. In this document we give an account of the key features of CaSL, and present a full account of the language in an appendix. We also discuss a few of the models that have been developed alongside the language development. These served to refine our ideas on how best to support spatial modelling as well as testing the implementation in the Eclipse plug-in supporting CaSL. One of this set of models is completely outside the domain of smart cities, which has been the main focus of our case studies, in order to demonstrate that the modelling pathway that we have developed could be suitable for a wide class of applications beyond smart cities.

Of course, to be practically useful a modelling language must be implemented in a robust set of software tools to allow the modeller to construct and analyse the model with confidence. In Section 3 we give an account of the software tools, whilst in Section 4 we describe the design workflow and analysis pathway that is supported in the tools. This takes into account the different phases that a model goes through, from initial design, elaboration, parameterisation and then use as a tool to investigate the behaviour of the system under study. The modeller needs different support at each of these stages and we have sought to provide what is appropriate for each stage, as far as is feasible within the limited time and resource of the project. Building within Eclipse has allowed us to provide many “hidden" features which nevertheless greatly enhance the support for the modeller. These internal checks seek to ensure that CaSL models are free from the type of minor error that can be frustrating and time-consuming during model development. However, once a modeller is fully confident of their model, a graphical user interface can become cumbersome and inconvenient. Thus we also provide a command line interface to support efficient exploitation of models under different experimental frames. Moreover, the results of model analysis are automatically enhanced with metadata to assist with their interpretation and reproducibility.

The deliverable concludes with a demonstration of the analysis of CaSL models on two of the scenarios from our smart city case studies. Specifically we consider a mesoscale model of buses within a city, particularly paying attention to the congestion that occurs when multiple routes share the same bus stopes, and issues related to regulatory compliance and appropriate spacing on frequent bus services. In the second example, we consider the key issue related to user satisfaction within urban bike sharing systems — whether a user will find a bike or a slot at a convenient location when they want one.

In addition to this document, we also deliver the software tool suite for CARMA which is available at `https://quanticol.github.io`.

Contents

1 Introduction

One of the main aims of WP4 was the design of a programming/specification language to be used to model, program and analyse collective adaptive systems (CAS). Since the beginning, in the the design of this language, we considered it important that the language offers the possibility of integrating behavioural description and knowledge management and provides specific abstractions or linguistic primitives for key concepts like knowledge, behaviour, aggregation, and interactions. Moreover, it was also clear that the language should provide different kinds of interaction patterns to take into account the different communication and synchronisation capabilities of CAS and to support the multilayer structure of collective systems. Another important aspect is that the language is mathematically founded to enable both qualitative and quantitative analysis in a scalable manner.

In Deliverable D4.1 we reported on 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.

Starting from these requirements we designed a new language to support the specification and analysis of CAS, with the particular objective of supporting quantitive evaluation and verification. This language was named CARMA, Collective Adaptive Resource-sharing Markovian Agents.

CARMA combines the lessons we learnt from other stochastic process algebras such as PEPA [16], EMPA [2], MTIPP [15] and MoDEST [4], with those learnt from languages specifically designed to model CAS, such as SCEL [8], the AbC calculus [1], PALOMA [9], and the Attributed Pi calculus [18], which feature attribute-based communication and explicit representation of locations.

A distinctive contribution of the language CARMA is the rich set of communication primitives that are offered. This new language supports both unicast and broadcast communication. This richness is important to enable 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 rich 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 feature of CARMA is the explicit representation of the environment in which processes interact, allowing rapid testing of a system under different open world scenarios. The environment in CARMA models can evolve at runtime, due to the feedback from the system, and it further modulates the interaction between components, by shaping rates and interaction probabilities.

Deliverable D4.2 presented the description of CARMA and its operational semantics together with a set of tools supporting analysis of CARMA models. In this toolset was included a Java library for simulating CARMA models and an Eclipse plug-in for supporting specification and analysis of CAS in CARMA. In this plug-in, CARMA systems are specified 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. CaSL was not introduced 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.

Progress in the reporting period. In the last reporting period there have been no significant changes to the syntax of CARMA, but the semantics of CARMA have been revised and simplified. The new semantics have been presented in [21] and reported in the Internal Report 4.2.

Much effort has been applied to improve the *usability* of CaSL. The syntax of the language has been simplified and new features included to ease the use of the language by users who are not familiar with formal languages. CaSL has been extended to include definitions of *spatial models* in which components operate (a first description of this new feature has been presented in Internal Report 4.2). A number of case studies have been undertaken to explore the expressiveness of the language and inform future development of both CARMA and CaSL, and their software tools.

Usability of the CARMA Eclipse Plug-in has also been substantially improved. New features and views have been included to improve the analysis workflow of CaSL models. The CARMA tool suite has been also extended with a graphical user interface to allow a modeller to work graphically when specifying the spatial aspects of a model and with a *command line interface* that allows users to perform some common tasks related to CaSL models through a simple, lightweight interface that is also amenable to scripting, thus providing programmatic access to some of the CARMA tools. Moreover, to provide further analysis options for CaSL models beyond what the plug-in offers, we have developed an interface to other tools developed in WP5 like the MultiVeStA platform, allowing CaSL users access to the statistical analysis capabilities offered by the software. Finally, to ease the access to CaSL and its tool, a specific web site has been instantiated where users can access tool documentation and examples. Moreover, a bug reporting system is also now available.

To guide less experienced users between the different types of modelling that can be done with CaSL, we defined an analysis pathway to guide the user through this process, directing them first to the simple and inexpensive analysis which must be done on models, then leading them to the more computationally expensive analyses which need only be done if initial inexpensive checks have been passed.

Structure of the deliverable. In this deliverable we will first present a detailed description of CaSL, a specification language for CARMA models, and its modelling environment. CaSL enriches CARMA with additional syntactic elements and type-checking to aid the correct construction of models. A first description of CaSL has been presented in Deliverable D4.2. However, the focus of D4.2 was more on CARMA and many details were omitted.

The rest of this deliverable is organised as follows. In Section 2 we present CaSL while in Section 3 all the tools developed around CaSL are described. Moreover, in the same section, we suggest how other techniques proposed in the project could be applied to CaSL in the future. In Section 4 we discuss an analysis pathway that will guide less experienced users through the process of designing CAS with CARMA and CaSL. In Section 5 we show two examples of applications of techniques developed in QUANTICOL.

2 CaSL: CARMA Specification Language

CARMA has been designed with the goal of identifying basic interaction mechanisms that are specific to CAS. For this reason, CARMA is in a certain sense *minimal* and abstracts from many details, such as the precise syntax of *expressions* or *values*, that are definitively needed when a *concrete* specification has to be provided. For this reason CaSL, the CARMA specification language, has been introduced to ease the task of modelling for users who are unfamiliar with process algebra and similar formal notations.

In this section we first recall basic features of CARMA, then a gentle introduction of CaSL constructs is provided. A first version of CaSL has been already presented in Deliverable 4.2. In this report we just recall basic language constructs that are used later in the document. More details are provided for new features (like the definition of space models) that have been recently included in CaSL. For the sake of completeness a detailed description of CaSL is provided in Appendix A.

2.1 CARMA in a nutshell

Recall that CARMA is a new stochastic process algebra for the representation of systems developed in the CAS paradigm [5]. The language offers a rich set of communication primitives, and exploits *attributes*, captured in a *store* associated with each component, to enable attribute-based communication. For example, for many CAS systems the location is likely to be one of the attributes. Thus in CARMA it is straightforward to model systems in which, for example, there is limited scope of communication, or interaction is restricted to co-located components, or where there is spatial heterogeneity in the behaviour of agents.

A CARMA system consists of a *collective* operating in an *environment*. The collective is a multiset of components that models the behaviour of a system; it is used to describe a group of interacting *agents*. The environment models all those aspects which are intrinsic to the context where the agents are operating. The environment mediates agent interactions. This is one of the key features of CARMA. The environment is not a centralised controller but rather something more pervasive and diffusive — the physical context of the real system — which is abstracted within the model to be an entity which exercises influence and imposes constraints on the different agents 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.

A CARMA component captures an *agent* operating in the system. It consists of a process, that describes the agent's behaviour, and of a store, that models its *knowledge*. A store is a function which maps *attribute names* to *basic values*.

Processes located within a CARMA component interact with other components via a rich set of communication primitives. Specifically, CARMA supports both unicast and broadcast communication, and provides locally synchronous, but globally asynchronous communication. Distinct predicates (boolean expressions over attributes), associated with senders and potential receivers are used to filter possible interactions. Thus, a component can receive a message only when its store satisfies the target predicate. Similarly, a receiver also uses a *predicate* to identify accepted sources. The execution of communicating actions takes time, which is assumed to be an exponentially distributed random variable whose parameter is determined by the environment.

2.2 A gentle introduction to CaSL

To simplify the use for system designers we have introduced the specification language CaSL that, while incorporating all the features of CARMA, provides rich syntactic constructs that are inspired by *main stream* programming languages.

Data types. In stochastic process algebras data is typically abstracted away. 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 processes. In the context of CAS, where we want to support attribute-based communication to reflect the flexible and dynamic interactions that occur in such systems, data cannot be abstracted.

For this reason CaSL supports four kind of basic types: booleans, integers, real values, *spatial locations*. Where the latter is used to refer to *locations* where agents operate. Moreover, to model complex structures, in CaSL *custom types* can be declared: *enumerations* and *records*. The former is a data type consisting of a set of *named values* that behave as constants in the language while the latter consist of a sequence of *typed field*.

CaSL also supports *collections*. These are aggregations of homogeneous data and can be either *sets* or *lists*. A *set* is, as usual, a collection that does not contain duplicated elements. While a *list* consists of a sequence of elements of the same type.

Expressions. CaSL is equipped with a rich set of expressions that combine expressive power with a compact representation. A detailed description of CaSL expressions is available in Appendix A.2 while the full syntax is reported in Appendix B.

Beside standard arithmetic/logical operators, CaSL expressions include also operators that can operate on collections. These are `exist`, `forall` and `map`[1]. Function `exist` can be used to check whether there exists an element in a collection that satisfies a given predicate. The function `exist` takes two parameters: one collection and a boolean expression. The latter may contain the special symbol `@`. This is used as a placeholder replaced by the elements in the collection when the predicate is evaluated. The function application `exist( e1 , e2 )` evaluates to `true` if there exists in `e1` an element `x` such that `e2[x/@]` is true. For instance, if `e1` is a collection of integers, `exist( e1 , @>5 )` is true if and only if there exists an element in `e1` that is greater than `5`. The function `forall` is similar, `forall( e1 , e2 )` is true if and only if all the elements in `e1` satisfy the predicate `e2`. It is possible to select all the elements satisfying a given predicate by using the function `filter`. If `e1` is a collection and `e2` a boolean expression containing the placeholder `@`, `filter( e1 , e2 )` is the collection that contains only the elements satisfying `e2`. For instance, `filter( {: 2 , 4 , 6 :} , @<3 )` will return the set `{: 2 :}`. Sometimes it is also useful to manage elements in a collection in an aggregated way. For this reason, the function `map` allows the creation of a new collection that is obtained from another one by applying a given function. This function is defined as an expression that contains the placeholder `@`. The expression `map( [: 1 , 2 , 3 :] , pow( @ , 2 ))` is equivalent to `[: 1 , 4 , 9 :]`. To improve readability of expressions, all the functions on collections can be used in *infixed* form. This means, for instance that `map( e1 , e2 )` can be expressed as `e1.map( e2 )`.

To model random behaviour, CaSL expressions provide different mechanisms for sampling random values. A first mechanism to include random values is to use the expression `RND`. When this expression is evaluated, this term is replaced with a value that is randomly selected in the interval $[0,1)$. To sample values according to a *normal distribution*, we use the expression `NORMAL( e1 , e2 )`. In this case, the next value is randomly selected according to a distribution with mean `e1` and variance `e2`.

To select values from a collection, the function `select( e1 , e2 )` can be used. There, `e1` is a collection, while `e2` is an expression (containing the placeholder `@`) that is used to compute the probability to select each element in `e2`. For instance, in the expression `select( {: 1, 2, 3, 4, 5 :} , @+1 )` a value $i \in \{1,2,3,4,5\}$ is selected with probability $\frac{i+1}{35}$. While in `select( {: 1, 2, 3, 4, 5 :} , 1 )` each value is selected with the same probability ($\frac{1}{5}$). Another statement for uniform selection of elements is `U( e1 , ... , en )`. This is used to uniformly select one of the values `e1`,..., `en`.

Constants and Functions. A CaSL model can contain *constant* and *function* declarations. A constant can be declared by using the following syntax:

```
const <name> = <exp>;
```

where, $<\text{name}>$ is the constant name while $<\text{exp}>$ is the expression defining the constant value. Constants are not explicitly typed. This is because the type of a constant is not declared but inferred directly from the assigned expression $<\text{exp}>$.

Function declaration has the following syntax:

```
fun <type> <name>( <type1> <arg1> , ... , <typen> <argn> ) <body>
```

where $<\text{name}>$ is the function name, each $<\text{arg}_i>$ is the name of parameter i of type $<\text{type}_i>$, while $<\text{type}>$ is the type of the value returned by the function. Finally, $<\text{body}>$ contains the statements used to compute the returned value. Elements in $<\text{body}>$ are standard statements in a high-level programming language. A detailed description of of function declaration syntax is available in Appendix A.3.

Component prototype. A *component prototype* provides the general structure of a component that can be later instantiated in a CaSL 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>( <type1> <arg1> , ... , <typen> <argn> ) {
    store {
```

[1]In CaSL `[: e1,...,en :]` is a list while `{: e1,...,en :}` is a set.

```
            attrib <type_1> <name_1> = <exp_1>;
            ...
            attrib <type_m> <name_m> = <exp_m>;
        }
        behaviour {
            <pdef_1>
            ...
            <pdef_p>
        }
        init { <name_1> | ... | <name_k> }
    }
```

Each component prototype has a possibly empty list of arguments. As expected, 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 e. The special attribute `loc` is always available in any store. An appropriate value is assigned to this attribute when the component is instantiated. Block `init` is used to specify the initial behaviour of a component. It consists of a sequence of terms `pname_i` referring to processes defined in block `behaviour` or a process in the argument list. The block `behaviour` is used to define the processes that are specific to the considered component and consists of a sequence of definitions (see Appendix A.5 for more details).

Space models. Each CaSL model consists of a set of *components*, which are elements that are deployable in a physical system, and an *environment*, which imposes limitations on, and defines the general rules for, communication. This approach allows us to separate system behaviour, identified by *components*, from the specification of the context which regulates the interaction of components.

It is important to avoid hardcoding the environment inside components' stores or behaviours, as this leads to cumbersome models which can less readily be used in experiments on how components perform in different contexts. When system behaviour is clearly separated from the environment the resulting models are flexible in terms of being able to easily represent the performance of the components when subjected to different kinds of external conditions.

Following this approach, we can think of space and components as two distinct layers of the model (see Fig.1). Components reside in the top, *behaviour layer* and they can only perform their actions if the topological structure defined in the underlying *space layer* allows that. The initial focus has been on graphs at the prototypical spatial structure. The space where a system operates can be defined as a graph in which edges have labels that contain tuples of *properties*. For example we can have a road lane with attribute $buses = true$, which means that buses can travel on it.

Each space is associated with a universe — a collection of nodes along with information about their location in space. This can be, for example, a grid, along with an indexing system, or a bounded plane with a coordinate system. The `nodes` block specifies which subset of nodes from the universe is used in the model. The `connections` block contains the specification of how these nodes are connected to each other. The `areas` block allows the user to define attributes associated with subsets of nodes belonging to the space. The complete syntax of spatial models is available in Appendix A.4.

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

```
system <name> {
  space <name>(<exp_1> , ... , <exp_n>)
  collective {
    <cblock>
  }
  environment { ···
  }
}
```

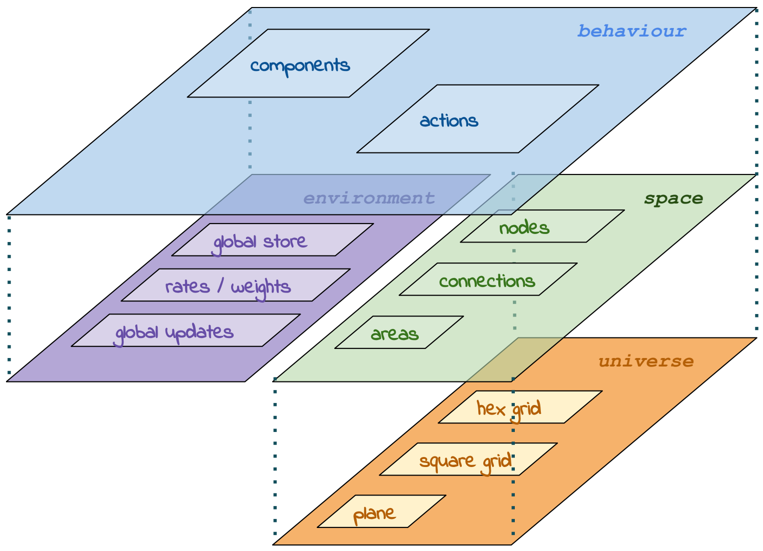


Figure 1: The layers of CASL

Space instantiation is used to define the space model where components are located. This instantiation is optional and can be omitted.

Above, $\underline{< \text{cblock} >}$ indicates a sequence of commands that are used to instantiate components. The basic command to create a new component is:

new $\underline{< \text{name} >}$($\underline{< \text{exp}_1 >}$, . . . , $\underline{< \text{exp}_n >}$)@$\underline{< \text{exp}_l >}$<$\underline{< \text{exp} >}$>

where $\underline{< \text{name} >}$ is the name of a component prototype, $\underline{< \text{exp}_i >}$ are the parameters, $\underline{< \text{exp}_l >}$ is the (optional) location where the created component is located (and that will be assigned to attribute `loc` having type `location`), and $\underline{< \text{exp} >}$ is the integer expression identifying the multiplicity (i.e. the number of copies) of the created component.

However, in a system a large number of collectives can occur. For this reason, following the same approach used to create spatial models, we can use *for-loops* and *selection* constructs for instantiating multiple components.

The syntax of a block `environment` is the following:

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

The block `store` defines the *global store* and has the same syntax as the similar block already considered in the component prototypes. Blocks `prob` and `weight` are used to compute the probability to receive a message, while `rate` is used to compute the rate of an unicast/broadcast output.

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

measure $\underline{< \text{name} >}$($\underline{< \text{type}_1 >}$ $\underline{< \text{name}_1 >}$, . . . ,$\underline{< \text{type}_n >}$ $\underline{< \text{name}_n >}$) = $\underline{< \text{exp} >}$;

Above $\underline{< \text{exp} >}$ can contain specific expressions that can be used to extract data from the population of components. To count the number of components in a given state, the following term can be used. For instance,

#{ Π | $\underline{< \text{exp} >}$ }

can be used to count the number of components in the system satisfying boolean expression $\underline{< \mathsf{exp} >}$ where a process of the form Π is executed. In turn, Π is a pattern of the following form:

$$\Pi \quad ::= \quad * \quad \Big| \quad *\texttt{[}\ proc\ \texttt{]} \quad \Big| \quad comp\texttt{[}\ *\ \texttt{]} \quad \Big| \quad comp\texttt{[}\ proc\ \texttt{]}$$

To compute statistics about attribute values of components operating in the system one can use: `min{` $\underline{< \mathsf{exp} >}$ `|` $\underline{< \mathsf{exp_g} >}$ `}`, `max{` $\underline{< \mathsf{exp} >}$ `|` $\underline{< \mathsf{exp_g} >}$ `}` and `avg{` $\underline{< \mathsf{exp} >}$ `|` $\underline{< \mathsf{exp_g} >}$ `}`. These expressions are used to compute the minimum/maximum/average value of expression $\underline{< \mathsf{exp} >}$ evaluated in the store of all the components satisfying boolean expression $\underline{< \mathsf{exp_g} >}$, respectively.

2.3 CaSL at work

With the development of CaSL, we have developed numerous models relating to smart cities and smart grids, as well as models for other domains. In this section, we illustrate the use of CaSL by considering three models. Two of these are smart city-related and the third considers of model of food security, thus demonstrating the use of CARMA beyond the case studies of the project.

This section considers the three models in turn, describing the modelling scenario, providing a brief overview of the model structure in terms of components and environment, the aims of the modelling and the results obtained.

2.3.1 Ambulance deployment

Jagtenberg *et al.* [17] have proposed a real-time approach to ambulance deployment based on a heuristic. The general goal of such systems is to minimise the time it takes to respond to medical incidents by ensuring good base locations for ambulances together with a distribution of ambulances over bases that leads to fast response. Traditionally, deciding how to deploy ambulances across a region has been done statically, in the sense that once an ambulance has completed its current task, it returns to a predefined base, and moreover determining the best bases is done in advance of deployment. In the dynamic approach of Jagtenberg *et al.*, depending on the locations of the other ambulances, an ambulance that is no longer busy can be requested to go to a specific location in a set of base locations to wait for its next task, thus allowing the system to adapt to the current circumstances.

The ambulance system is modelled as a graph of locations with edges representing roads, annotated with information about how long it takes to traverse the edge. Locations may be cities, towns, road junctions or other points of interest. Each location has an incident probability, and some locations have ambulance bases or hospitals. Evaluation of the system is based on the proportion of ambulances that fail to reach an incident within a specified time period, T. This is known as the *late proportion*[2], *lp*. A lower value indicates a better system using this approach. This is used to evaluate the performance of a heuristic that determines where an ambulance should wait between incidents. The heuristic is based on a function that chooses the location that will provide the maximum increase in coverage [7] with respect to the time limit for attendance at accidents.

To explore the behaviour of the heuristic further, CARMA models were developed of the scenario [13], and were specified in CaSL. The main components of the models were a component that generated incidents, a queue component to keep track of incidents until they were handed over to an idle ambulance, ambulance components, and route components which interact with ambulance components to direct them to incidents, hospitals and bases. Two global variables are defined in the environment: one to count ambulances that arrive at incidents within the required time and one to count those ambulances that miss the deadline. From these two values, a measure that describes the evaluation metric mentioned above can be defined. Figure 2 illustrates results for a particular road network. Each combination of time limit and hospital location was simulated for 500 runs over 20 hours of simulated time.

There are three different possibilities for hospital location (shown in the three different columns of Figure 2), and the deadline T is given together with the proportion of late arrivals *lp* for each example. The shaded circles

[2]Previously called *late rate* in [17].

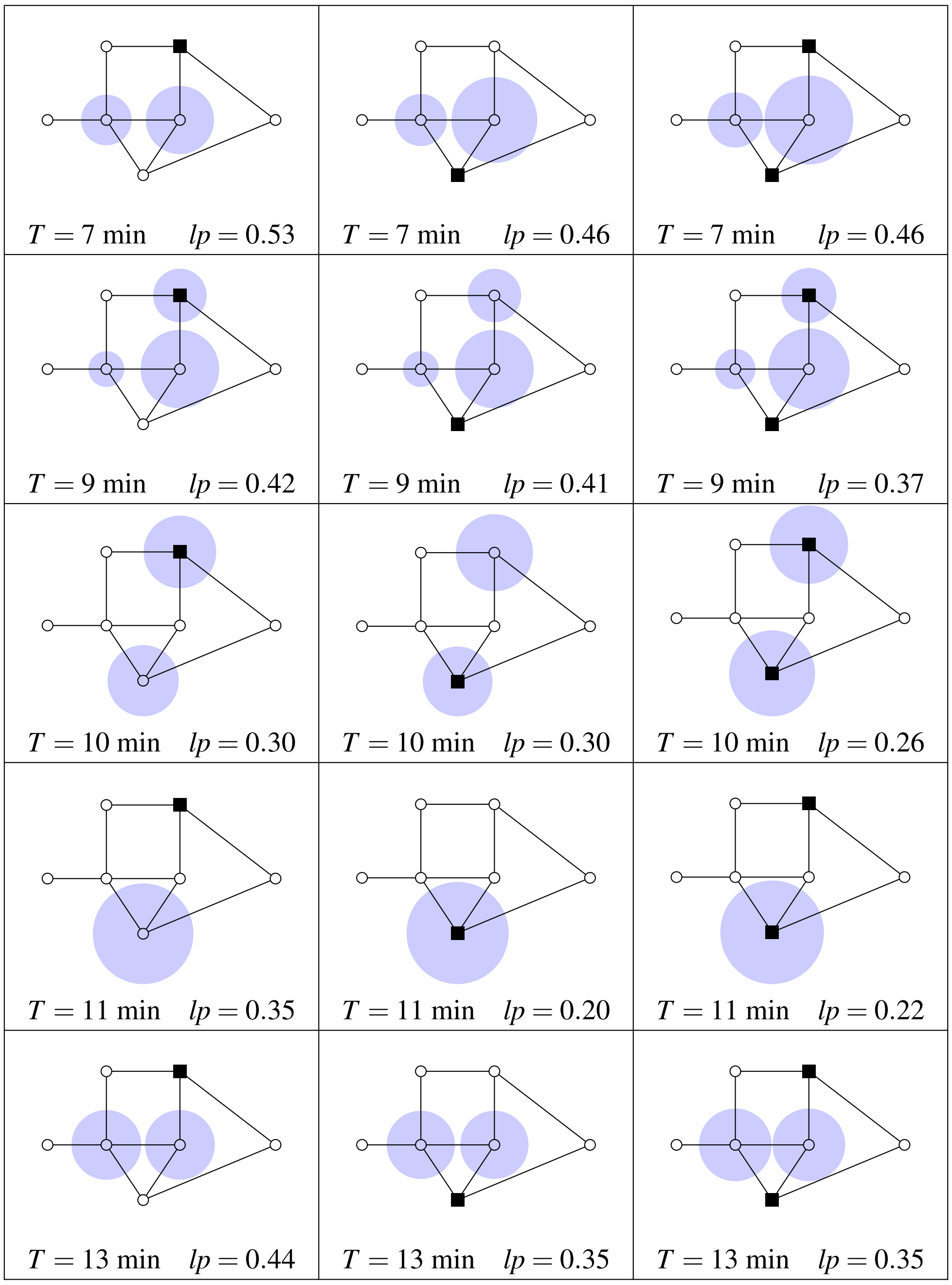


Figure 2: Idle ambulances and proportions of late arrivals for time limits and hospital locations. Black squares indicate hospital locations and shaded circles indicate locations of idle ambulances. The area of the circle represents the proportion of simulations in which idle ambulances are at a location (or on their way to that location as a base) at the time point of 1200 minutes (20 hours). T is the time limit and lp is the late proportion.

```
system EMS {
  collective {
    new Incident_Queue();
    new ClosestIdleAmbulance();
    new Ambulance(1,L1);
    new Ambulance(2,L2);
    new Ambulance(3,L3);
  }
  environment {
    store {
      attrib ontime := 0.0;
      attrib late   := 0.0;
    }
    prob   { default { return 1.0;} }
    weight { default { return 1.0;} }
    rate {
      incident*    { return 1.0/mean_time_betw_incidents;}
      pickup*      { return 1.0/12;}
      treat*       { return 1.0/12;}
      dropoff*     { return 1.0/15;}
      default      { return 100;}
    }
    update{
      timecheck*   { if (now>sender.itime+T)  { late := late + 1;}
                     else                     { ontime := ontime + 1;}}
      incident*    { new Incident_Queue_Item(sender.inum,now);}
      new_handler* { new Incident_Handler(IncidentLocation(), IncidentType(),
                                 sender.itime);}
      tobase*      { new Return_Handler(sender.anum,sender.aloc,sender.dest,
                               GetBase(...);}
      makeroute*   { new Route(sender.anum,sender.dest, sender.aloc, sender.locn,
                          NextHop(...), RouteLength(...));}
                 }
         }
}
```

Figure 3: CaSL system definition for the ambulance deployment model. `GetBase(...)`, `NextHop(...)` and `RouteLength(...)` are function calls.

indicate at which locations the idle ambulances were based, and all locations were considered as possible base locations. The area of the circle represents the proportion of simulations that idle ambulances are at a location (or on their way to that location as a base) at the time point of 1200 minutes (20 hours). Since there are three ambulances, fewer than three circles indicate that multiple ambulances are idle at a location.

The results show that the heuristic does not have monotonic behaviour since an increased time limit can lead to a choice of different base locations with a worse late proportion. The heuristic does not use the hospital location but obviously the distance from the hospital back to base will impact availability, and hence late proportion. Two hospitals at different locations can lead to a shorter average distance to potential base locations, thus ensuring ambulances become available again faster, and reducing time to get to incidents. The lowest late proportions occur when there are two hospitals at the two cities, and an ambulance goes to the closest hospital. This experiment shows how the late proportions would be affected if it was necessary to close one of the hospitals. However, for some time limits, the presence of two hospitals has little effect on the late proportion when compared with just one hospital at the second city, where for others it makes a significant difference. The fact that hospital location can affect the proportion of late arrivals suggests a role for the hospital location in the heuristic function.

This model was developed before the implementation of the space syntax of CaSL and the CARMA Graphical Plug-in. It was straightforward to map the elements of the ambulance scenario to CaSL components, and to define the metric used to evaluate the scenario as a CaSL measure. The original scenario used deterministic travel times and these can be encoded in CaSL. The details of the road network were expressed using CaSL functions and this meant that the components were independent of this information. When an ambulance was required to move between nodes, `Handler` and `Route` components were added to the collective to direct the `Ambulance` component. This meant that only the functions need to be changed to model a different road network. Furthermore, the calculation of the heuristic can also be changed by substituting a different function. The ability to add components to the collective is a significant feature of CARMA, and was very important in representing the generation of incidents and their handling as well as the route modelling. Figure 3 presents the environment of the model expressed in CARMA, and shows how the model starts with a fixed number of `Ambulance` components, an `Incident_Queue` component and a component to determine the location of the closest free ambulance to an incident. The update section of the environment illustrates how new components are generated when actions occur. For example, in the case of a `makeroute*` action, a `Route` component is created. The global variables are also updated when a `timecheck*` action happens.

The ambulance deployment example was the first major CARMA model with an explicit space element to be developed and the space aspects of the model were a time-consuming part of this work. This experience had a strong influence on the development of the space syntax in CaSL.

It would also be interesting to apply statistical model checking using MultiVeStA to the model, as this would provide a way to describe and investigate properties of the scenario. Rather than working with the metric defined in the original paper, it would now be possible to ask more complex questions about the probability of poor response times to incidents.

2.3.2 Pedestrian movement

As we have seen, CAS consist of multiple components or agents and are characterised by the fact that each component does not have a global view of the whole system but rather has local information on which to act. This emphasis on *local* versus *global* means that space can play an important role in CAS and to investigate these spatial aspects, a model of pedestrian movement was developed using CARMA and CaSL [26]. This model considers pedestrians moving over a network of paths. This could be a specific part of a city, a pedestrianised network of lanes, or paths through a large park. The defining feature of our example is that there are two groups of pedestrians (indicated by the colours red and blue) that start on opposite sides of the network who wish to traverse the paths to get to the side opposite to where they started. This scenario could arise in a city where there are two train stations on opposite sides of the central business district serving the eastern and the western suburbs of the city, and a number of people who commute from the west work close to the east station and vice versa. If there are multiple paths, it would seem that it makes sense to use some paths for one direction and other paths for the other direction.

The model considers four different path topologies as shown in Figure 4 (the $n \times m$ notation describes how many X-structures there are in the topology: n indicates the height and m the width). The CARMA model consists of pedestrian components as well as two arrival components which generate new pedestrians at each side of the network. The rate at which pedestrians move is determined by the congestion in the network, and the average time to traverse the network of paths is calculated for the two pedestrian types. This is done for three distinct scenarios: one in which only one type of pedestrian is present to give a base time for traversal without congestion from oncoming pedestrians (no-congestion scenario) and one in which there is no routing and pedestrians choose between paths depending on movement rate when there is a choice of paths (no-routing scenario). In the scenario with routing, on each side of the network, only one of the paths is available for entry to the network. For the left side of the network, it is the lower path that is available and on the right side of the network, it is the upper path. At the internal nodes, the choice is the same as in the no-routing scenario.

The results from our experiments are presented in Figure 2. An inspection of the results shows that, unsurprisingly, for any structure the best average travel times are obtained when there is no congestion in the network. As anticipated, networks with greater height have lower average travel times because they have greater capacity, due to the inclusion of additional routes (thus 2×1 results are better than 1×1 results, and 2×2 results are better than 1×2 results). Finally, we see that routing is always advantageous, especially so in the case of

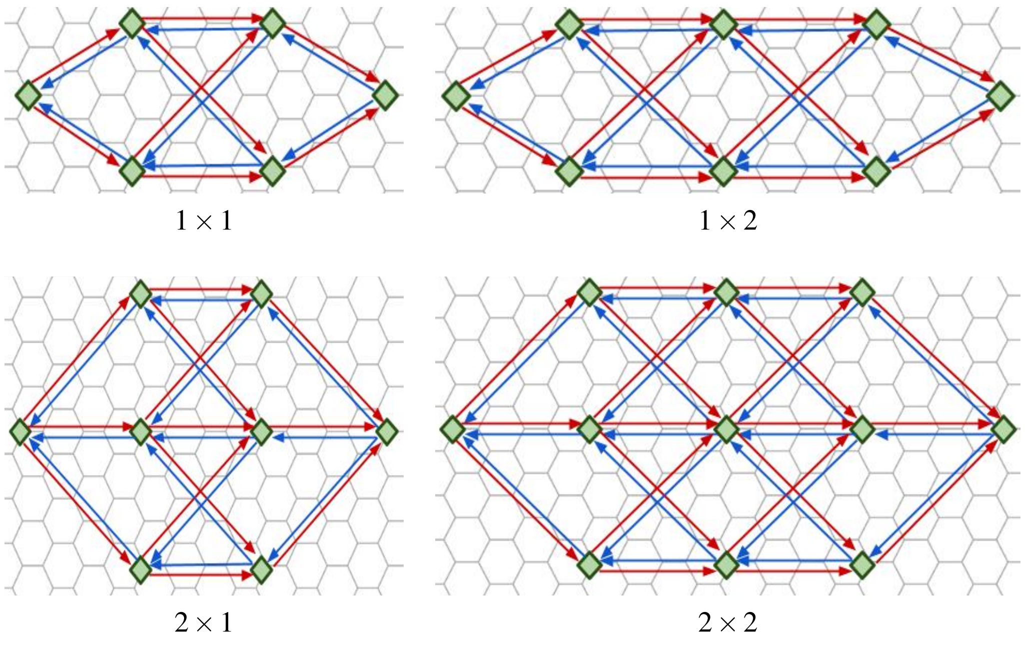


Figure 4: Four path topologies of increasing size and complexity.

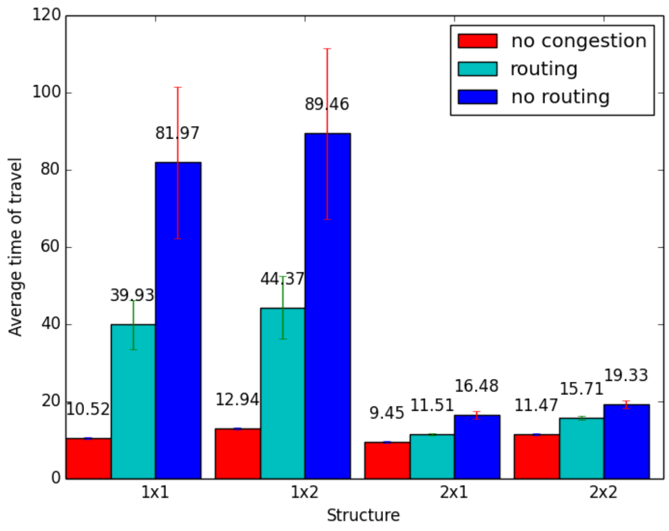


Figure 5: Average travel time results from the experiments on structure and network usage.

narrow networks where congestion in experienced most (i.e. in the 1×1 structure and the 1×2 structure).

In developing this model, there were a number of different directions that could be taken to model pedestrians, including a component for each pedestrian, or a component with counters for each segment of path and junctions between these segments. We chose the former, and used the Graphical CARMA Plug-in to develop the space aspects of the model (before the space syntax was added to CaSL). This allowed for the generation of functions and components that describe pedestrian movement. A `Pedestrian` component consists of a store with current location in the network, pedestrian type and arrival time. The behaviour of the `Pedestrian` component involves a choice between paths that are available (which are determined by CaSL functions) and a check as to whether the pedestrian's goal has been reached. The rate of movement over the paths takes into account which path is being traversed and how many pedestrians are present at the next node and travelling in

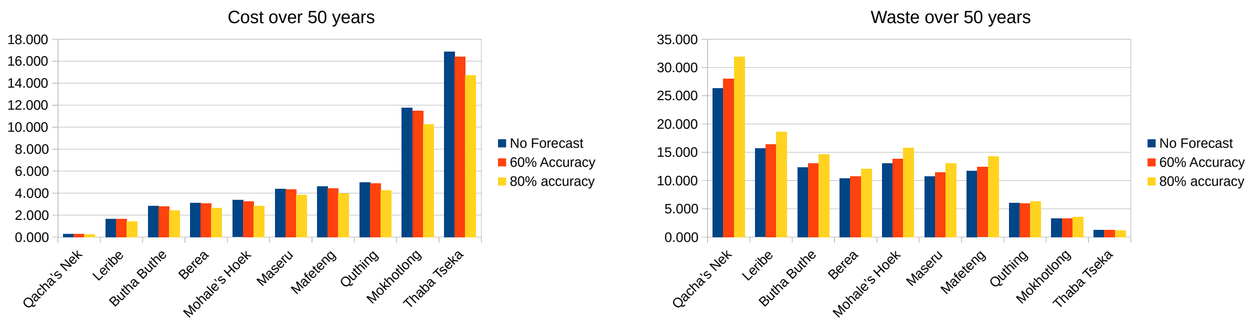


Figure 6: Cumulative cost and waste for each district in Lesotho over a fifty year period for different forecast accuracies.

the opposite direction to that of the pedestrian whose movement rate is being calculated. Each `Arrival` component generates new pedestrians at a set rate for its pedestrian type at the entry points of the network. In the global store, there are two variables for each pedestrian type, namely the number of pedestrians reaching their goal, and the total time taken by all pedestrians. These variables are updated whenever a pedestrian reaches its goal: the counter is incremented and the time taken by the pedestrian is added to the total. From these variables, a measure for the average time taken for each type of pedestrian is calculated.

This research has now being taken further [24]. In the original research, the number of pedestrians at each node was considered, but we have now considered congestion on edges (rather than nodes as described above) in determining pedestrians' choices which illustrates interestingly different behaviour where routing leads to worse outcomes that can be explained in term of the visibility of edges. We have also developed a model of an actual park with pedestrian paths, namely the Meadows in Edinburgh.

2.3.3 Food security

We have also applied CARMA in a context beyond the original case studies to evaluate the suitability of CARMA in a more arbitrary CAS setting, and here we describe how CARMA can be applied to food security agent-based models [14].

Investigating food security in the case of a changing climate is important for the developing world. Recent papers have considered agent models of farmers in sub-Saharan Africa [3, 25]. These two scenarios have been expressed in CaSL and their behaviour has been validated by comparing the output of the simulation from the CARMA Eclipse Plug-in with the data in the original papers. The models have similar structures, with components that provide weather information as well as forecast for the weather using a specified level of accuracy. The CARMA components in the models that represent the farmers have a yearly sequential cycle which involves getting weather forecasts, making planting decisions, harvesting crops and then consuming or selling the produce. For each of the original papers, the model is of a single community of farmers, and we are now developing models with spatial aspects, which will use the spatial syntax of CaSL. For the Lesotho example [25], we are developing a model that covers the eight regions of Lesotho with groups of farmers in each region. The aim of the model is to explore distribution of food between regions taking into account the mountainous terrain of Lesotho that constrains travel. Figure 6 illustrates the excess grain required to ensure hunger does not occur compared to the grain that is wasted because it cannot be stored for two years for each district of Lesotho. This suggests that with redistribution of grain, hunger may be prevented.

Although these models are still in development, they illustrate how CARMA can be used for modelling scenarios beyond the CAS related to smart city applications identified for the QUANTICOL project.

3 CARMA tool suite

3.1 CARMA Eclipse Plug-in

The CARMA Eclipse plug-in is available at `https://quanticol.github.io`. At the same site detailed installation instructions can be found together with a set of case studies that shows how CAS can be modelled and verified with the provided tool.

The CARMA Eclipse Plug-in provides a rich editor for CAS specification in CASL. In addition to syntax highlighting, the editor continuously checks the model for syntax errors and ensures that it adheres to the CaSL standard. In case of problems, a tool-tip message explains to the user what error has been encountered.

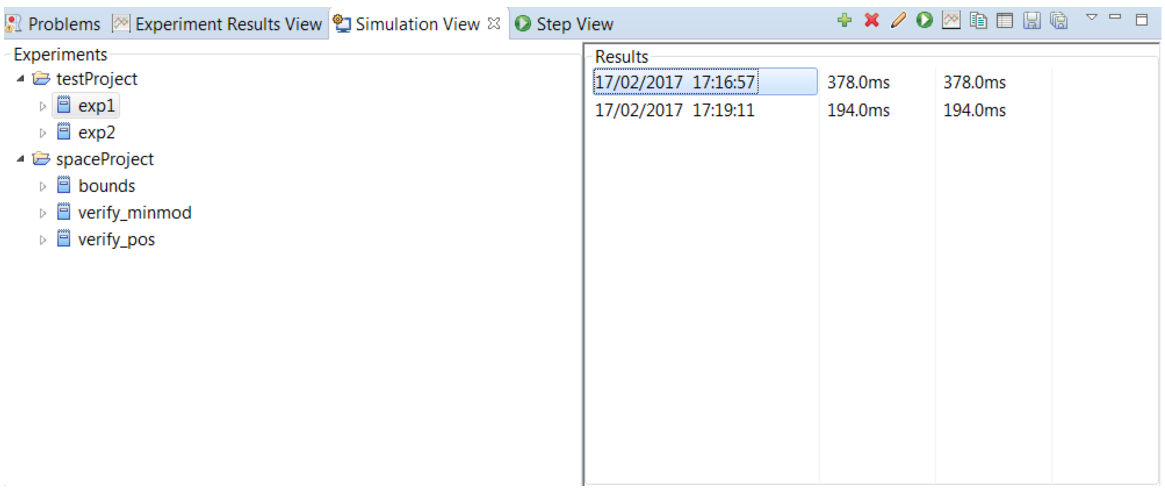
Given a CaSL specification, the CARMA Eclipse Plug-in automatically generates the Java classes needed to simulate the model. This generation procedure can be specialised to different kinds of simulators. Currently, a simple ad hoc simulator is used. The simulator provides generic classes for representing *models* to be simulated. To perform the simulation each *model* provides a collection of *activities* each of which has its own *execution rate*. The simulation environment applies a standard *kinetic Monte-Carlo* algorithm to select the next activity to be executed and to compute the execution time. The execution of an *activity* triggers an update in the simulation model and the simulation process continues until a given simulation time is reached. From a CARMA specification, these activities correspond to the *actions* that can be executed by processes located in the system components. Indeed, each such activity mimics the execution of a transition of the CARMA operational semantics. Specific *measure functions* can be passed to the simulation environment to collect simulation data at given intervals. To perform statistical analysis of collected data the *Statistics package* of *Apache Commons Math Library* is used[3].

To access the simulation features, a modeller can make use of the *Simulation View* (Figure 7).

This offers an overview of the experiments previously defined for the different models, as well as their results. Experimental results can be stored as CSV files, while experiments themselves can also be exported in a text file for future reference. The user also has the option of defining a new experiment; for this, they are asked to choose a model and specify the experimental configuration, such as the final time of the simulation, the number of replications and the measures to track during the experiment. This is done graphically (Figure 8), with the plug-in validating the user's input to each field and ensuring that all required information is given.

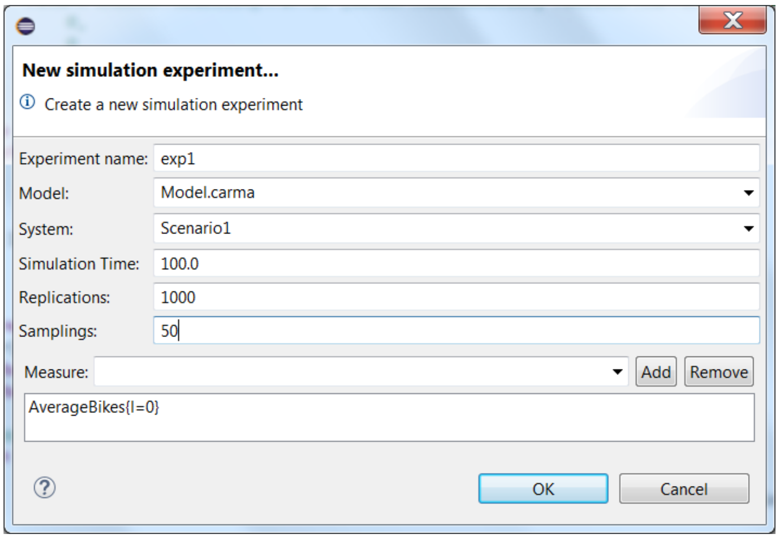
The results are reported within the *Experiment Results View* (see Figure 9). There, the user can visualise the measures they selected on a plotting area offering interactive zooming and other convenient features.

[3] `http://commons.apache.org`



17/02/2017 17:16:57	378.0ms	378.0ms
17/02/2017 17:19:11	194.0ms	194.0ms

Figure 7: Simulation View.



New simulation experiment...

Create a new simulation experiment

Experiment name: exp1
Model: Model.carma
System: Scenario1
Simulation Time: 100.0
Replications: 1000
Samplings: 50
Measure:

AverageBikes{l=0}

Figure 8: Dialog for the creation of a new simulation experiment.

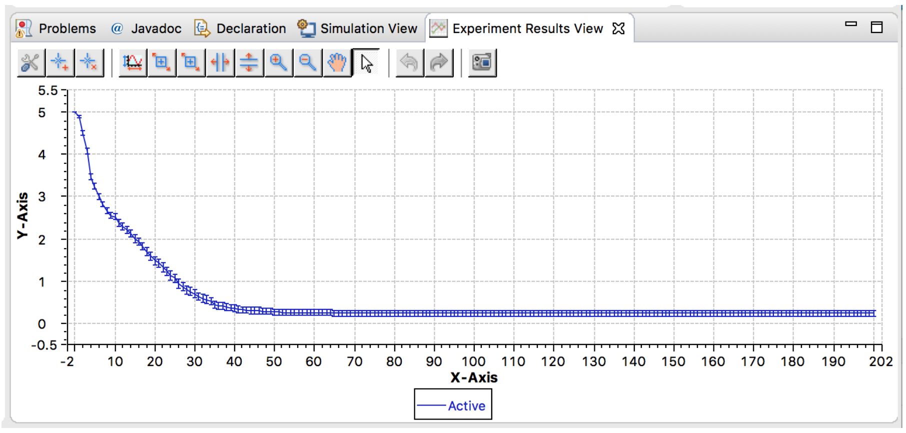


Figure 9: Results View.

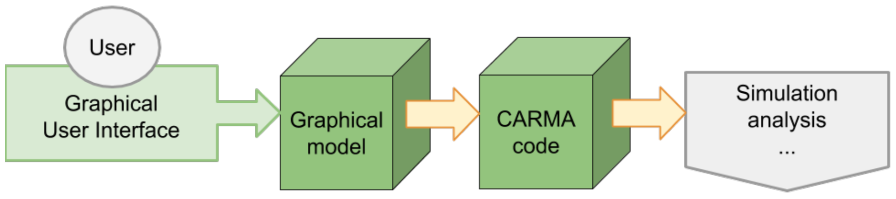


Figure 10: A flowchart depicting CARMA code generation from graphical input.

3.2 CARMA Graphical Plug-in

In this section we present an automatic tool for generating the CARMA model code from a graphical input.

The CARMA Graphical Plug-in (CGP) comprises a Graphical User Interface (GUI) for defining the locations and possible movements of components, as well as a module for the automatic generation of CARMA code that can be later used for simulation and analysis.

The GUI supports a graphical modelling layer on top of CaSL. The graphical modelling tool, consisting of a graphical editor and an implementation in the form of an Eclipse IDE plug-in, provides the user with visual ways of representing scenarios involving stationary, mobile and path-restricted agents. The graphical representation is then automatically translated into a CaSL model template. The code generation scheme is depicted in Fig. 10.

In the current form of the graphical modelling tool we focus on systems in which the movement of components is constrained to follow certain routes in space, each route defined by a path. More precisely, we consider systems which have the following properties:

1. *The environment of the system contains the definition of one or more paths (represented by graphs) which specific groups of components can traverse in order to change their location.*
2. *Components can be classified into one of three groups based on their ability to move in space:*
   (a) *Stationary components* — their location attributes are constant (e.g. bus stops).
   (b) *Path-bounded components* — can only move along specified paths, their location attribute values belong to the set of node locations of nodes within the specified paths (e.g. buses following their routes).
   (c) *Free components* — can freely change their location attribute to any value (but are still bound by the environment's definition of space, i.e. a grid) (e.g. bus passengers walking to bus stops).
3. *The spatial locations of components within the system contribute either directly or indirectly to* measures *calculated during model evaluation (in other words we are interested not only in the topological arrangements of the locations of components but also in the distances between nodes).*

Examples of systems with constrained movement include public/private transport networks, heterogeneous computer networks, pedestrian city networks, secure computer networks, and many others.

In the CGP, paths are represented by graphs consisting of nodes, connected by edges. Nodes are placed on a grid which is an unbounded 2D plane, that can be tessellated to define grid points. To reflect their placement on the plane, every node has a location attribute which is a co-ordinate in two-dimensional space. The nodes are labelled, and the labels are non-unique. This means that they can be grouped into sets according to their labels, and also that in order to uniquely identify a node, its label and location are both needed. For a given component type, the user can define the set of node labels that the component is allowed to visit in a given state.

The edges in a path graph are directed and labelled. The direction of an edge constrains movement on that edge to be in that direction. The label of an edge constrains the types of components which can move along the edge, in a given state.

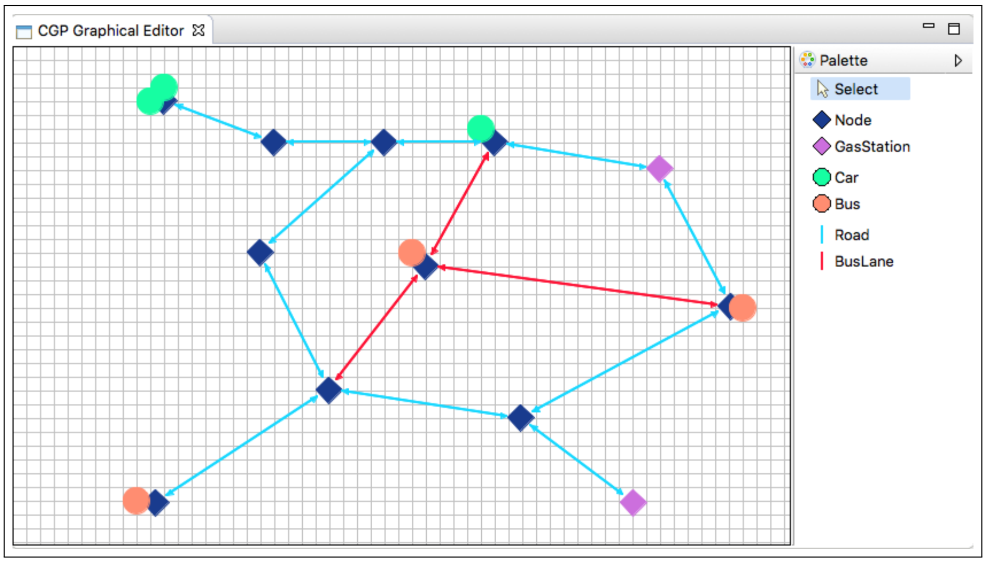


Figure 11: A screenshot of the graphical interface for path and components layout.

For example, in the carpooling scenario in [27], Car components are allowed to move on the FastLane path only if they are in the state PRIVILEGED, otherwise they can only move on the SlowLane path.

The graphical palette (see Fig. 11) allows the user to instantiate nodes, and the path connecting them, by laying out the nodes on the plane. From the user's point of view, the creation of path node instances is very similar to the creation of component instances.

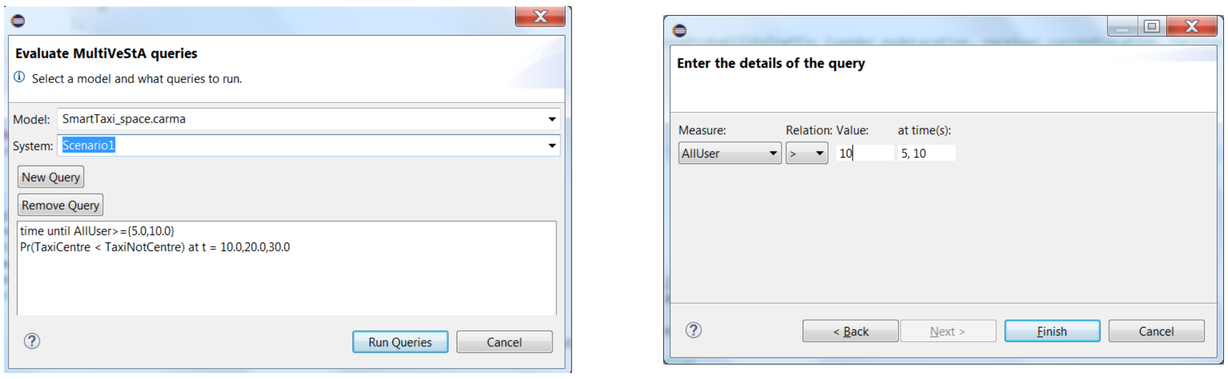
The user can specify a component type using structured input. The identifier and appearance of the component can be defined as well as the processes defined in the component, its allowable path and non-movement actions. Once a component type has been defined instances of that component type can then be placed within the graphical layout (by drag and drop). Component instances of the same type differ only in the values of their attributes and therefore can be represented by identical symbols. Their placement on the plane determines their location attribute. The state of a component, given by the value of one of its attributes, can determine if that instance is allowed to move on a particular path.

CAS by their nature are large-scale systems so concepts such as location, separation, distance and movement very often have roles to play in their models. By concentrating on location and movement, the CGP provides a convenient separation of concerns between the spatial aspects of a model (such as location, proximity and movement) and the dynamic aspects of a model (such as attribute and state update, communication, and synchronisation). We believe that this separation can be helpful in allowing the modeller to focus their attention on particular aspects of the model in isolation.

The CGP handles all of the low-level aspects of location representation such as placement on a co-ordinate system and the consistent handling of co-ordinate values throughout the model. This level of detail is often tedious and error-prone to maintain manually so we believe that the model generation approach also benefits CARMA modellers.

3.3 Statistical Analysis of CAS

The Eclipse plug-in can be used to simulate models and thus gain an understanding of their behaviour. To provide further analysis options for CARMA models beyond what the plug-in offers, we have developed an interface to the MultiVeStA platform, allowing CARMA users access to the model-checking capabilities offered by the software. MultiVeStA, developed by Stefano Sebastio and Andrea Vandin partially within the project, can be used to evaluate the expected value of quantities of interest in different kinds of systems through a simulation-based procedure. The expressions of interest are defined using the special-purpose MultiQuaTEx language. In the case of CARMA, such quantities can involve the current time in the simulation, the value of


Evaluate MultiVeStA queries

Select a model and what queries to run.

Model: SmartTaxi_space.carma

System: Scenario1

```
time until AllUser>=(5.0,10.0)
Pr(TaxiCentre < TaxiNotCentre) at t = 10.0,20.0,30.0
```

Enter the details of the query

Measure: AllUser

Relation: >

Value: 10

at time(s): 5, 10

Figure 12: Visual interface for creating MultiVeStA queries: main dialog (left) and sample wizard for creating a new query (right).

a measure or the number of times an action has occurred. These give a user access to a rich set of queries to place on a model, and provide deeper insight than what could be gained by simulation alone.

There are two ways in which MultiVeStA has been integrated into the existing tools. The first way is through the GUI of the Eclipse plug-in. In this case, the query creation is a guided process: a menu entry opens a wizard presenting options for different kinds of queries. Specifically, we allow the user to estimate the following types of quantities:

- time until the value of a measure becomes equal, less, or greater than a specified value;
- time until the value of a measure becomes equal, less, or greater than that of another measure;
- probability that an inequality condition holds between the values of two measures, or a measure and a concrete value, at one or more specified time points;
- number of times an action has occurred until one or more specified time points.

The offered options cover a large range of common situations; the user can, for example, ask questions like "What is the probability that the number of free taxis is greater than the number of travelling taxis at time . . . ?" or "How long does it take, on average, until the number of waiting users becomes less than a desired threshold?". The queries are specified by selecting elements in the graphical interface (as illustrated in Figure 12), and the plug-in automatically generates the corresponding MultiQuaTEx expressions, invokes MultiVeStA and presents the evaluation results. This way, the user does not need to be familiar with the underlying query language.

The second way of accessing the MultiVeStA functionality is through the command line interface. In this case, the user must provide a file with the expressions to be evaluated. Additional parameters can be given to customise the default behaviour of the algorithm, such as by specifying the desired confidence level of the result. In contrast with the plug-in integrated workflow, using the command line interface requires some familiarity with the MultiQuaTEx syntax, but in return offers access to an ever wider range of potential queries.

3.4 Command Line Interface

In addition to the Eclipse plug-in, we have developed a tool that can be executed from the command line. Its goal is to allow users to perform some common tasks related to CARMA models through a simple, lightweight interface that is also amenable to scripting, thus providing programmatic access to some of the CARMA tools. The command line tool is available as a self-executing Java package and does not require an installation of the Eclipse environment or any additional libraries. It can therefore be used on any machine where Java 1.8 is installed and on any major operating system.

The main task of the command line tool is to serve as an interface to the CARMA simulation engine. This is useful for running jobs over server machines or for scheduling consecutive simulations, avoiding the need to initiate and oversee each individual task through the graphical interface.

The user provides a file describing one or more experiments to be performed, specifying parameters such as final time of the simulation and the measures to be recorded. The format of this description is the same as the one produced by the Eclipse plug-in, so that files generated through the GUI can be reused in the CLI. If desired, the user can override certain parameters of the experiment file, such as the number of replications to be executed, and can also set the seed of the random number generator used in the simulations. These features allow for easier programmatic manipulation and replication of experiments.

To take advantage of the multi-core architectures found in many computing servers and clusters, the tool allows the user to specify an optional *parallelization* parameter N. If so provided, this will automatically split each experiment into N subtasks and attempt to execute them in parallel, using different processing threads or cores as determined by the operating system used. When all the subtasks are completed, the tool collects each set of results and aggregates them to produce the overall statistics. A diagram of the process is shown in Figure 13.

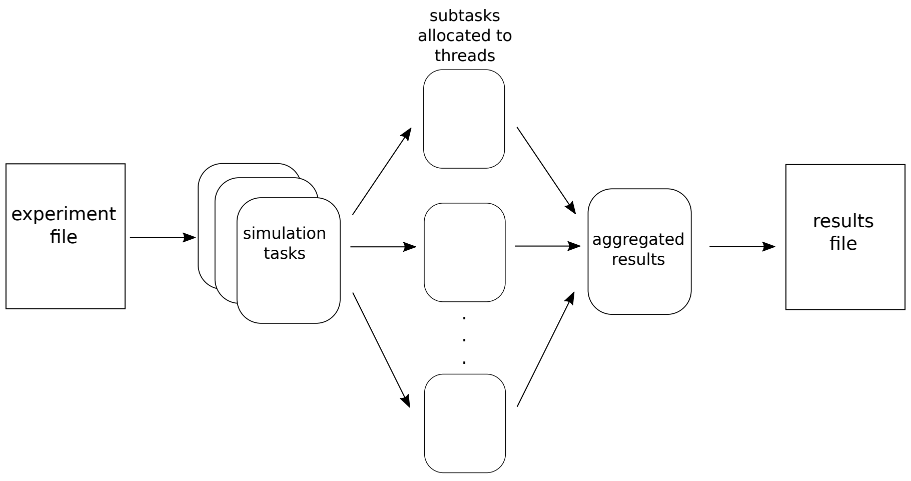


Figure 13: A depiction of simulation tasks executed in parallel through the CARMA command line interface. An experiment description is read from a text file, the corresponding simulation job is split into subtasks to be run in parallel, the tool aggregates the simulation results and stores them in a text file. The process is repeated for each experiment contained in the input file.

Once all simulations are finished (whether executed sequentially or in parallel), the results are stored in CSV format, with one file for each measure requested, in a location optionally specified by the user. For each measure, the file contains the mean and variance of its value at the different time points sampled. To help with the organisation of experimental results and to facilitate potential replications of the experiment, several important aspects of the simulation are recorded. Specifically, in addition to the results, the following files are created to provide metadata for the process:

- a copy of the model used for the simulation;
- a copy of the segment of the experiments file corresponding to the particular experiment (reflecting any overriden parameters), which can then be reused as input to the tool;
- a text file containing a human-readable summary of the experiment, including the model, aspects of the simulation (such as the stopping time), any user-specified parameters, the time required for the experiment, and the date and time of execution (Figure 14).

Furthermore, a script file for the gnuplot or MATLAB software is created, allowing the user to easily produce visualisations from the saved results if desired. The script can be run as-is or further edited by the user as required.

While simulation is the primary goal of the command line interface, it also offers other functionality. A second option is to perform more elaborate statistical analysis through MultiVeStA (Section 3.3). This is done

```
Summary for experiment exp1:
--------------------------
This experiment used the model /path/to/file/Model.carma. A copy has been
    saved in this directory.
The scenario considered was Scenario1.
The experiment tracked the following measures: TotalUsers, MaxBikes{l=0}.
The final time of the simulation was 25.000 and 100 samplings were taken  (
    sampling interval: 0.25000).
10 replications were performed in 529 ms using the CARMA simulator.
The data from individual replications were combined and statistics (mean,
    variance) were computed using the Apache Commons Mathematics library.
This experiment finished at 20:38:23 on 16 February 2017.
```

Figure 14: Sample output generated by the command line tool, as a human-readable description of a simulation experiment.

by providing a file defining the quantities of interest in the MultiQuaTEx syntax recognised by the model-checker. The evaluation results are stored in a text file, and a plot is displayed and also stored in an image file. The tool can also be used to present a summary of the model to the user, as well as to produce a LaTeX file with information about the model.

3.5 Ongoing work

In the QUANTICOL project other tools and techniques have been developed to support analysis of CAS that have not yet been applied to CARMA and CaSL.

PALOMA is a stochastic process algebra developed during the first year of the project to explore ideas around the explicit representation of spatial information as an attribute and attribute-based communication. Although in some senses it has been superseded by CARMA, it has provided a useful vehicle for exploring techniques for model reduction and scalable analysis which could later be incorporated into CARMA. Many of these ideas and techniques have been incorporated into the PALOMA tool suite, meaning that this now provides a template for developing such techniques in the context of the CARMA tool suite. Moreover, PALOMA is also available as a modelling tool in its own right for modelling studies that do not need the full expressivity of CARMA.

As previously reported, the PALOMA Eclipse Plug-in provides a fully-featured development environment for modelling with the PALOMA process algebra. The plug-in consists of:

- An editor for PALOMA models with syntax highlighting functions;
- A simulator which supports population-level stochastic simulation of PALOMA models using Gillespie's algorithm;
- An alternative simulator that uses information about the requested measures to reduce the scale of the model to improve simulation efficiency [10];
- Plotting facilities for simulation results;
- A generator which can translate a PALOMA model to directly runnable Matlab scripts for moment-closure analysis using ODEs [11].

The technique developed in [11], involves deriving a set of ODEs to represent moments of the random variables characterising the system; typically higher moments are included rather than only the mean derived by fluid approximation or mean field approximation. This allows for a fuller understanding of the behaviour of the system, including quality-of-service type measures. The difficulty of this approach is that the analytically derived moment equations are not *closed*, in the sense that moments at each level depend on expressions involving higher level moments. The PALOMA tool suite includes a generator which automatically applies moment closure to the user-specified level, and as reported elsewhere, experimental results have shown that a high-degree of accuracy is achieved with significant speed-up. As there is a trade-off in the solution efficiency and the level of moments calculated, it is important that this is an aspect that the modeller is able to control, and a menu option is provided for this. In the implementation the generator outputs Matlab scripts of the appropriate ODEs for the selected level of moments, subject to the moment-closure approach. In order to port the approach to CARMA, the generator would need to be extended to consider all the syntax of CARMA but the basic method of moment-closure could remain the same.

Similarly in [10], an efficient simulation approach for PCTMC models, such as those that underlie PALOMA and CARMA models, is developed. In this approach, the agents of interest within the measures to be collected from a model are taken as the focus of a simulation run. During initial runs, statistical inference is used to identify the parts of the model that have little or no influence on the measures and those parts are systematically pruned from the model, reducing the overall simulation time substantially (previously reported in D3.2). This technique has also been implemented for PALOMA models.

We envisage a two stage approach to making these techniques accessible to CARMA models. In the first step we seek to identify a PALOMA-dialect of CARMA. In this way we will define the syntactic restrictions that allow a CARMA model to be translated into a PALOMA model. In this case, once the translation has been implemented it will be possible to apply both the moment closure and the efficient simulation analysis to some CARMA models. However, CARMA is a much richer language than PALOMA and so this will necessarily be quite a strong restriction on the models that can be handled. The longer term goal is to extend the techniques to address a fuller set of CARMA models. Even with this approach there may still need to be some syntactic limitations. For example, the efficient simulation technique relies on a fixed population of components within the system, so it could only be applied to CARMA models in which the environment does not create new components during the evolution rule.

Another tool/technique that could be integrated with CARMA is the one presented in [6]. There a FlyFast front-end modelling language has been extended in order to deal with *components* and *predicate-based interaction*. The extension has been inspired by CARMA. Each component is expressed as a pair *process-store* and incorporates an outbox for communication; actions are *predicate based multi-cast* output and input primitives operating on outboxes. Associated with each action there is also an (atomic) probabilistic store-update. As in the original FlyFast language, component interaction is probabilistic, but now the *fraction* of the components satisfying the relevant predicates plays a role in the computation of transition probabilities. The formal probabilistic semantics of the extended language has been provided, as well as a translation to the original FlyFast language that makes the model-checker support the extended language (more details are available in Deliverable D3.3). This approach can be used as another base for a *scalable analysis* of CARMA models. However, some limitations would need to be overcome. Indeed, even if the language proposed in [6] and CARMA/CaSL are both based on attributed based communication, they are based on different semantic models.

4 A design workflow and analysis pathway for CARMA models

CaSL is sufficiently expressive that it can be used for many different types of modelling and analysis, as illustrated in Section 2.3. The QUANTICOL project has therefore defined an analysis pathway that will guide less experienced users through this process, directing them first to the simple and inexpensive analysis which must be done on models, then leading them to a selection of the more computationally expensive analyses which need only be done if initial inexpensive checks have been passed.

The CARMA modelling tools defined by the project support the design and analysis pathway from the early phases of model specification and formation through detailed model development to efficient analysis through the definition and execution of a suite of related experiments with the model and its variants in the form of a simulation ensemble study in which the model is repeatedly simulated with different values of the model parameters in order to investigate the effect of environmental changes on system behaviour.

We can think of the pathway as having three distinct phases, *model formation*, *model development* and *model analysis*, each of which is served by one of the CARMA tools: the Graphical Plug-in, the Eclipse Plug-in, and the command-line simulator, as illustrated in Figure 15.

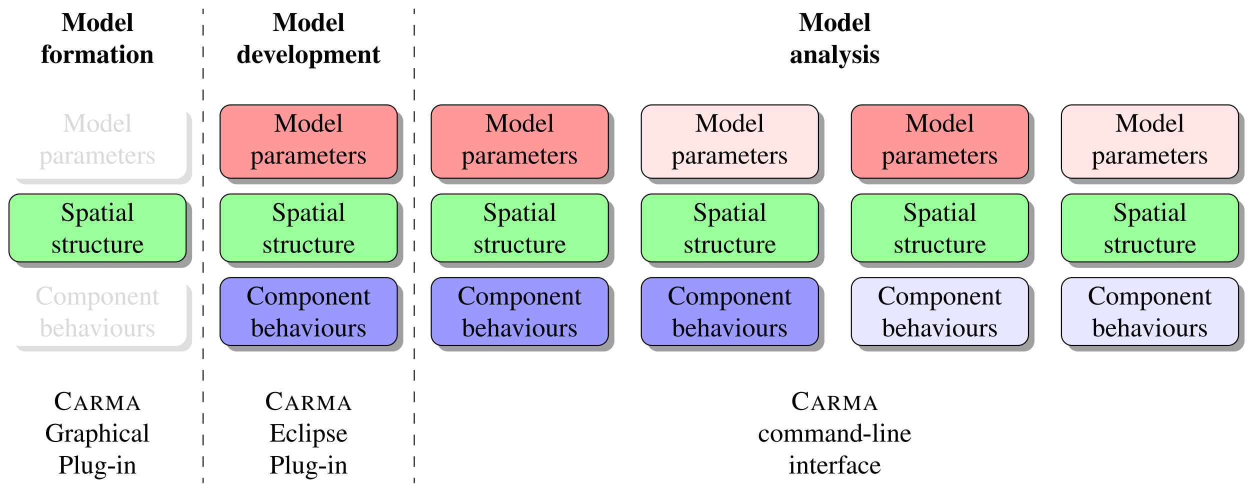


Figure 15: The CARMA design workflow and analysis pathway leading from model formation, through model development, to analysis of a family of model variants. The colours indicate particular sets of parameters or behaviours, showing that the command-line interface is used to perform a series of experiments on different combinations of parameters and behaviours, against the background of a fixed spatial structure.

Model formation: In the early stages of model formation the goal is to identify the modelling problem which is to be precisely formulated. This can often involve interaction between modellers who bring relevant technical modelling expertise and problem stakeholders who bring relevant domain knowledge. In order to ease communication between these groups the CARMA *Graphical Plug-in* provides a visual language for describing location-based systems. Concepts can be expressed in this visual language separately from the formal syntax of a specific modelling language, helping problem stakeholders communicate effectively with modellers.

The end result of this stage of interaction is a model *fragment* which includes component concepts and location information but does not include model dynamics in terms of inter-component behaviour or probabilities or rates of communication events. In the process of developing this CARMA model fragment in discussion with the problem stakeholders the modeller is likely to have gained some significant insights into the model dynamics and the model behaviour but these are not yet formally expressed in the model. The next phase of the design workflow and analysis pathway is to add these to the model fragment to produce a syntactically-complete CaSL model which can be analysed by simulation or model-checking.

Model development: In this phase of the pathway the main focus of the activity is on extending the CARMA model with compelling abstract representations of the events, processes, and interactions which will capture the essence of the CAS which is being modelled. This is the kernel of the creative process which we know as the art of modelling and at this stage the model will be extended, re-drafted, and revised considerably as ideas are added, enhanced, refined, and polished. There is a strong expectation that the model will go through many revisions during development.

CARMA models are textual in nature and therefore can be checked into version control repositories, updated and restored to earlier versions, and managed just like application source code. Although there is generally no requirement for reproducibility in the *model development process* itself, it can still be very useful to archive different versions of model drafts created at this stage in case once-promising abstractions turn out to be too generic to accurately capture the essence of the system dynamics, or they are instead too specific to allow some system-permissible behaviours to be observed at all. After uncovering this flaw, the modeller can revisit model versions which have been committed into the model repository and reconsider earlier drafts which do not include the flawed abstraction.

At this stage when fleshing out the model fragment with decisions about the model dynamics the model developer must identify the synchronisation activities between components and the nature of communication events (as broadcast or unicast). Parameter values are chosen for rates, weights, and probabilities. This stage of model development and debugging is supported by the CARMA *Eclipse Plug-in*, a fully-featured integrated development environment for CARMA models. It contains a rich syntax-aware editor with powerful editing functions and an iterative compiler which provides the modeller with interactive feedback on model errors and insecurities. Model debugging is supported explicitly by providing an experiment view (Section 3.1) which allows the modeller to step through a simulation event-by-event, observing the states between, looking for modelling errors such as synchronisations which can never occur, or are matched with the wrong communication partner.

It is here that the advantages of the CaSL language become most apparent because its intention is to make it easier to express CARMA models concisely and succinctly, even when the models have many components and complex spatial structure. It is important to provide strong language support for the correct expression of models; errors in the text of a model give rise to a (perhaps plausible-seeming) representation of the modelling concepts which does not faithfully capture the modeller's intentions. Analysis results computed from an erroneous model text will be misleading at best and positively harmful at worst.

CaSL allows for the declaration of components and the environment as in the definition of CARMA but it also gives additional features that are necessary when making a model concrete for automated analysis. In particular, it allows for constants and functions to be defined to support the description of models. In addition to this, it adds a layer of typed data structures including enumerations, record types, and heterogeneous collections such as sets and lists, complemented by syntactic constructs in the language to make defining collections over parameter ranges simple and convenient, reducing the likelihood of modeller error. Taken together, these supplement the core CARMA process calculus and provide a level of semantic and type security which is not offered by process calculi with untyped value-passing, eliminating a whole class of modelling errors which are not caught by untyped languages.

Furthermore, as we have seen, the CaSL language has an explicit spatial syntax to describe space, an important feature determining the behaviour of many CAS. This allows the definition of nodes (either as coordinates or names) and links between these nodes. There is also syntax to support the use of this space, in particular, a way to refer to both the pre-set and post-set of a node, which then permits a generic definition of moving components that can traverse over any spatial structure that can be captured as a finite graph. Taken together, these additional language features in CaSL provide a basis for strong static analysis of models, catching modelling errors at compile-time which would not be detected in modelling languages without this kind of support for representation of typed data and spatial structure.

Type-checking and static analysis of models are applied automatically at every edit on models at this stage. These are low-cost analyses which must succeed before the model is allowed to progress to more costly dynamic analysis, e.g. via simulation. The end product of this stage of the design and analysis pathway is a thoroughly-debugged model which is ready for efficient model analysis.

Model analysis: At this stage, model editing, extension, and revision are not the main activity so the rich graphical interface which supported interactive model development no longer has the central role that it did. As an alternative to the CARMA Eclipse Plug-in, the CARMA *Command-line Interface* tool can be used to schedule a series of experiments which will then run headless without user interaction (for example, on a multi-core compute server). Perhaps the last useful service which the CARMA Eclipse Plug-in provides for us here is support for the creation of experiment definition files. Experiments are treated as first-class objects in the model analysis stage, and the focus here is on them as much as on the CARMA model which is the input to the experiments.

Reproducibility of the analysis results becomes a primary concern here because it may not always be obvious to the modeller which analysis results will be preferred by the problem stakeholder so it must be possible to defend the provenance of a set of results by reproducing them to confirm the model parameters and model version which gave rise to the desirable outcomes. The CARMA command-line interface tool pays special attention to this by adding explicit support for reproducibility by archiving model versions together with experiment settings and analysis results. Simulation results can be made reproducible down to the level of pseudo-random number generation by specifying the initial random number seed for the CARMA command-line simulator, and this reproducibility of results is maintained even in the presence of parallel execution of simulation ensembles on a multi-core architecture (see Section 3.4).

Taken together, the CARMA modelling tools support the design workflow and analysis pathway from initial model creation with problem stakeholders through detailed model development to efficient reproducible analysis of families of related models. The design of the CARMA modelling tools is intended to support good modelling practice by working harmoniously with version control systems which allow model versions to be archived.

As with all model-based development, it is prudent when modelling with CARMA to invest some resources into creating a build process to automate re-running analyses. This applies particularly if one has any expectation that the CARMA model will be likely to change. The use of a build process together with experiment definition files and an archive of model versions under revision control facilitates rolling back to an earlier version of the model and applying exactly the same form of analysis if the results of that analysis on the current version of the model are found to be unfavourable. Crucial to the effectiveness of automating model analysis is the command-line interface to the CARMA simulator, which allows model analysis to be scriptable, with the scripts themselves becoming part of the modelling project, and archived in a version control system, facilitating repeatable analysis results.

5 Analysis of CaSL models

In this section we will show pathways of application of techniques developed in QUANTICOL, to support specification and analysis of CAS. Two scenarios taking inspiration from the *Smart Cities* context are considered.

In Section 5.1 we first use CaSL and its tools to model a *Bus System*. After that, the spatio-temporal logic SSTL [23] and the tool jSSTL are used to specify properties of the considered system.

In Section 5.2 CaSL is used to model the homogeneous bike sharing system (BSS) model, originally proposed in [12]. Then, following the procedure proposed in [19], the model is transformed into a IDTMC that is used in turn as an input for FlyFast. This is an *on-the-fly mean field* probabilistic model checker for bounded PCTL (Probabilistic Computation Tree Logic) properties of a *selected individual* in the context of systems that consist of a *large number* of independent, *interacting objects* (see Deliverable 5.3). Thanks to the use of FlyFast we estimate the probability that a given bike station will be either full or empty within a given timeframe.

5.1 Analysis of a Bus Scenario with CaSL

In this section we first show how CaSL and its tools can be used to model a *bus transportation system*. Our goal is to build a CaSL model that can be used to identify critical aspects that users can experience when the *real system* is in operation.

One of the first advantages we have in modelling a bus scenario with CaSL is the fact that in our language we can explicitly model the space, that in our case consists of the network of routes. We consider here a simple scenario composed by two routes, identified with the integers 1 and 2. These routes connect 8 stops numbered from 0 to 7. Route 1 is a *slow line* and connects all the stops in a sequential order. Differently, route 2 is a a *fast line* and only connects even stops. A special location numbered -1 is also used to identify the bus depot. Route 1 starts at location 0, while route 2 at location 4.

The model in CGP is reported in Fig. 16. The space definition associated with this graphical model is the following:

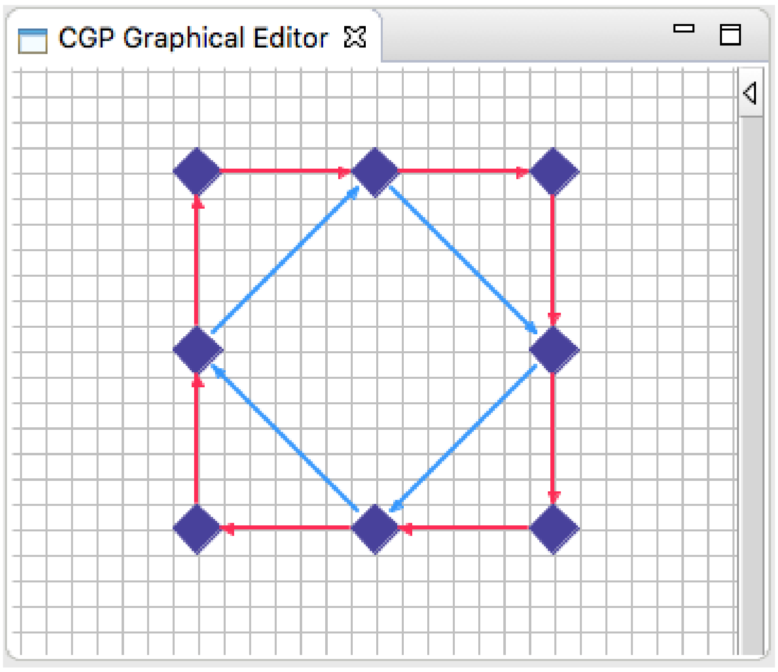


Figure 16: A simple mode for bus routes

```
space SimpleBusRoute(int stops) {
  universe <int zone>
  nodes {
    [ -1 ]; //Bus deposit location.
    for i from 0 to stops {
      [ i ];
    }
  }
  connections {
    //route 1
    for i from 0 to 8 {
      [i] -> [(i+1)%8] {route=1,mr=STANDAR_RATE_ROUTE_1};
    }

    //route 2
    for i from 0 to (stops/2) {
      [2*i] -> [(2*(i+1))%8] {route=2,mr=STANDAR_RATE_ROUTE_2};
    }
  }
}
```

Each connection in the model is equipped with two features: `route` and `mr`. The former indicates the route number that can traverse the connection, while the latter represents the *traversal rate*, that is the parameter of the exponential distributed random variable modelling the time needed to move from one node to the next one.

We can define the following CaSL function that can be used to select the next step of a bus route from a location:

```
fun location nextDestination( int route , location current ) {
  return current.outgoing().filter(
          @.route == route
        ).map(
          @.target
        ).select( 1.0 );
}
```

Above `current.outgoing()` is used to select all the edges exiting from the current location. From this set we *filter* only the ones that are in the route of interest `.filter( @.route == route )` and we select the set of possible destinations (`.map( @.target )`). Finally, we randomly select one of these (uniform distribution is used).

Similarly, the rate of a movement from one location to another can be computed using the following function:

```
fun real moveRate( int route, location current , location next ) {
  set<real> rates =
    current.outgoing(next)
      .filter( @.route == route )
      .map( @.mr );
  if (size( rates )==0) {
    return 0.0;
  } else {
    return rates.select(1.0);
  }
}
```

Both the functions above do not depend on the specific space definition, while it is only assumed that connections have a features `route` and `rm`. This allows us to consider more complicated configurations.

Three kinds of components are considered in the system: *stops*, *buses*, and *arrivals*. *Stops*, that are located at each bus stop, are used to coordinate the activities of buses. Indeed, when a bus arrives at a stop it receives a position in the *queue*. The bus at position 1 is the next to leave the location. Each time a bus leaves the stop all the other buses decrement their position in the queue by 1. To count the number of buses currently at a stop, attribute `buses` is used.

The prototype of component `Stop` is the following:

```
component Stop( ){
  store{
    attrib buses := 0;
  }

  behaviour{
    S = enter[my.loc == loc](){buses := buses + 1;}.A;
    + [buses>0]leave*[loc == my.loc]<loc>{buses := buses - 1;}.S
    A = queueorder*[loc == my.loc]<buses>.S;
  }

  init{
    S
  }
}
```

The behaviour of component `Stop` is initialised to state `S`. In that state a component can either receive a bus or, if there are buses at the stop, let the first bus leave the stop. A bus is received by executing input action `enter`: attribute `buses` is incremented by 1 and state `A` is activated and the position in the queue is sent to the bus that has just arrived. If there is at least one bus at the stop (`buses>0`) then component `Stop` performs the broadcast output `leave`. This will be received by all the buses at the stop: the first one will leave the stop, while all the others will advance in the queue.

The prototype of a bus component is the following:

```
component Bus(int number){
  store{
    attrib route := number;
    attrib location next := none;
    attrib queuepos := 0;
    attrib end = -1;
  }

  behaviour{
        G = arrive*[true](x,y) {
          loc = x;
          end = y;
        }.S;
        S = enter[my.loc == loc]<>.Q;
        Q = queueorder*[loc==my.loc](x){
          queuepos = x;
        }.W;
        W = [queuepos==1]leave*[loc == my.loc](x){
          next = nextDestination(my.route,loc);
        }.T
        + [queuepos>1]leave*[loc == my.loc](x){
          queuepos = queuepos-1;
        }.W;
        T = [loc.zone != end ]move*[false]<loc,next>{
          loc = next;
          next = none;
        }.S
        + [loc.zone == end]maintenance*{
          loc = [ -1 ];
          next = none;
        }.G;
      }

      init{
        G
      }
    }
```

Component `Bus` has four attributes: the traversed route (`route`), the next stop (`next`), the position in the queue at a stop (`queuepos`) and the index of the final stop (`end`). We assume that at the beginning all buses are located in the depot (state `G`). A bus returns to the garage when it terminates its trip.

When in state `G` a bus is waiting for the assignment of a route that is identified by a starting and a terminating location. A bus receives the assignment via input action `arrive`. When the route is assigned a bus *moves* to the starting location (this movement is *modelled* via the assignment `loc=x`) and is ready to *enter* the assigned bus stop.

The arrival of a bus at the bus stop is managed by state `S`. This state first performs (unicast) output action `enter` that can be received by one component (in our case a `Stop` component). After that the bus waits (in state `Q`) for a position in the queue. Having received a position, the bus evolves to state `W` where the attribute `queuepos` is decremented by 1 each time a message `leave*` is received. When the bus is *ready* to leave the station (`queuepos==1`) it evolves to state `T` where it can either move to the next location or, when the final destination has been reached, return to the garage.

To model the assignment of buses to trips, component `Arrival` is used:

```
    component Arrival(int route, int end) {

      store {
        attrib route = route;
        attrib end = end;
      }

      behaviour {
        A = arrive[my.route==route]<loc,end>.A;
      }

      init {
        A
      }

    }
```

This component, following a common pattern in CaSL specifications (see for instance Deliverable D4.2), continuously sends the appropriate message to buses in the garage to assign them a trip.

The system specification is reported below:

```
system ScenarioTest1{
  space SimpleBusRoute(SIZE)
  collective{
    for l in locations {
      new Stop()@l;
    }
    new Arrival( 1 , 7, 1.0/5.0 )@[ 0 ];
    new Arrival( 2 , 2, 1.0/5.0 )@[ (SIZE/2) ];
    new Bus(1)@[-1]<SIZE_ROUTE_1>;
    new Bus(2)@[-1]<SIZE_ROUTE_2>;
  }

  environment{
    rate{
      arrive{
        if (sender.route==1) {
          return ARRIVAL_RATE_ROUTE_1;
        } else {
          return ARRIVAL_RATE_ROUTE_2;
        }
      }
      leave* {
        return LEAVE_RATE;
      }
      maintenance* {
        return MAINTENANCE_RATE;
      }
      move* {
        return moveRate( sender.route , sender.loc , [ sender.next ] )
      }
      enter {
        return ENTER_RATE;
      }
      queueorder* {
        return QUEUE_RATE;
      }
    }
  }
}
```

we have a component `Stop` at each location while all the buses are located in the garage. The component that manages the arrival of buses for route 1 is located at `[ 0 ]`, while the one that manages buses for route 2 is located at `[ 4 ]`.

A measure we can consider is the amount of time that a user can wait for a bus. We can observe that a bus is accessible by users at a location `l` when it is in the state `W` at that location. The following measures can then be used:

```
measure RouteOneReady( int i ) = #{Bus[W] | my.loc == [ i ] && my.route == 1};
measure RouteTwoReady( int i ) = #{Bus[W] | my.loc == [ i ] && my.route == 2};
```

The data collected from the simulation for locations 0 and 4 are reported in Fig. 17 and Fig. 18.

However, these results are not enough to estimate correctly the availability of buses for users. Indeed, a user is interested in the amount of time she has to wait until a bus will be available. In particular one could be interested in verifying (or monitoring) a property of the form: *a bus will be available in the next* 10 *time units*. This property should be monitored while the system is operating.

To specify this property we can use Signal Spatio-Temporal Logic (SSTL), while the analysis can be performed via the jSSTL software tool. SSTL is a recently-designed linear-time temporal logic [22, 23], suitable

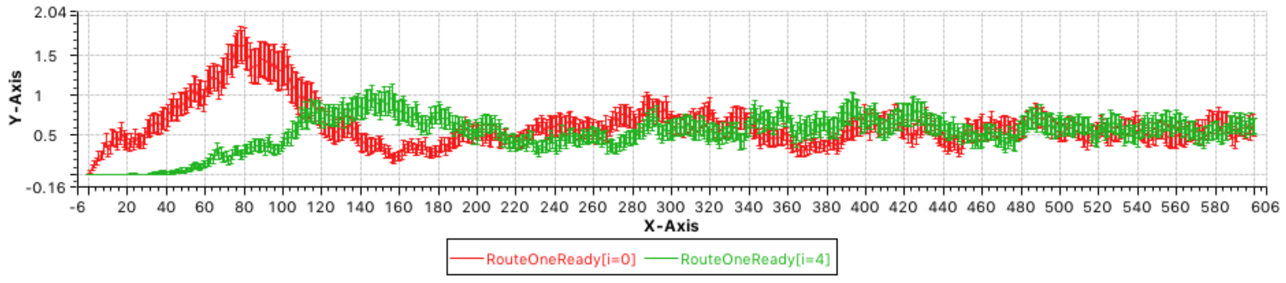


Figure 17: Available Buses at Locations 0 and 4: Route 1

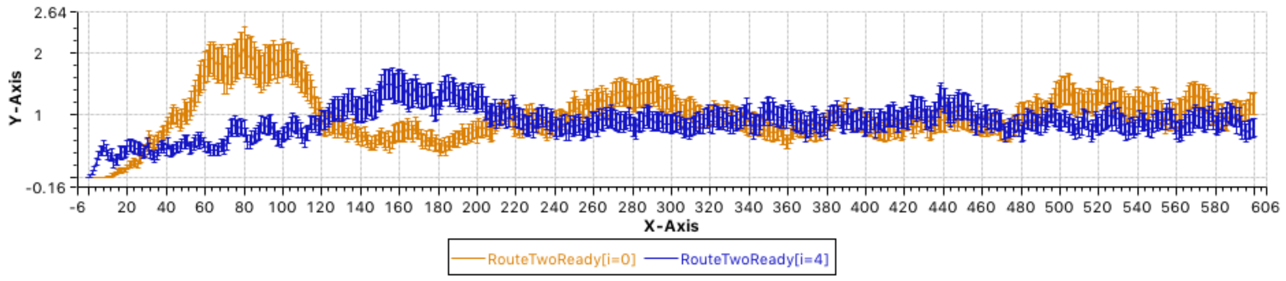


Figure 18: Available Buses at Locations 0 and 4: Route 2

for describing behaviours of spatio-temporal traces generated from simulations or measured from real systems, while jSSTL is a front-end developed as an Eclipse plug-in that provides a user friendly interface to the tool. Both jSSTL and SSTL are described in Deliverable 5.3.

The property of interest is expressed in SSTL via the following formula:

$$\Diamond^{\leq 10} w_i > 0$$

where w_i represents the number of buses of route i ready at a given location (this value is obtained for each location from the simulations in the CARMA Eclipse Plugin). The analysis of the obtained results are reported in Fig. 19.

We can observe that after an initial startup phase, both the routes satisfy the requested property with a robustness greater than 1. This means that, in the average, more than 1 bus will be available in the next 10 time units. However, this initial phase is longer for route 1. This because the first bus must follow a longer trip to reach location 4 from the first stop (location 0) than route 2. This is somehow expected. To solve this problem one could let buses start from different locations to minimise the waiting time at the beginning of the service.

5.2 Analysis of a Bike Sharing System with CaSL

In this section we show a pathway of application of techniques developed in QUANTICOL, starting from a formal CARMA model specification, up to probabilistic mean field model-checking using FlyFast. We consider the homogeneous bike sharing system (BSS) model, originally proposed in [12], consisting of $N = 1000$ stations of capacity $K = 30$ and a fleet size of sN bikes, where s is the average number of bikes per station. The rate of users taking a bike at a single station is λ; if there is no bike available in the station, the user does not use the BSS. Moreover, we assume that the average travel time is $1/\mu$. The model abstracts from the actual distribution of bike stations in space (hence the attribute "homogeneous") and assumes that stations are randomly chosen by users.

In [12] a normalised population CTMC $(Y_1(t), \ldots, Y_K(t))$ for the system is considered where $Y_k(t)$ is the current fraction y_k of stations with k bikes parked, over N, for $0 \leq k \leq K$. The rate of a transition associated

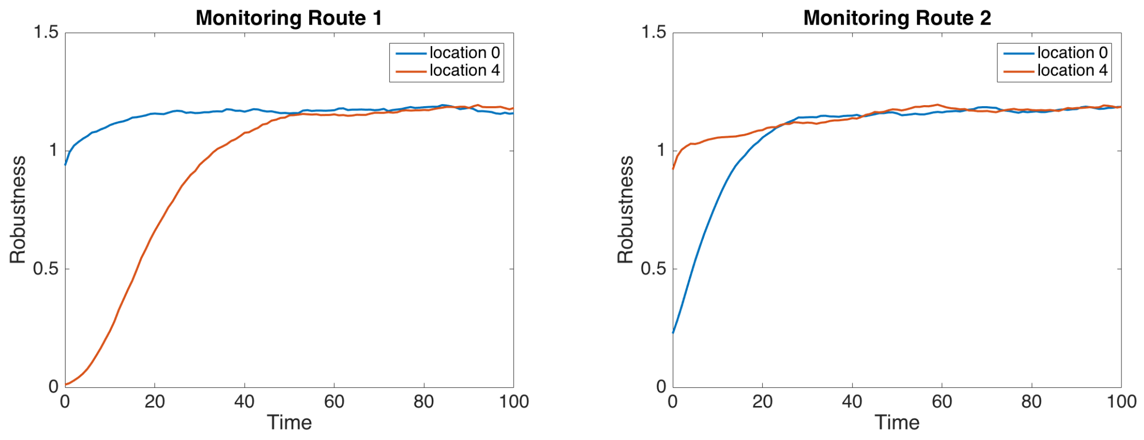


Figure 19: Availability of buses at locations 0 and 4: for route 1, property: $\Diamond^{\leq 5} w_1 > 0$ (left) and for route 2, property: $\Diamond^{\leq 5} w_2 > 0$ (right).

with taking a bike from a station with k bikes (for $0 < k \leq K$) is $\lambda y_k N$. The rate of a transition associated with returning a bike to a station with k bikes (for $0 \leq k < K$) is $\mu y_k (sN - \sum_{n=0}^{K} n y_n N)$.

The CARMA model of the system for $s = 5$ is shown in Fig. 20,where we assume that, initially, each station has s parked bikes as reflected by the definition of functions `cap` and `parked`.

A generic station is modelled by the component `HBSStation` with two attributes: `cp`, recording the capacity (initialised to K and constant in time); and `npb`, recording the number of bikes parked in the station. The behaviour of the station is straightforward; it consists of a single state `Y` since the relevant information is kept in attribute `npb`. The actions modelling `get`ting and `return`ing bikes have no synchronisation requirement (they play the role of “internal” actions in the process algebra sense). The rate of a `get` transition of an individual station is constant ($\lambda = 1.0$). The rate of a `ret` transition is given by μ multiplied by the number of bikes in circulation in the system divided by the total number of stations N. In the CARMA model, the number of bikes in circulation is kept in the global attribute `incirculation`.

Fig. 21 shows the number of problematic stations (i.e. stations which are empty or full) when the simulation (i.e. the average of 10 simulation runs) of the CARMA model reaches a stable state (after about 100 time units), for $s \in \{5, 10, 15, 20, 25, 30\}$. It can be seen that the values obtained with the CARMA model are very close to those shown in Fig. 1 in [12], for the above values of s. This is not surprising since in the (normalized) population model associated with the CARMA specification, the rate of a `get` transition from a station with k bikes parked is $\lambda y_k N$, where y_k is the current fraction of stations with k bikes parked, as usual; the rate of a `ret` transition from a station with k bikes parked is $\mu y_k N \frac{c}{N} = \mu c y_k$, where c is the current value of global attribute `incirculation`; obviously $c = sN - \sum_{n=0}^{K} n y_n N$.

In the sequel, for the sake of simplicity[4], we let $K = 10$. We first of all note that the information kept in attribute `npb` can be coded in state `Y` itself, thus giving rise to a model with $K + 1$ states, say `Y0, Y1, ...Y10` and the obvious transitions between states `Y0` and `Y1`, `Y1` and `Y2`, `Y2` and `Y3`, etc. as in Fig. 22. The latter is actually an ICTMC where the probabilistic behaviour of an individual station is a function of the occupancy measure vector.

Then, following the procedure proposed in [19], the model is transformed into a IDTMC in such a way that it has the same local states and branching structure as the ICTMC. Moreover, from the IDTMC we get a set of difference equations that can be used to approximate the solution of the ODEs underlying the ICTMC model according to the Euler forward method. We first of all choose a value q which is at least as large as the maximum of the exit rates in the ICTMC, in a similar way as in the uniformisation technique for CTMCs. We

[4]This simplification is due only to the fact that the CARMA to FlyFast translation has been carried out manually since no automatic tool is available yet to that purpose. We anyway point out that FlyFast could easily deal with a model specification with $K = 30$; the upper bound for the set of agent states of a FlyFast model specification is currently set to 256.

```
const N = 1000;
const K = 30;
const s = 5;
const lambda = 1.0;
const mu = 1.0;

fun int cap(int station_id){return K;}

fun int parked(int station_id){return s;}

fun int totparked(int n){
  int acm = 0;
  for i from 1 to n+1 {
    acm := acm + parked(i);
  }
  return acm;
}

component HBSStation(int capacity , int npbikes) {
  store {
    attrib npb:= npbikes;
    attrib cp := capacity;
  }

  behaviour {
    Y = [my.npb > 0] get*[false] <> {my.npb := my.npb - 1;}.Y
      + [my.npb < my.cp] ret*[false]<>{my.npb := my.npb + 1;}.Y;
  }
  init {Y}
}

measure Empty = #{HBSStation[Y] | my.npb==0};
measure Half = #{HBSStation[Y] | my.npb==s};
measure Full = #{HBSStation[Y] | my.npb>=K};
measure BikesCircling = global.incirculation;

system HBS {
  collective {
    for (i ; i < N ; 1) {
      new HBSStation(cap(i), parked(i));
    }
  }

  environment {
    store {attrib incirculation := N*s - totparked(N);}
    prob {}
    weight {}
    rate {
      get* {return lambda;}
      ret* {return mu * real(global.incirculation) / real(N);}
    }
    update {
      get* {incirculation := global.incirculation + 1;}
      ret* {incirculation := global.incirculation - 1;}
    }
  }
}
```

Figure 20: CARMA BSS Simple Model

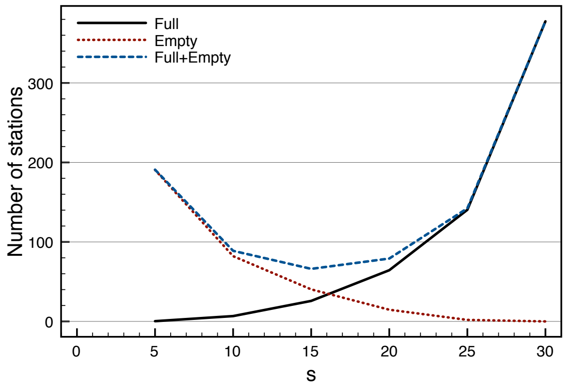


Figure 21: CARMA simulation results for $N = 1000$ and $K = 30$ and $s \in \{5, 10, 15, 20, 25, 30\}$.

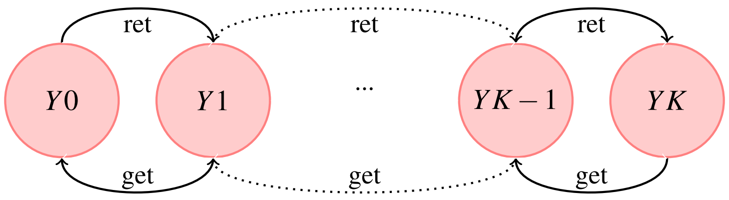


Figure 22: Bike station.

observe that generic state Yk has (at most) two outgoing transitions, one with rate λ and the other with rate $\mu y_k(s - \sum_{n=0}^{K} n y_n)$; note that the latter depends on the occupancy measure $(y_1, \ldots, y_K)$. In addition, we note that $s - \sum_{n=0}^{K} n y_n \leq s$ since, in the worst case, all bikes are in circulation. So it is sufficient to choose $q = \lambda + \mu s$. This leads to the following probabilities for the IDTMC: $\frac{\lambda}{\lambda+\mu s}$ for a get and $\frac{\mu}{\lambda+\mu s}(s - \sum_{n=0}^{K} n y_n)$ for a ret transition.

Using the transformed model as an input for FlyFast [20] we can compute the mean-field approximation of the dynamics of the occupancy measure vector, shown in Fig. 23 (right). In Fig. 23 (left), the results of the average of 10 simulation runs of the CARMA model of Fig. 20 are shown, for $K = 10$. The result shows close correspondence up to the necessary rescaling of time; 1 time-unit in the CARMA model corresponds to $q = \lambda + \mu \cdot s = 1 + 1 \cdot 5 = 6$ steps in the FlyFast model.

Using the full model-checking functionality of FlyFast in Fig. 24 we show the probability of an individual station to become full (after being empty initially) and to become empty (after being full initially) within 100

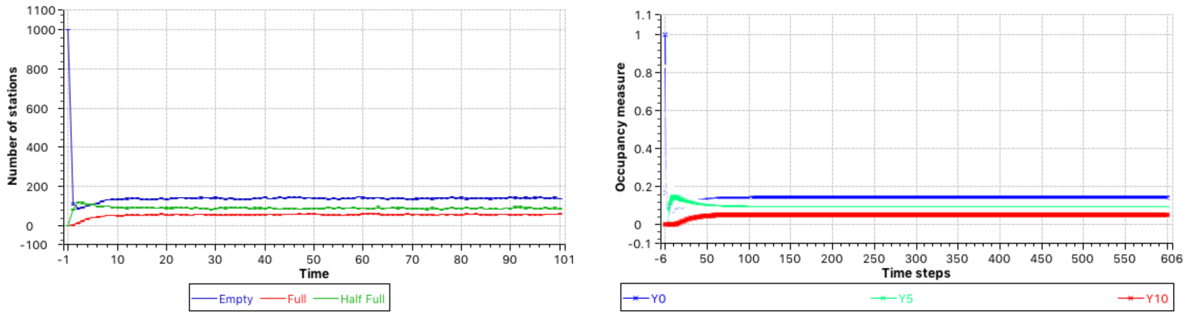


Figure 23: CARMA simulation (left) and FlyFast (right) results for $N = 1000$, $K = 10$, $s = 5$ and all stations initially empty.

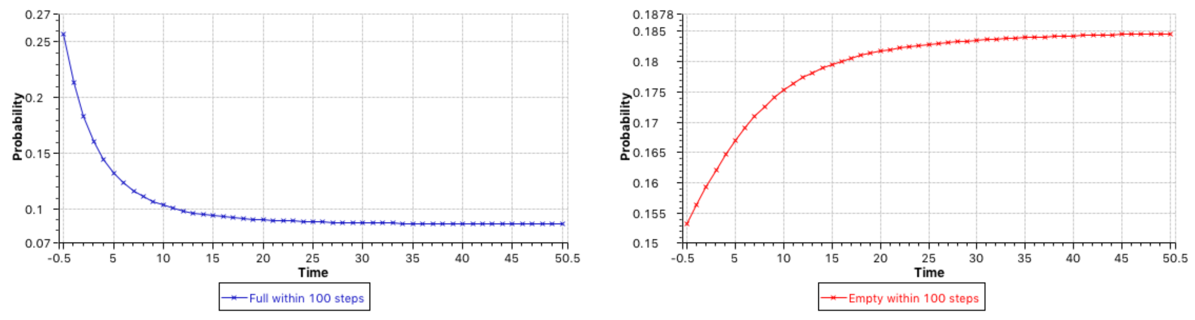


Figure 24: Probability of the individual station to become full (left) and to become empty (right) within 100 steps starting from the individual station in state Y0, respectively Y10, and the overall system empty initially; the formulas are evaluated at times ranging from 0 to 50.

time steps[5] and for initial times ranging from 0 to 50. The former is expressed by $\mathscr{P}_{=?}(\ \texttt{true}\ \mathscr{U}^{\leq 100}\ Y10\ )$. It is assumed that the overall system at time 0 is empty, i.e. all stations other than the individual one are empty. The formula for the latter situation (from full to empty) is similar. It is interesting to see that for time step 0 the probability of an individual full station to become empty is lower than that of an empty station to become full. This situation is reversed when the same properties are checked for an example starting from the overall system in time step 20. This shows the dependence of the properties (or probabilities) on the time at which they are checked, i.e. the time-inhomogeneity of the stochastic model of the individual station. For more details we refer the reader to [20].

6 Concluding Remarks

In this deliverable we reported the work done in the last reporting period of the WP4 of the QUANTICOL project. In this period many efforts have been made to improve the *usability* of CaSL, the CARMA Specification Language. In this deliverable we presented some of the new features included in CaSL to ease the use of the language by users who are not familiar with formal languages. One of the key features is the possibility to include definitions of *spatial models* in which components operate. We also presented some case studies that aim to show how CaSL can be used to model typical scenarios of CAS and also in the more general setting of an agent-based food security model.

We also presented the set of tools we have developed to support CARMA modelling, the CARMA tool set. The new features introduced in the CARMA Eclipse Plug-in to simplify the analysis workflow of CaSL models were illustrated. These features include the integration with other tools developed in the project. We also presented the CARMA Graphical Plug-in, that allows a modeller to work graphically when specifying the spatial aspects of a model, and a *command line interface* to provide more flexible support to users at the stage of model deployment and experimentation.

To guide less experienced users between the different types of modelling that can be done with CaSL, we defined an analysis pathway which can guide them through the model development process, with different levels of support provided for different stages of model development. In particular, strong static checking during model design helps the modeller rapidly construct a model for initial exploration of the system under study. In contrast the lightweight support of the command line interface grants more freedom for complex experiments, possible executed in parallel.

Finally, we use two case studies borrowed from the *smart cities scenario* to show this *pathway in action* and how different tools developed in QUANTICOL can be integrated to perform analysis of CAS.

[5] We remind the reader that this corresponds to $100/6 = 17$ time units in the original CARMA model, due to time-rescaling by factor $q = 6$.

Relationship to other work packages. The work presented in this deliverable is strongly related to the work of several other work packages, specifically WP2, WP3 and WP5:

WP2 Definitions of included in CaSL(see Section 2.2) as well as the CARMA Graphical Plug-in (presented in Section 3.2) have been developed in collaboration with WP2 where different approaches for modelling spatial informations have been considered; the internal report IR2.1 "*Language constructs for spatial representation*" played a key role in shaping our ideas.

WP3 Some of the verification techniques developed in Task 3.1 of WP3, supporting model checking of spatio-temporal aspects of system behaviour, have been used to analyse CARMA models specified with CaSL. Two examples, taken from our smart urban transportation case study, have been presented in some detail in Section 5.

WP5 All the CARMA Tool Suite has been developed in strong collaboration with WP5. Moreover, an effort has been made to integrate this tool suite with some of the other tools in the QUANTICOL software tool suite: MULTIVESTA, FLYFAST and jSSTL.

Foresight. The main effort of WP4 has been the definition of a new specification language, CARMA, that is based on a formal semantics and can be used to support quantitative analysis of CAS. The tool suite built around CARMA provides a promising environment that can be used to support specification and analysis of a class of systems that exceeds that of CAS.

The work done in this WP with the development of the CARMA tool suite will continue. New tools and features will be integrated in order to improve the usability of the tools. Moreover, we plan to use CARMA and its tools also outside the context of CAS.

Another line of work will be centred on the completion of a fluid semantics for CARMA. This is far from straightforward because the inclusive nature of broadcast communication is counter to the usual assumptions required in order to get convergence results for fluid approximations. Thus we have been studying the problem of a simpler language so that we can focus on the core aspects of the problem, and have now arrived at a suitable definition for a fluid approximation. In future work we will extend the work to other features of CARMA (i.e. considering attributes etc.) to provide a fluid semantics for CARMA and incorporate it into the tool. This will enable the use of specification and analysis techniques developed in WP1 to enlarge the class of analysis that can be performed on CARMA models, and the use of ODE-based model reduction techniques from WP3 to improve the scalability.

Another aspect that will be investigated is the definition of *equivalences* for CARMA. The notion of equivalence is fundamental to process algebras. Typically a semantic equivalence, based on the notion of bisimulation, is developed and in the case of a language with an underlying Markovian semantics the equivalence is related to aggregation through lumpability. Thus as well as interest for model comparison, equivalences for stochastic process algebras also play a role in model reduction. We have been investigating the definition of a suitable notion of bisimulation for CARMA. This is a difficult problem because of the involvement of the environment in CARMA models. Thus a promising direction seems to be identifying components that have the same behaviour when considered in the same environment, or identifying environments which cannot distinguish valid components. In particular in the ongoing work on equivalence relations for PCTMCs, we are considering mean field equivalences which identify components that behave in a stochastically identical way when placed in the same collective and the same environment.

Almost in the same direction, we will also consider the study of the appropriate approach for specifying and verifying properties of CARMA systems. Indeed, while in QUANTICOL properties are mainly focussed on *spatio-temporal* ones, other aspects could also be of interest. In particular, one could be interested in verifying properties of a specific *single agent* when executed in a complete system. Under this perspective, properties of the complete system could be also specified as the composition of properties of single agents.

References

[1] Y. A. Alrahman, R. De Nicola, M. Loreti, F. Tiezzi, and R. 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] M. Bernardo and R. 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] S. Bharwani, M. Bithell, T. Downing, M. New, R. Washington, and G. Ziervogel. Multi-agent modelling of climate outlooks and food security on a community garden scheme in Limpopo, South Africa. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 360:2183–2194, 2005.

[4] H. Bohnenkamp, P. D'Argenio, H. Hermanns, and J.-P. Katoen. MODEST: A compositional modeling formalism for hard and softly timed systems. *IEEE Trans. Software Eng.*, 32(10):812–830, 2006.

[5] L. Bortolussi, R. D. Nicola, V. Galpin, S. Gilmore, J. Hillston, D. Latella, M. Loreti, and M. Massink. CARMA: Collective adaptive resource-sharing markovian agents. In *Proc. of the Workshop on Quantitative Analysis of Programming Languages 2015*, volume 194 of *EPTCS*, pages 16–31, 2015.

[6] V. Ciancia, D. Latella, and M. Massink. On-the-fly mean-field model-checking for attribute-based coordination. In A. Lluch-Lafuente and J. Proença, editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2016.

[7] M. Daskin. A maximum expected covering location model: Formulation, properties and heuristic solution. *Transportation Science*, 17:48–70, 1983.

[8] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7, 2014.

[9] C. Feng and J. 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 *Lecture Notes in Computer Science*, pages 265–280. Springer, 2014.

[10] C. Feng and J. Hillston. Speed-up of stochastic simulation of PCTMC models by statistical model reduction. In M. Beltrán, W. J. Knottenbelt, and J. T. Bradley, editors, *Computer Performance Engineering - 12th European Workshop, EPEW 2015, Madrid, Spain, August 31 - September 1, 2015, Proceedings*, volume 9272 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2015.

[11] C. Feng, J. Hillston, and V. Galpin. Automatic moment-closure approximation of spatially distributed collective adaptive systems. *ACM Trans. Model. Comput. Simul.*, 26(4):26:1–26:22, 2016.

[12] C. Fricker and N. Gast. Incentives and redistribution in homogeneous bike-sharing systems with stations of finite capacity. *EURO J. Transport. Log.*, pages 1–31, 2014. Online version at http://arxiv.org/abs/1201.1178 accessed Nov. 14, 2016.

[13] V. Galpin. Modelling ambulance deployment with CARMA. In A. Lluch Lafuente and J. Proença, editors, *18th IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION 2016)*, pages 121–137. Springer, 2016.

[14] V. Galpin and D. Reijsbergen. CARMA for food security: a progress report. Technical Report TR-QC-03-2017, QUANTICOL, 2017. `http://www.quanticol.eu`.

[15] H. Hermanns and M. 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.

[16] J. Hillston. *A Compositional Approach to Performance Modelling*. CUP, 1995.

[17] C. Jagtenberg, S. Bhulai, and R. van der Mei. An efficient heuristic for real-time ambulance redeployment. *Operations Research for Health Care*, 4:27–35, 2015.

[18] M. John, C. Lhoussaine, J. Niehren, and A. M. Uhrmacher. The attributed Pi calculus. In *Proc. of Computational Methods in Systems Biology*, volume 5307 of *LNBI*, pages 83–102, 2008.

[19] D. Latella, M. Loreti, and M. Massink. On-the-fly fluid model checking via discrete time population models. In M. Beltrán, W. J. Knottenbelt, and J. T. Bradley, editors, *Computer Performance Engineering - 12th European Workshop, EPEW 2015, Madrid, Spain, August 31 - September 1, 2015, Proceedings*, volume 9272 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2015.

[20] D. Latella, M. Loreti, and M. Massink. On-the-fly PCTL fast mean-field approximated model-checking for self-organising coordination. *Sci. Comput. Program.*, 110:23–50, 2015.

[21] M. Loreti and J. Hillston. Modelling and analysis of collective adaptive systems with CARMA and its tools. In M. Bernardo, R. D. Nicola, and J. Hillston, editors, *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016, Advanced Lectures*, volume 9700 of *Lecture Notes in Computer Science*, pages 83–119. Springer, 2016.

[22] L. Nenzi and L. Bortolussi. Specifying and monitoring properties of stochastic spatio-temporal systems in signal temporal logic. In *8th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2014, Bratislava, Slovakia, December 9-11, 2014*, 2014.

[23] L. Nenzi, L. Bortolussi, V. Ciancia, M. Loreti, and M. Massink. Qualitative and Quantitative Monitoring of Spatio-Temporal Properties. In E. Bartocci and R. Majumdar, editors, *Runtime Verification*, number 9333 in Lecture Notes in Computer Science, pages 21–37. Springer International Publishing, 2015.

[24] V.Galpin, N. Zoń, P. Wilsdorf, and S. Gilmore. Mesoscopic modelling of pedestrian movement using CARMA and its tools, 2017. Submitted for journal publication.

[25] G. Ziervogel, M. Bithell, R. Washington, and T. Downing. Agent-based social simulation: a method for assessing the impact of seasonal climate forecast applications among smallholder farmers. *Agricultural Systems*, 83:1–6, 2005.

[26] N. Zoń, V. Galpin, and S. Gilmore. Modelling movement for collective adaptive systems with CARMA. In *Proceedings of the Workshop on FORmal methods for the quantitative Evaluation of Collective Adaptive SysTems, (FORECAST@STAF 2016)*, EPTCS 217, pages 43–52, 2016.

[27] N. Zon, S. Gilmore, and J. Hillston. *Rigorous Graphical Modelling of Movement in Collective Adaptive Systems*, pages 674–688. Springer International Publishing, 10 2016.

A CaSL: a detailed description

Each CaSL specification provides definitions for: structured *data types* and related *functions*; prototypes of *components* occurring in the system; *systems* composed by collective and environment; and the *measures*, that identify the relevant data to observe during simulation runs.

A.1 Data types

Four basic types are natively supported in CaSL: `bool`, for booleans, `int`, for integers, `real`, for real values, and `location` for *spatial locations*. To model complex structures, it is often useful to introduce custom types. In CaSL two kind of custom types can be declared: *enumerations* and *records*. Finally, data types *sets* and *lists* are used to represent *collections* of homogeneous data.

Enumerations. As in many 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 its 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 designer as having any particular concrete representation. The syntax to declare a new *enumeration* is:

enum $\underline{<\text{name}>}$ = $\underline{<\text{name}_1>}$, . . . , $\underline{<\text{name}_n>}$;

where $\underline{<\text{name}>}$ is the name of the declared enumeration while $\underline{<\text{name}_i>}$ are its value names. We adopt the convention that the names of all the enumeration values should only be composed of capital letters.

Example 1. Enumerations can be used to define predefined sets of values that can be used in a specification. For instance, one can introduce an enumeration to identify the possible four directions of a movement:

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

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

record $\underline{<\text{name}>}$ = [$\underline{<\text{name}_1>}$ $\underline{<\text{field}_1>}$, . . . , $\underline{<\text{name}_n>}$ $\underline{<\text{field}_n>}$];

Each field has a type $\underline{<\text{type}_i>}$ and a name $\underline{<\text{field}_i>}$: $\underline{<\text{type}_i>}$ can be either a built-in type or one of the new declared types in the specification; $\underline{<\text{field}_i>}$ can be any valid identifier.

Example 2. Records can be used to model structured elements. For instance, a direction in a 2D grid can be rendered via a record as follows:

```
record direction = [ int dx, int dy];
```

Collections. Sometimes, when a CAS system is modelled, it is useful to consider collections of homogeneous data. In CaSL two kinds of collections are supported: *sets* and *lists*.

A *set* is, as usual, a collection that does not contain duplicated elements. A set containing elements of type `type` is declared as `set <type>`.

A *list* consists of a sequence of elements of the same type. Differently from a *set*, a *list* can contain multiple copies of the same element. Moreover, elements can be retrieved by their index, that is the position in the sequence. A list containing elements of type `type` is declared as `list <type>`.

`||`	Disjunction/Set intersection
`&&`	Conjunction/Set union
`==,!=,<,<=,>,>=`	Comparison
`+`	Sum/List concatenation
`-`	Difference/Set difference
`*`	Multiplication
`/`	Division
`%`	Modulo
`!`	Negation
`-`	Minus
`in`	Belongs to
`( ? : )`	Conditional operator

Table 1: Basic expression operators

A.2 Expressions

In CaSL an expression is built starting from *references*, *literals*, *operators* and *function invocations*[6]. The list of CaSL operators is reported in Table 1 in the order of priority. Some of these operators are *overloaded* and can be used with different meanings depending on the type of their arguments.

In an expression four kinds of literals can occur: *none*, *boolean*, *integer* and *reals*. The syntax of these constant values is standard:

None literal: is the value `none` used to refer to an undefined value;

Boolean literal: one of the two constant values `true` and `false`;

Integer literal: a base 10 integer represented by a non empty sequence of digits[7], i.e. an element in $(0..9)^{+}$;

Real literal: a floating point value represented via the standard syntax $(0..9)^{*}.(0..9)^{+}$.

References. An expression can contain references to (local and global) *attributes*, *constants*, *parameters* or *variables*. The scope for these references will be clarified later in this document. Attribute references can be prefixed with the keywords `my`, `global`, `sender` or `receiver`, to specify the *store* used to evaluate an attribute.

Each reference is associated with a *data type*. This binding can be done either explicitly or implicitly. For instance, the parameters of a function are explicitly typed. While the type of other *references*, such as for instance *constants*, is automatically inferred from their definition.

Arithmetic expressions. Integer and real expressions can be combined by using the standard arithmetic operators: `+`, `-`, `*` and `/`. Operator `%` can be used to compute the modulo of two integer expressions. It is required that the two arguments of all these operators must have the same type (i.e. either `int` or `real`). This means that the expression `1+1.0` is not well typed (and an error is reported in the editor).

To convert `int` values into `real` (and vice versa), the two cast operators `int(e)` and `real(e)` can be used. For instance, `real(2)` is evaluated to the `real` value `2.0`, while `int(2.7)` is evaluated to the `int` value `2`.

CaSL expressions are equipped with a set of constants and built in functions. The constants `E` and `PI` can be used to refer to the Euler's number e and the value π, while constants `MAXINT` and `MININT` are used to refer to the max and min storable integers. Similarly, `MAXREAL` and `MINREAL` identify the max and min storable reals. The

[6] The complete syntax of CaSL expressions is reported in Appendix B.

[7] We let $(0..9)$ denote the set of digit from 0 to 9; X^{*} denote any (possibly empty) sequence of elements in X; while X^{+} denote any non empty sequence in X^{*}.

`abs( e )`	The absolute value of `e`
`acos( e )`	The arc cosine of `e`; the returned angle is in the range `0.0` through `PI`
`asin( e )`	The arc sine of `e`; the returned angle is in the range `-PI/2` through `PI/2`
`atan( e )`	The arc tangent of `e`; the returned angle is in the range `-PI/2` through `PI/2`
`atan2( e1 , e2 )`	The angle θ from the conversion of rectangular coordinates `(e1, e2)` to polar coordinates (r, θ)
`cbrt( e )`	The cube root of `e`
`ceil( e )`	The smallest (closest to negative infinity) double value that is greater than or equal to `e` and is equal to a mathematical integer
`cos( e )`	The trigonometric cosine of angle `e`
`exp( e )`	The Euler's number e raised to the power of `e`
`floor( e )`	The largest (closest to positive infinity) double value that is less than or equal to `e` and is equal to a mathematical integer
`log( e )`	The natural logarithm (base e) of `e`
`log10( e )`	The base 10 logarithm of `e`
`max( e1 , e2 )`	The max value between `e1` and `e2`
`min( e1 , e2 )`	The min value between `e1` and `e2`
`pow( e1 , e2 )`	The value `e1` raised to the power of `e2`
`sin( e )`	The trigonometric sine of `e`
`sqrt( e )`	The correctly rounded positive square root of `e`
`tan( e )`	The trigonometric tangent of `e`

Table 2: Built-in functions

complete list of built-in functions is reported in Table 2. Finally, the special name `now` can be used to refer to the *current time* in the simulation.

Boolean and conditional expressions. Boolean expressions are built starting from constants `true`/`false`; from the comparison of two values `e1` *op* `e2` (where *op* is one among `==`, `!=`, `<`, `<=`, `>` or `>=`); or from the *belongs to* expression `e1 in e2`. The first two cases are standard, while the latter is used to check whether the evaluation of `e1` belongs to the *collection* resulting from the evaluation of `e2`.

Standard boolean operators `||`, `&&` and `!` can be used to compute *disjunction*, *conjunction* and *negation* of a boolean expression.

Boolean expressions can be used directly in the conditional operator `(e1 ? e2 : e3 )`. When evaluated, this expression is equal to `e2` when `e1` is `true` otherwise it is evaluated to `e3`. As expected, expression `e1` must have type `bool` while `e2` and `e3` must have the same type.

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

[$field_1$:= $expression_1$, ... , $field_n$:= $expression_n$]

Example 3. If we consider the record definition of Example 2, the instantiation of a direction has the following form:

```
[ dx := 0 , dy := 0 ]
```

Given a reference with a record type, each field can be accessed using the *dot* notation:

$ref\,.\,field_i$

Operations on collections. Collections can be created either by enumerating their elements, or by creating an empty collection of a given type. In the first case, the special brackets `[: :]` and `{: :}` are used as delimiters of a *list* and a *set*, respectively. Hence, `[: 5 , 3 , 4 :]` identifies a list of three elements, while `{: 2 , 1 :}` indicates the set $\{1,2\}$.

To create an empty list or an empty set the functions `newList` and `newSet` can be used as well. Both functions take as argument the type of elements contained in the created collection. For instance, `newSet( int )` can be used to create a set that can contain integers, while `newList( list<int> )` creates an empty list that in turn contains lists of integers.

If `e1` and `e2` are two expressions of type `set<t>`, for some type `t`, `e1||e2`, `e1&&e2` and `e1-e2` denote the union, the intersection and the difference between `e1` and `e2` respectively.

If `e1` and `e2` are two expressions of type `list<t>`, for some type `t`, `e1+e2` represent the concatenation of the two lists.

If `e` is a collection, `size(e)` is used to compute the number of elements stored in the collection.

Elements in a list can be accessed via their index: if `e1` is a list of type `list<t>` and `e2` is an integer expression with value i, `e1[e2]` indicates the element at position i in `e1`. Moreover, functions `head` and `tail` can be used to retrieve the *head* and the *tail* of a list, respectively.

The function `exist` can be used to check whether there exists an element in a collection that satisfies a given predicate. The function `exist` takes two parameters: one collection and a boolean expression. The latter may contain the special symbol `@`. This is used as a placeholder replaced by the elements in the collection when the predicate is evaluated. The function application `exist( e1 , e2 )` evaluates to `true` if there exists in `e1` an element `x` such that `e2[x/@]` is true. For instance, if `e1` is a collection of integers, `exist( e1 , @>5 )` is true if and only if there exists an element in `e1` that is greater than `5`. The function `forall` is similar, `forall( e1 , e2 )` is true if and only if all the elements in `e1` satisfy the predicate `e2`.

It is possible to select all the elements satisfying a given predicate by using the function `filter`. If `e1` is a collection and `e2` a boolean expression containing the placeholder `@`, `filter( e1 , e2 )` is the collection that contains only the elements satisfying `e2`. For instance, `filter( {: 2 , 4 , 6 :} , @<3 )` will return the set `{: 2 :}`.

Sometimes it is also useful to manage elements in a collection in an aggregated way. For this reason, the function `map` allows the creation of a new collection that is obtained from another one by applying a given function. This function is defined as an expression that contains the placeholder `@`. The expression `map( [: 1 , 2 , 3 :] , pow( @ , 2 ))` is equivalent to `[: 1 , 4 , 9 :]`.

To improve readability of expressions, all the functions on collections can be used in *infixed* form. This means, for instance that `map( e1 , e2 )` can be expressed as `e1.map( e2 )`.

Random expressions. To model random behaviour, CaSL expressions provide different mechanisms for sampling random values. A first mechanism to include random values is to use the expression `RND`. When this expression is evaluated, this term is replaced with a value that is randomly selected in the interval $[0,1)$.

To sample values according to a *normal distribution*, we use the expression `NORMAL( e1 , e2 )`. In this case, the next value is randomly selected according to a distribution with mean `e1` and variance `e2`.

To select values from a collection, the function `select( e1 , e2 )` can be used. There, `e1` is a collection, while `e2` is an expression (containing the placeholder `@`) that is used to compute the probability to select each element in `e2`. For instance, in the expression `select( {: 1 , 2, 3, 4, 5 :} , @ )` a value $i \in \{1,2,3,4,5\}$ is selected with probability $\frac{i}{30}$. While in `select( {: 1 , 2, 3, 4, 5 :} , 1 )` each value is selected with the same probability $\frac{1}{5}$.

Another statement for uniform selection of elements is `U( e1 , ... , en )`. This is used to uniformly select one of the values `e1`,..., `en`.

In the remainder of this document we say that an expression is *random* if it contains one of the random expressions described above. It is *deterministic* if it is not *random*.

A.3 Constants and Functions.

A CaSL model can contain *constant* and *function* declarations.

A constant can be declared by using the following syntax:

```
const <name> = <exp>;
```

where, `<name>` is the constant name while `<exp>` is the expression defining the constant value. We can notice that constants are not explicitly typed. This because the type of a constant is not declared but inferred directly from the assigned expression`<exp>`.

A constant can be used to represent some parameters in a model like, for instance, the constant `SIZE`:

```
const SIZE =  1000;
```

after this declaration, the name `SIZE` can be used as an integer of value 1000.

Function declaration has the following syntax:

```
fun <type> <name>(  <type_1> <arg_1>  ,... , <type_k> <arg_k> ) <body>
```

where `<name>` is the function name, each `<arg_i>` is the name of parameter *i* of type `<type_i>`, while `<type>` is the type of of the value returned by the function. Finally, `<body>` contains the statements (denoted by `<stm>`) used to compute the returned value.

Possible statements that can be used in the body of a function are:

Variable declaration: This is used to declare a local variable:

```
<type> <name> = <exp>;
```

The command above declares a new variable `<name>` of type `<type>` assigned to value `<exp>`. Assignment is optional and can be omitted. Standard scoping rules are applied to a variable declaration: it can be used only by next statements that are inside that function or block of code.

Assigment: Declared variables can be assigned by using the standard assignment command:

```
<ref> = <exp>;
```

where `<ref>` is a reference generated by the following syntax:

```
<ref> ::= <name> | <ref>.<field> | <ref>[ <exp> ]
```

where `<name>` is the name of either a local variable or one of the function parameters, `<field>` is the name of a field, and `<exp>` is an expression (of type `int`).

If-then-else: This is the standard *if-then-else* construct, having the syntax:

```
if (<exp>) <body>
else <body>
```

For iteration: Two kinds of iterators can be used in a function definition.

The first is the usual *for-loop* that can be used to iterate a given statement while a local variable is incremented until a given value is reached:

```
for <name> from <exp_1> by <exp_2> to <exp_3>
    <stm>
```

Above `<name>` is the name of the variable (of type `int`) that initially is assigned to `<exp_1>` and is incremented by `<exp_2>` at the end of each iteration. The loop terminates when the variable reaches value `<exp_3>`. Increment (`by` $\underline{< \text{exp} >}$) is optional and can be omitted. In this case the standard increment step 1 is used.

The other iterator implements a *for-each* and can be used to iterate over all the members of a collection. The syntax of this statement is the following:

```
for <name> in <exp> <stm>
```

This means that for each value `v` in the collection `<exp>`, statement `<stm>` is executed with value `v` assigned to variable `<name>`.

Return: This is the standard statement used to return a value after a function call:

```
return <exp>;
```

It is required that the type of expression `<exp>` is the same as the one declared in the function declaration.

Block: This consists of a sequence of statements between brackets:

```
{
    <stm>*
}
```

Example 4. The following function can be used to sum all the values in a collection of integers:

```
fun int sumAll( list<int> c ) {
    int sum = 0;
    for v in c {
        sum = sum + v;
    }
    return sum;
}
```

A.4 Space models

The space in which a system operates can be defined as a graph in which edges have labels. For example we can have a road lane with attribute $buses = true$, which means that buses can travel on it.

Each space is associated with a universe - a collection of nodes along with information about their location in space. This can be, for example, a grid, along with an indexing system, or a bounded plane with a coordinate system. The `nodes` block specifies which subset of nodes from the universe is used in the model. The `connections` block contains the specification of how these nodes are connected to each other. The `areas` block allows the user to define attributes associated with subsets of nodes belonging to the space.

The syntax for a graph definition is:

```
space <name>(<type_1> <name_1>,...,<type_n> <name_n>) {
   universe universe_def;
   nodes {
      <node_def>*
   }
   connections {
      <connection_def>*
   }
   areas {
      <area_def>*
   }
}
```

Above `<name>` indicates the graph name, while each `<type_i> <name_i>` is a parameter that can be used to build the graph. Parentheses can be omitted when the list of parameters is empty.

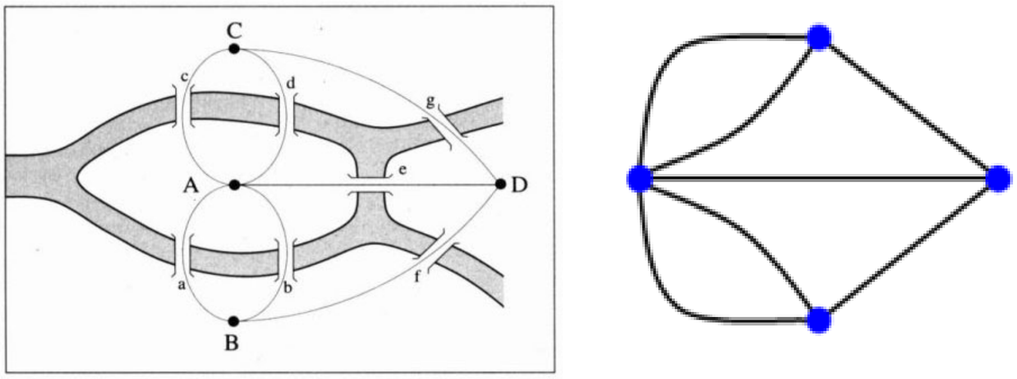


Figure 25: Königsberg Graph

Example 5. The declaration of a grid graph can have the following syntax:

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

Universe. The universe is defined as a sequence of typed fields:

```
universe < <type_1> <name_1>, ..., <type_n> <name_n> >;
```

where `type_i` is the type of the field, while `name_i` is its name.

Example 6. The grid can be indexed by two numbers:

```
space grid(int height, int width){
     universe <int x, int y>;
}
```

The universe is optional and can be omitted if it is not needed in the model.

Nodes. The nodes in the space, namely the vertices in the considered graph, are declared via the `<node_def>` statement. This statement can be used to assign a (optional) name to each node and the corresponding position in the coordinate system of the universe:

```
<name_1>[e_1,...,e_n];
```

Example 7. The declaration of the graph corresponding to the classical Königsberg bridge scenario of Fig.25 is:

```
space Konigsberg {
   nodes {
      A;
      B;
      C;
      D;
   }
   ...
}
```

Example 8. In a space with a universe, the same example can be of this form:

```
space Konigsberg() {
  universe<int x, int y>;
   nodes {
      A[0,0];
      B[0,-1];
      C[0,1];
      D[1,0];
   }
   ...
}
```

When we have models with a large number of locations it is not convenient to explicitly list all the vertices in a graph. For this reason iteration and selection statements can be used to declare multiple nodes:

```
for x from e_1 by e_2 to e_3 {
    <node_def>
}

for x in e {
    <node_def>
}

if e {
    <node_def>
} else {
    <node_def>
}
```

Example 9. Locations of the grid model considered earlier can be declared as follows:

```
space grid( int width , int height ) {
   universe <int x, int y>;

   nodes {
      for x from 0 to width {
         for y from 0 to height {
            [x,y];
         }
      }
   }
   ...
}
```

Example 10. For a grid with an origin at its centre the nodes are defined in a similar way:

```
space centerdGrid(int width, int height) {
   universe <int x, int y>;

   nodes {
        for x from -width/2 to width/2{
           for y from -height/2 to height/2{
               [x, y];
           }
        }
   }
   ...
}
```

Connections. In the `connections` block we can list the edges of our model. An edge can be declared either explicitly or via predicates over the position of a node. In the first case vertices are explicitly referenced in the edge declaration:

```
//Directed edge;
name[e1,...,en] -> name[e1,...,en] { label_1=w_1,...,label_n=w_n };

//Undirected edge;
name[e1,...,en] <-> name[e1,...,en] { label_1=w_1,...,label_n=w_n };
```

Each edge is equipped with a set of *features*. For instance, the *weight* of the edge. The braces around the labels can be omitted in the case that there is only a single label.

Example 11. In the case of Königsberg bridge example, the edge declaration can be:

```
space Konigsberg {

   nodes {
      A;
      B;
      C;
      D;
   }

   connections {
      A <-> B { w=1.0 };
      A <-> C { w=2.0 };
      A <-> C { w=1.0 };
      A <-> D { w=1.0 };
      D <-> C { w=3.0 };
      D <-> B { w=3.0 };
   }

   ...
}
```

As for `nodes`, also in the `connections` block iteration and selection blocks can be used.

Example 12. The edges of the grid model considered before can be declared as follows:

```
space grid( int width , int height ) {
    universe <int x, int y>;

    nodes {
        for x from 0 to width {
            for y from 0 to height {
                [x,y];
            }
        }
    }

    connections {
        for i from 0 to width-1 {
            for j from 0 to height-1 {
                [i,j] <-> [i+1,j] { weight=1.0 };
                [i,j] <-> [i,j+1] { weight=1.0 };
            }
        }
    }
    ...
}
```

Areas. An area is a collection of nodes. Nodes can be associated with an area via enumeration possibly relying on iteration and selection statements.

```
<area_name> {
  <name>[e1,...,en];
}
```

Example 13. In the grid the identified sets of locations could be `corner`, `border` and `diagonal` as shown below:

```
space grid( int width , int height ) {
   universe <int x, int y>;
   nodes {
      for x from 0 to width {
         for y from 0 to height {
            [x,y];
         }
      }
   }
   connections {
      [x,y]: x<width-1 <-> [x+1,y]: weight=1.0;
      [x,y]: y<height-1 <-> [x,y+1]: weight=1.0;
   }
   areas {
      corner {
         [0,0];
         [width-1,0];
         [0,height-1];
         [width-1,height-1];
      }
      diagonal {
         for i from 0 to min(width,height) {
            [i,i];
         }
      }
      border {
         for i from 0 to width {
            [i,0];
            [i,height-1];
         }
         for j from 0 to height {
            [0,j];
            [height-1,j];
         }
      }
   }
}
```

Location expressions: a *location expression* is an expression of type `location` that is evaluated to a *node* in a space model. Syntax of a location is:

```
<name_1>[e_1,...,e_n]
```

To access the data related to space the following expressions can be also used used (below `l` is an expression of type `location`):

- `l.post`: indicates the locations in the postset of location `l`;
- `l.pre`: indicates the locations in the preset of `l`;
- `l.name`: indicates the location name;
- `l.xi`: refers to the element `xi` of the universe point of `l`;

- `l.area`: is a boolean expression that can be used to check if location `l` is part of the area `area`;
- `locations`: indicates the set of all locations;
- `area_name`: is used to access the set of locations with label `area_name`;
- `l.outgoing(e)`: get all the edges exiting from `l` and reaching `e`, the parameter is optional and when omitted all the edges exiting from `l` are returned;
- `l.incoming(e)`: get all the edges entering in `l` and starting from `e`, the parameter is optional and when omitted all the edges entering in `l` are returned;
- `edgeValues(e1,v,e2)`: get all the values associated with label `v` in edges connecting `e1` to `e2`.

A.5 Component prototype.

A *component prototype* provides the general structure of a component that can be later instantiated in a CaSL 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> <name_1>,...,<type_n> <name_n>) {
    store {
        (<type> <attribute_name> = <expr>;)*
    }
    behaviour {
        <pdef>*
    }
    init { <pname_1>|...|<pname_n> }
}
```

Each component prototype has a possibly empty list of arguments. As expected, 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 declaration of the actual type of an attribute is optional, since the type of an attribute is inferred from the expression providing its initialisation value. The special attribute `loc`, having type `location`, is always available in any store. An appropriate value is assigned to this attribute when the component is instantiated.

Block `init` is used to specify the initial behaviour of a component. It consists of a sequence of terms `pname_i` referring to processes defined in block `behaviour` or a process in the argument list.

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

```
<pname> = <pbody>;
```

where `<pname>` is the process name while `<pbody>` is the *process body* and can be one of the following:

Choice: `<pbody_1> + <pbody_2>`

Guard: `[<expr>]<pbody>`

Action: `<act>.<proc>`

Above `<pbody_1> + <pbody_2>` indicates the *choice* between the behaviours `<pbody_1>` and `<pbody_2>`. The guard `[<expr>]<pbody>` indicates that behaviour `<pbody>` is enabled when boolean expression `<expr>` evaluates to `true`. Finally, `<act>.<proc>` represents a process that performs action `<act>` and then evolves to `<proc>`. The latter represents the behaviour activated after the action execution and can be a process name `<pname>` or one of the process constants `nil` or `kill`. These two values represent the *inactive process* or the process that *destroys* the component. When process `kill` is activated the hosting component is removed from the system.

In CaSL, as in CARMA, two kinds of synchronisations are provided: *broadcast synchronisation*, and *unicast synchronisation*. The first one represents a *one-to-many* interaction, while the second one is the usual *one-to-one* interaction. In both the cases the senders and the receivers select their counterpart(s) in the communication via an *activity* and a predicate (that is a boolean expression) that filters possible receivers/senders depending on the values of their attributes. The execution of an action may trigger some updates on the store.

Broadcast output. The syntax of broadcast output is the following:

```
<name>*[ g ]< e_1 , ... , e_n >{ <update> }
```

Above `<name>` is the activity name, `g` is the boolean guard expression used to select receivers, `e_1 , ... , e_n` is the tuple of values sent with the action, and `<update>` is the update performed after the action execution. The latter is a sequence of assignments of the form:

```
a1 = exp1;
...
an = expn;
```

each `ai` is the attribute to update, while `expi` is the new value assigned to `ai`.

In `g` attributes prefixed with `my` will be evaluated with the local store. For instance,

```
forward*[ my.group == group ]< v >{ my.counter = my.counter + 1; }
```

is used to send with activity `forward` the value `v` to the components having the attribute `group` equal to `my.group`, that is the one evaluated locally. After the action is executed, attribute `counter` is incremented by 1. Note that this action can be executed even if there are no components ready or able to execute a complementary action.

It is required that the guard `g` and all the sent values `ei` must be *deterministic expressions* (i.e. without random values).

Broadcast input. The syntax of broadcast input is similar:

```
<name>*[ g ]( x_1,...,x_n ){ <update> }
```

However, in this case the guard `g` can also contain references to received variables `xi`. For instance:

```
forward*[ my.value < x ]( x ){ my.value = x; }
```

guarantees that the synchronisation occurs only when the received value is greater than attribute `value`. When such a value is received, attribute `value` is updated.

Unicast output. The syntax of unicast output is the following:

```
<name>[ g ]< e_1 , ... , e_n >{ <update> }
```

Differently from *broadcast output*, this action may be executed only when a complementary input is executed at the same time. Like for *broadcast output*, the guard `g` and all the sent values `ei` must be *deterministic expressions*.

Unicast output. The syntax of unicast input is the following:

```
<name>[ g ]( x_1,...,x_n ){ <update> }
```

A.6 System definitions.

A system definition consists of a space instantiation and two blocks, namely `collective` and `environment`, that are used to declare the collective in the system and its environment, respectively:

```
system name {
  space <name>( e1 ,... ,en )
  collective {
    inst_stmt
  }
  environment { ···
  }
}
```

Space instantiation is used to define the space model where components are located. This instantiation is optional can be omitted.

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

```
new name( e_1 ,... , e_n )@l<n>
```

where `name` is the name of a component prototype, `e_i` are the parameters, `l` is the (optional) location where the created component is located (and that will be assigned to attribute `loc` having type `location`), and `n` is the integer expression identifying the multiplicity (i.e. the number of copies) of the created component.

However, in a system a large number of collectives may be present. For this reason, following the same approach used to create spatial models, we can use *for-loops* and *selection* constructs for instantiating multiple components.

The syntax of a block `environment` is the following:

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

The block `store` defines the *global store* and has the same syntax as the similar block already considered in the component prototypes.

Blocks `prob` and `weight` are used to compute the probability to receive a message. The syntax of `prob` and `weight` is the following:

```
prob { ···
    <act> { <body>
    }
    ...
    default { <body>
    }
}

weight { ···
    <act> { <body>
    }
    ...
    default { <body>
    }
}
```

In the above, `<act>` denotes the action used to interact while `<body>` defines the function used to compute the probability/weight of the considered action. There attributes of sender and receiver are referred to using `sender.a` and `receiver.a`, while the values published in the global store are referenced using `global.a`.

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 { ···
    <act> { <body>
    }
    ...
    default { <body>
    }
}
```

Differently from `prob`, in `rate` `<body>` only refers to attributes defined in the store of the component performing the action, referenced as `sender.a`, or in the global store, accessed via `global.a`. This because the rate of an action cannot be influenced by the state of receivers.

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 { ···
    <act> { <body>
    }
    ...
}
```

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 instantiation). 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.

A.7 Measure definitions.

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

```
measure <name>( <type_1> <name_1>, ... ,<type_n> <name_n>) = <expr>;
```

Beside the expressions considered in the previous sections, the `<expr>` can contain specific expressions that can be used to extract data from the population of components.

To count the number of components in a given state, the following term 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:

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

To compute statistics about attribute values of components operating in the system one can use: `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.

B CaSL Syntax

In this section the full syntax of CaSL is reported in EBNF form. Below, we will use $\underline{<\text{symb}>}$ to denote a non terminal symbol. We also let $\underline{<\text{name}>}$ to denote a valid identifier. Moreover, we let $\underline{<\text{int}>}$ and $\underline{<\text{real}>}$ denote an integer and a real token, respectively.

Model

```
< model > = < elem >*

< elem > =
        < record > //Record definition
    |   < enum > //Enum definition
    |   < const > //Constant definition
    |   < fun > //Function definition
    |   < proc > //Process definition
    |   < comp > //Component definition
    |   < coll > //Collecive definition
    |   < space > //Space definition
    |   < sys > //System definition
    |   < meas > //Measure definition
```

Types

```
< type > =
        'int'
    |   'real'
    |   'bool'
    |   'location'
    |   'process'
    |   'list' '<' < type > '>'
    |   'set' '<' < type > '>'
    |   < name > //Reference to declared type.
;
```

Expressions

```
< exp > =
        \tsymbol{int}
    |   \tsymbol{real}
    |   'true'
    |   'false'
    |   < name >
    |   'global' '.' < name >
    |   'sender' '.' < name >
    |   'receiver' '.' < name >
    |   'my' '.' < name >
    |   'MAXINT'
    |   'MININT'
    |   'MAXREAL'
    |   'MINREAL'
    |   '@'
    |   'locations'
    |   'loc'
    |   'now'
    |   'none'
    |   'NORMAL' '(' < exp > ',' < exp > ')'
    |   < exp > '||' < exp >
    |   < exp > '&&' < exp >
    |   '!' < exp >
    |   < exp > '==' < exp >
    |   < exp > '!=' < exp >
    |   < exp > '<' < exp >
    |   < exp > '<=' < exp >
    |   < exp > '>=' < exp >
    |   < exp > '>' < exp >
    | < exp >  '+'  < exp >
    | < exp >  '-'  < exp >
    | < exp >  '*'  < exp >
    | < exp >  '/'  < exp >
    | < exp >  '%'  < exp >
    | (< exp >  '?'  < exp >  ':'  < exp >)
    | '+'  < exp >
    | '-'  < exp >
    | < exp >  '.'  < exp >
    | < exp >  '['  < exp >  ']'
    | < exp >  '.'  'pre'
    | < exp >  '.'  'post'
    | < exp >  '.'  'incoming"  '('  (< exp >)?  ')'
    | < exp >  '.'  'outgoing"  '('  (< exp >)?  ')'
    | < exp >  '.'  'source'
    | < exp >  '.'  'target'
    | < exp >  '.'  'map'  '('  (< exp >)?  ')'
    | < exp >  '.'  'filter'  '('  (< exp >)?  ')'
    | < exp >  '.'  'exist'  '('  (< exp >)?  ')'
    | < exp >  '.'  'exist'  '('  (< exp >)?  ')'
    | < exp >  '.'  'forall'  '('  (< exp >)?  ')'
    | < exp >  '.'  'select'  '('  (< exp >)?  ')'
    | '['  < name >  '='  < exp >  (','  < name >  '='  < exp >)*  ']'
    | 'U'  '('  < exp >  (','  < exp >)*  ')'
    | '('  < exp >  ')'
    | 'real'  '('  < exp >  ')'
    | 'int'  '('  < exp >  ')'
    | (< exp >)?  '['  < exp >  (','  < exp >)*']'
    | '[:'  < exp >  (','  < exp >)*  ':]'
    | '{:'  < exp >  (','  < exp >)*  ':}'
    | 'abs'  '('  < exp >  ')'
    | 'acos'  '('  < exp >  ')'
    | 'asin'  '('  < exp >  ')'
    | 'atan'  '('  < exp >  ')'
    | 'atan2'  '('  < exp >  ')'
    | 'cbrt'  '('  < exp >  ')'
    | 'ceil'  '('  < exp >  ')'
    | 'cos'  '('  < exp >  ')'
    | 'exp'  '('  < exp >  ')'
    | 'floor'  '('  < exp >  ')'
    | 'log'  '('  < exp >  ')'
    | 'log10'  '('  < exp >  ')'
    | 'max'  '('  < exp >  ','  < exp >  ')'
    | 'min'  '('  < exp >  ','  < exp >  ')'
    | 'pow'  '('  < exp >  ','  < exp >  ')'
    | 'sin'  '('  < exp >  ')'
    | 'sqrt'  '('  < exp >  ')'
    | 'tan'  '('  < exp >  ')'
    | 'pre'  '('  < exp >  ')'
    | 'post'  '('  < exp >  ')'
    | 'edgeValues'  '('  < exp >  ,  < name >  ,  < exp >  ')'
    | 'newList'  '('  < type >  ')'
    | 'newSet'  '('  < type >  ')'
    | 'size'  '('  < type >  ')'
    | 'head'  '('  < type >  ')'
    | 'tail'  '('  < type >  ')'
```

```
    | 'map' '(' < type > , < type > ')'
    | 'filter' '(' < type > , < type > ')'
    | 'exist' '(' < type > , < type > ')'
    | 'forall' '(' < type > , < type > ')'
    | 'min' '{' \tsymbol{exp} '|' \tsymbol{exp} '}'
    | 'max' '{' \tsymbol{exp} '|' \tsymbol{exp} '}'
    | 'avg' '{' \tsymbol{exp} '|' \tsymbol{exp} '}'
    | '#' '{' \tsymbol{patter} '|' \tsymbol{exp} '}'

< pattern > = '*' | < pattern > '[' '*' ']' | < pattern > '[' < name > ('|' < name >)* ']'
```

Enum definition

```
< enum > = 'enum' < name > '=' < name > (',' < name > )* ';'
```

Record definition

```
< record > = 'record' < name > '=' '[' < type > < name > (',' < type > < name > )* ']' ';'
```

Constant definition

```
< const > = 'const' < name > '=' < exp > )* ';'
```

Function definition

```
< fun > =
    'fun' < type > < name > '('
        (< type > < name > (',' < type > < name > )*)?
  ')' < stmt >

< stmt > =
    '{' < stmt >* '}'
    | 'return' < exp > ';'
    | < ref > '=' < exp > ';'
    | < type > < name > '=' (< exp >?? ';'
    | 'for' < name > 'from' < exp > ('by' < exp >)? 'to' < exp > < stmt >
    | 'for' < name > 'in' < exp > < stmt >
    | 'if' '(' < exp > ') < stmt > ('else' < stmt >)?

< ref > =
    < name >
    | < ref > '.' < name >
    | < stmt > '[' < exp > ']'
```

Processes

```
< proc > =
    'abstract' '{'
        (< pdef >)*
    '}'

< pdef > =
    < name > '=' < pexp > ';'
< pexp > =
    < pexp > '+' < pexp >
    | '[' < exp > ']' < pexp >
    | '(' < pexp > ')'
    | < act > '.' < next >

< act > =
    < act > ('*')? ('[' < exp > ']')? ('<' < exp > (',' < exp >)* '>')? ('{' (< updt >)*
         '}')?
    < act > ('*')? ('[' < exp > ']')? '(' < name > (',' < name >)* ')') ('{' (< updt >)*
         '}')?

< act > =
    ('my' '.')? < ref > = < exp > ';'
    | ('my' '.')? < ref > '.' 'add' '(' < exp > ')'
    | ('my' '.')? < ref > '.' 'remove' '(' < exp > ')'

< next > = < name > | 'kill' | 'nil'
```

Component prototype

```
< comp > =
   'component' < comp > '(' (< type > < name > (',' < type > < name > )*)? ')'
   '{'
      'store' '{'
          (('attrib'|'const')? (< type >)? < name > '=' < exp >;)*
      '}'
      'behaviour' '{'
            (< pdef >)*
      '}'
      'init' '{'
          < name > ('|' < pdef >)*
      '}'
   '}'
```

Collective definition

```
< coll > = 'collective' < name > < cblock >

< cblock > =
     'new' < name > '(' ()? ')' '@' < exp > '<' < exp > '>' ';'
     | 'for' < name > 'in' < exp > < cblock >
     | 'for' < name > 'from' < exp > ('by' < exp >)? 'to' < exp > < cblock >
     | 'if' '(' < exp > ')' < cblock > ('else' < cblock >)?
     | '{' ( < cblock > )* '}'
     | ('global' '.')? \tsymbol{ref} '=' < exp > ';'
```

Space Definition

```
< space > =
     'space' < name > '(' (< var > (',' < var > )*)? ')' '{'
         ('universe' '<' < type > < name > (',' < type > < name > )* '>')?
         'nodes' '{'
       (< nodes >)*
     '}'
     'connections' '{'
       (< edges >)*
     '}'
```

```
        ('areas' '{'
          < area >*
        '}')?
      '}'

    < nodes > =
          < node > ';'
      | 'for' < name > 'from' < exp > ('by' < exp >)? 'to' < exp > < nodes >
      | 'for' < name > 'in' < expr > < nodes >
      | '{' < nodes > '}'
      | 'if' '(' < exp > ')' < nodes > ('else' < nodes >)?

    < node > = ( < name > )? '[' < exp > (',' < exp >)* ']'

    < edges > =
          < node > ('->'|'<->') < node > ('{'
              < name > '=' < exp > (',' < name > '=' < exp >)*
              '}'
          )? ';'
      | 'for' < name > 'from' < exp > ('by' < exp >)? 'to' < exp > < edges >
      | 'for' < name > 'in' < expr > < edges >
      | '{' < edges > '}'
      | 'if' '(' < exp > ')' < edges > ('else' < edges >)?

    < area > = < name > '{' < nodes > '}'
```

System definition

```
    < sys > =
      'system' < name > '{'
        ('space' < name > '(' (< exp > ( ',' < exp > )*)? ')')?
        'collective' ( < name > | < cblock > )
            'environment' '{'
                < store >
                < prob >
                < weight >
                < rate >
                < update >
            '}'

      '}'

    < store > = 'store' '{'
          ( ('attrib'|'const')? (< type >)? < name > '=' < exp > ';')*
    '}'

    < prob > = 'prob' '{'
          < name > ('*')? '{'
              < stmt >*
          '}'
          ('default' '{'
              < stmt >*
          '}')?
    '}'

    < weight > = 'weight' '{'
          < name > ('*')? '{'
              < stmt >*
          '}'
          ('default' '{'
              < stmt >*
          '}')?
      '}'

    < rate > = 'rate' '{'
          < name > ('*')? '{'
              < stmt >*
          '}'
          ('default' '{'
              < stmt >*
          '}')?
    '}'

    < update > = 'update' '{'
          < name > ('*')? '{'
              < cblock >*
          '}'
    '}'
```

Measures

```
    < meas > = 'measure' < name > '(' (< type > < name > )?) = < exp > ';'
```