Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

STATISTICAL ANALYSIS OF CARMA MODELS: AN ADVANCED TUTORIAL

Vashti Galpin

LFCS, School of Informatics
University of Edinburgh
10 Crichton Streeet
Edinburgh, EH8 9AB, UNITED KINGDOM

Anastasis Georgoulas

Research IT Services
University College London
Gower Street
London, WC1E 6BT, UNITED KINGDOM

Michele Loreti

School of Science and Technology
University of Camerino
Via Madonna delle Carceri, 9
62032, Camerino, ITALY

Andrea Vandin

DTU Compute
Technical University of Denmark
Richard Petersens Plads, Building 324
DK-2800 Kgs. Lyngby, DENMARK

ABSTRACT

CARMA (Collective Adaptive Resource-sharing Markovian Agents) is a process-algebra-based quantitative language developed for the modeling of collective adaptive systems. A CARMA model consists of an environment in which a collective of components with attribute stores interact via unicast and broadcast communication, providing a rich modeling formalism. The semantics of a CARMA model are given by a continuous-time Markov chain which can be simulated using the CARMA Eclipse Plug-in. Furthermore, statistical model checking can be applied to the trajectories generated through simulation using the MultiVeStA tool. This advanced tutorial will introduce some of the theory behind CARMA and MultiVeStA as well as demonstrate its application to collective adaptive system modeling.

1 INTRODUCTION

CARMA, Collective Adaptive Resource-sharing Markovian Agents (Bortolussi et al. 2015; Loreti and Hillston 2016), is a modeling language, based on stochastic process-algebra, developed in the EU-funded QUANTICOL project to support the quantitative analysis of Collective Adaptive Systems (CAS).

CAS consist of massive numbers of components, featuring complex interactions among components and with humans and other systems. Each component in the system may exhibit autonomic behavior depending on its properties, objectives, and actions. Decision-making in such systems is complicated and interaction between their components may introduce new and sometimes unexpected behaviors. CAS also operate in open and non-deterministic environments. Components may enter or leave the collective at any time. Components can be highly heterogeneous (machines, humans, networks, etc.) each operating at different temporal and spatial scales, and having different (potentially conflicting) objectives.

CARMA combines the lessons which have been learned from the long tradition of stochastic process algebras – like PEPA (Hillston 1995), EMPA (Bernardo and Gorrieri 1998), Stochastic π-Calculus (Priami 1995), and others (Hermanns et al. 2002; Bortolussi and Policriti 2010) – with those more recently acquired from developing languages to model CAS, such as SCEL (De Nicola et al. 2014) and PALOMA (Feng and Hillston 2014), which feature attribute-based communication and explicit representation of locations.

Over the years, a number of modeling formalisms have been developed that consider space- or attribute-based communication including Cardelli and Gordon (2000), John et al. (2008), Parvu et al. (2015), Abd Alrahman et al. (2016), and Reinhardt and Uhrmacher (2017), but it is beyond the scope of this tutorial to offer a comparison.

A CARMA system consists of a *collective* operating in an *environment*. The collective is a multiset of components modeling the behavior of a system; it describes a group of interacting *agents*. The environment models all aspects that are intrinsic to the context where the agents are operating, and it 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 that 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 has an effect on *what* an agent can do.

To support the analysis of CAS with CARMA, a set of tools have been developed. The CARMA toolset consists of a rich Eclipse plug-in and of a *Command Line Inteface* (CLI) that, thanks to the integration with MultiVeStA (Sebastio and Vandin 2013), a statistical model checker (Agha and Palmskog 2018), supports statistical analysis and model checking of CAS.

CARMA has been used to model a number of different scenarios including taxi usage (Hillston and Loreti 2015), carpooling (Zoń et al. 2016), ambulance deployment (Galpin 2016), pedestrian mobility (Galpin et al. 2018) and network security (Chen et al. 2018). MultiVeStA has also been applied in different domains, including software product lines (ter Beek et al. 2015; ter Beek et al. 2018), crowd-steering (Pianini et al. 2014), public transportation systems (Gilmore et al. 2014; Ciancia et al. 2016), and swarm robotics (Belzner et al. 2014).

The CARMA Eclipse Plug-in provides a rich editor that can be used to develop models in CaSL, the CARMA Specification Language. 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 which 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.

The CLI tool is available as an executable and does not require the installation of Eclipse or of any additional library. It can, therefore, be used on any machine where Java 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.

In this advanced tutorial, we will show how CARMA and its toolset can be used to support the quantitative analysis of collective adaptive systems. A simple case study will be used. We will consider a robot search example where a swarm of robots move over city streets after a disaster. The robots enter the search area and spread out, searching. The first robot to find a casualty stops and communicates its location to other robots. When a base radio receives the location of a victim, it starts broadcasting a return signal, which is repeated by all robots outside the base, as they move home.

In Section 2, a brief description of CaSL is provided together with a description of the considered case study. In Section 3, the CLI and its integration with MultiVeStA is provided. In Section 4, we show the results of experiments with the robot search model, after which the conclusion is presented.

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*, which 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 modeling for users who are unfamiliar with process algebra and similar formal notations. CaSL incorporates all the features of CARMA. However, it also provides rich syntactic constructs that are inspired by *mainstream* programming languages and that simplify the work of system designers and modelers.

In the rest of this section we provide a gentle introduction to the main CaSL constructs that are needed to understand the considered case study. An interested reader can refer to the CaSL Web site (http://quanticol.github.io/CARMA/), where a detailed user manual is available. The model and scripts for its analysis are available at http://quanticol.github.io/CARMA/extra/wsc2018.html.

We now present the case study of this tutorial. We consider a robot search example where a swarm of robots move over a city landscape in a disaster recovery scenario. The robots enter the grid of streets from a base location and randomly move around the streets, searching for a casuality. When a robot finds a casualty, it stops and and broadcasts its location. Other robots rebroadcast the location while still traversing the streets. This is a flooding/gossiping protocol over a mobile ad-hoc network (Haas et al. 2002). Once the location of the casualty has been received by the base radio, it broadcasts a return signal. Each robot that receives this signal repeatedly rebroadcasts it, until that robot has returned to base.

In stochastic process algebras, data are typically abstracted away. The influence of data on behavior is captured only stochastically. When data are important to differentiate behaviors, they must be implicitly encoded in the state of processes. For CAS, where we want 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 kinds of *basic types*: booleans, integers, reals, and *spatial locations*, where the latter 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 consists of a sequence of *typed fields*. CaSL also supports *collections* as aggregations of homogeneous data, which can be either *sets* or *lists*. As usual, a *set* does not contain duplicated elements, while a *list* is a sequence of elements of the same type.

CaSL has a rich set of *expressions* that combine expressiveness and compact representation. Besides standard arithmetic and logical operators, CaSL expressions include operators for collections.

To model random behavior, CaSL expressions provide different mechanisms for sampling random values. When the expression `RND` is evaluated, the term is replaced with a value that is randomly selected from the interval $[0,1)$. To perform uniform selection of elements from a collection or sequence of values, the expression `U( e1 , ... , en )` can be used.

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

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

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

Constants can be used in our *robot scenario* to represent some parameters in the model like, for instance, the dimension of the city (`city_w` and `city_h`) or the number of robots (`num_robots`):

```
const city_w = 11;
const city_h = city_w;
const num_robots = 25;
```

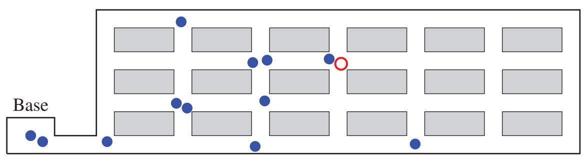
A *function declaration* has the following syntax:

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

where $\underline{<\text{name}>}$ is the function name, $\underline{<\text{arg}_i>}$ is the name of parameter i of type $\underline{<\text{type}_i>}$, while $\underline{<\text{type}>}$ is the return type. Finally, $\underline{<\text{body}>}$ contains the statements (in a high-level programming language) used to compute the returned value.

In our *scenario*, a function `OneBlock` selects (randomly) the next location of a robot:

```
fun location OneBlock (location current_loc){
    return U(current_loc.post); }
```



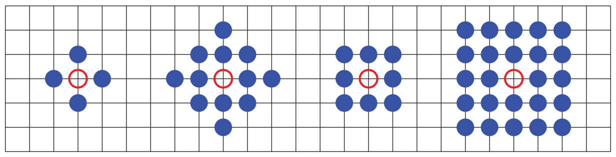


Figure 1: (Left) An example city grid where the red circle indicates the location of the casualty and the blue dots represent robots. (Right) Von Neumann neighborhood of size one, Von Neumann neighborhood of size two, Moore neighborhood of size one, Moore neighborhood of size two (from left to right). The blue dots define the grid points in the neighborhood, which includes the grid point of the red dot.

In the above, `current_loc` is a *location*, while `current_loc.post` indicates the collection of its neighbors out of which one is randomly selected.

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 behavior, 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 behaviors, as this leads to cumbersome models which can less readily be used in experiments on how components perform in different contexts. When system behavior 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. 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 for that. The initial focus has been on graphs as 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 with information about their location in space. This can be, for example, a grid 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.

In our scenario, we assume a city grid similar to the form given in Figure 1. In the model, the spatial structure we use is specified as `space GridPlusBase` where the appropriate number of nodes are generated, one for each intersection of the grid using an x- and y-coordinate. An additional node is defined for the base. The connections between the grid intersections that represent the streets are specified and the base is linked to the bottom left hand corner of the grid. The definition of `space GridPlusBase` in CaSL is as follows and defines a complete grid.

```
space GridPlusBase(int width, int height){
universe <int x, int y>
nodes { for x from 0 to width {
          for y from 0 to height { [x,y];} }
        [-1,0]; }
connections {
        for i from 0 to width {
          for j from 0 to height {
            if (i < width-1) {[i,j] <-> [i+1,j] { }; }
            if (j < height-1) {[i,j] <-> [i,j+1] { }; } } }
        [0,0] <-> [-1,0] { }; }
areas { base {[-1,0];} } }
```

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

```
component <name>( <type_1> <arg_1> ,..., <type_n> <arg_n> ) {
    store {
        attrib <type_1> <name_1> = <exp_1>;
        ...
        attrib <type_m> <name_m> = <exp_m>;
    }
    behaviour {
        <pdef_1>
        ...
        <pdef_p>
    }
    init { <pname_1>|...|<pname_k> }
}
```

Each component prototype has a (possibly empty) list of arguments. 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. 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 behavior of a component. It consists of a sequence of terms $\underline{<\mathsf{pname_i}>}$ referring to processes defined in the 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:

```
<pbody_1> + <pbody_2>
[<expr>]<pbody>
<act>.<proc>
```

Above, `<pbody_1> + <pbody_2>` indicates the *choice* between the behaviors `<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>` is a process that performs action `<act>` and then evolves to `<proc>`, representing the behavior activated after the action execution; this can be a process name `<pname>` or one of the process constants **nil** or **kill**. These represent the *inactive process* and the process that *destroys* the component. When **kill** is activated, the hosting component is removed from the system.

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

The syntax of *broadcast output* is the following:

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

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

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

where 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 with attribute `group` equal to **`my`**`.group` (the one evaluated locally). After the action is executed, attribute `counter` is incremented by 1. Note that this action is executed even if there are no components ready or able to execute a complementary action. It is required that `g` and all the sent values `ei` are *deterministic expressions* (i.e., without random values).

The syntax of *broadcast input* is similar:

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

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 the attribute `value`. When such a value is received, attribute `value` is updated.

The syntax of unicast output is the following:

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

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

A *unicast input* has the following shape:

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

In our scenario, we have three kinds of components, which are shown in Figure 2. The component `Casualty` is very simple; it is ready to communicate a request for help to anything or anyone in `AudioRange` and then has no further behavior. For this model, `AudioRange` requires co-location and hence the casualty can only be found by a robot at the same location.

The `Robot` component has five states, four of which are described by the comments above the state definitions `R1,...,R4` in Figure 2, and a final state **nil** in which no further behavior is possible. Initially, each robot leaves the base and moves randomly through the grid – this movement is generated by the prefix `move_one_block*[false]<>{`**`my.loc`**`:=OneBlock(`**`my.loc`**`);}` where broadcast with a `false` predicate is used to specify an internal action. Since broadcast is non-blocking and no other component can match `false`, only the sending component can take part in such an action.

The update calls the function `OneBlock`, which chooses randomly and uniformly from the post-set of the current location of the robot using the expression **`U`**`(current_loc.`**`post`**`)`. The post-set of a location is defined as all locations that are linked to it using `->` in the space description. The robot that finds the casualty remains at that location and starts broadcasting that location. Other robots that hear the location broadcast start broadcasting it as well as still moving randomly.

Component `Base` models a base station that listens for the location broadcast and then starts to broadcast a return-home signal once it knows the location. It counts in all but one robot.

Communication between the casualty and any robot, as well as counting-in communication between the base station and any robots, are unicast. All other communication is broadcast. CARMA allows the range of that communication for easily being expressed with a model with an explicit description of space. Three functions are defined to describe the range of talking or shouting (`AudioRange`), the range of robot radio broadcast (`RadioRange`), and the range of base station broadcast (`BaseRange`). These in turn depend on two functions `VonNeumannNgbrhd` and `MooreNgbrhd` (not shown), which are defined using `current_loc.`**`post`** where `current_loc` is the current location of the component.

```
component Robot (){
  store{ bool found = false;
         location cloc = none; }
  behaviour{
    // searching for casualty
    R1 = move_one_block*[false]<>{my.loc:=OneBlock(my.loc);}.R1 +
         help[AudioRange(my.loc,loc)](){my.found=true;}.R2 +
         relay_location*[RadioRange(my.loc,loc)](c){my.cloc:=c;my.found:=true;}.R3+
         return_signal*[RadioRange(my.loc,loc)]().R4;
    // found casualty, stop moving
    R2 = relay_location*[RadioRange(my.loc,loc)]<my.cloc>.R2;
    // received casualty location, continue moving while broadcasting location
    R3 = relay_location*[RadioRange(my.loc,loc)]<my.cloc>.R3 +
         move_one_block*[false]<>{my.loc:=OneBlock(my.loc);}.R3 +
         return_signal*[RadioRange(my.loc,loc)]().R4;
    // received return signal, continue moving while broadcasting return signal
    R4 = return_signal*[RadioRange(my.loc,loc)]<>.R4 +
         move_one_block*[false]<>{my.loc:=OneBlock(my.loc);}.R4 +
         [my.loc in base]at_base[true]<>.nil; }
  init{R1} }

component Base (){
  store{ bool report = false;
         bool returned = false;
         location cloc = none;
         int count = 0; }
  behaviour{
    B1 = relay_location*[BaseRange(my.loc,loc)](c){my.cloc:=c;my.report:=true;}.B2;
    B2 = return_signal*[BaseRange(my.loc,loc)]<>.B2 +
         at_base[my.loc==loc](){my.count:=my.count+1;}.B2 +
         [my.count==Num_robots-1]stop*[false]<>{my.returned:=true;}.nil; }
  init{B1} }

component Casualty (){
  store{}
  behaviour{ C1 = help[true]<>.nil; }
  init{C1} }
```

Figure 2: Robot, base, and casualty components.

These return `true` whenever the current location of a component and the location of the component with which it potentially could communicate, are in the Von Neumann neighborhood and Moore neighborhood of the specified size, respectively. Figure 1 illustrates the definition of these neighborhoods, for size 1 and size 2, and it can be seen that the Moore neighborhood is larger than the Von Neumann neighborhood.

Communication between two components can only take place if the predicates in both components are satisfied, i.e., if the locations of the two components taking part in the communication are in the specified neighborhood. In the case of the base communicating with a robot, both `RadioRange` and `BaseRange` must be satisified. In the example, `RadioRange` defines a smaller region than `BaseRange`, hence it is the truth value of `RadioRange` that determines whether communication is possible. Note that `AudioRange` is set to a neighborhood of size zero; this is equivalent to requiring that components be colocated.

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

```
system <name> {
  space <name>(<exp_1>, ..., <exp_n>)
```

```
    collective {
      < cblock >
    }
    environment { ···
    }
  }
```

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

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

```
new < name >( < exp1 >, . . . , < expn > )@< expl ><< exp >>
```

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

However, a large number of collectives can occur in a system. 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 an `environment` block is the following:

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

The `store` block 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.

As can be seen from the definition of the collective (contained in the definition of `system RS`) in Figure 3, one casualty component and one base component are instantiated as well as a number of robots. Both the base and the robots are initially located at the base location. The casualty is located within the city, and for our experiments we will assume a fixed location as specified in the collective definition. In the collective, space is instantiated using the description in Figure 3 and takes two parameters that describe the width and the height of the city grid.

We can observe that robots move at different speeds depending on their current task. In the case of searching, they typically move at a slower speed compared with when they are broadcasting the location or return-home signal. This is captured in the environment where the rate of `move_one_block*` is dependent on the value of `found` in the component (see Figure 3).

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

```
measure < name >( < type1 > < name1 >, ... ,< typen > < namen >) = < exp >;
```

Above, < 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 | < exp > }
```

```
system RS {
  space GridPlusBase (city_h,city_w)
  collective{
    new Casualty()@[city_h/2,city_h/2];
    new Base()@U(base);
    new Robot()@U(base)<num_robots>; }
  environment{
    rate{ move_one_block* {if (!sender.found) return move_search;
                                  else return move_nosearch;}
          default {return 100.0;} }
    store{} prob{} weight{} update{} }
```

Figure 3: System definition.

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

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

To compute statistics about attribute values of components operating in the system one can use the expressions: **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 the boolean expression $\underline{<\mathsf{exp}_g>}$, respectively.

In our scenario, we can use measures to count the number of robots in a given state or to test if the location of the casualty has been reported:

```
measure CountR1 = #{Robot[R1]|true};
measure CasualtyFound = #{Robot[R2]|true};
measure LocationReported = #{Base[*]|my.report==true};
```

3 CARMA TOOLS

The formal semantics of CARMA implicitly defines how to simulate models, and the implemented simulation algorithm makes use of this to help users understand a model's behavior. We now consider the software that supports the use of CARMA with a focus on the command line interface (CLI) and the integration of MultiVeStA into this interface.

3.1 Command Line Interface

Creating and editing models is made easier by the CARMA Eclipse Plug-in, which offers helpful features such as syntax highlighting and the ability to simulate models written in CaSL (Loreti and Hillston 2016). In addition to this, we have developed a CLI tool (http://quanticol.github.io/CARMA/cli.html). 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 an executable Java package and the tool (`CARMA-CL.jar`) can be downloaded at https://github.com/Quanticol/CARMA/releases. It does not require an installation of Eclipse or of any additional libraries. It can, therefore, be used on any machine where Java 1.8 is installed.

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 the final time of the simulation and the measures to be recorded. If desired, the user can override

```
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 4: Sample output of the CLI tool, as a human-readable description of a simulation experiment.

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 simulation. 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 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. An illustration of this process can be found at https://quanticol.github.io/CARMA/cli.html.

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 that can be controlled by the user. For each measure, the file contains the mean and standard deviation 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;
- two files containing the time taken for the whole experiment to run and the time taken up only by simulation, which can offer insight when comparing different versions of a model;
- 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 4).

Furthermore, a script file for the gnuplot 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 CLI, it can also be used to perform more elaborate statistical analysis through MultiVeStA (Section 3.2). In this case, the user must provide a file with the expressions to be evaluated. This gives access to the full expressive power of MultiQuaTEx, allowing for the formulation of complex queries. Additional parameters can be given to customise the default behavior of the algorithm, such as by specifying the desired confidence level of the result. At the end of the analysis, the results are stored in a text file, while a plot of them is displayed and also stored in an image file.

3.2 Statistical Analysis of CAS

While useful, simple simulation is by its nature limited in the information it can offer, and other approaches such as model checking can be used to complement its insights.

To provide further analysis options for CARMA models, we have developed an interface to the MultiVeStA platform, allowing CARMA users to access the model checking capabilities offered by this software. MultiVeStA, originally proposed by Sebastio and Vandin (2013), has been further developed within the QUANTICOL project (Gilmore et al. 2017). The tool belongs to the family of statistical model checkers of VeStA (Sen et al. 2005) and PVeStA (AlTurki and Meseguer 2011), and it can be used to compute 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, which generalises (the transient fragments of) other languages such as the logics PCTL (Hansson and Jonsson 1994), CSL (Baier et al. 2003) for Markov chains, and UTSL (Younes 2005) for general discrete-event systems. We refer to Agha et al. (2006) for examples of encodings in MultiQuaTEx of classic temporal operators like bounded until. MultiQuaTEx expressions are estimated by MultiVeStA according to a confidence interval (α, δ) provided by the user. Specifically, MultiVeStA estimates the expected value of a MultiQuaTEx query as the mean $\bar{x}$ of n samples (taken from n simulations), with n large enough (but minimal) to guarantee that the size of the $(1-\alpha)\cdot 100\%$ confidence interval is bounded by δ. In other words, with statistical confidence of $(1-\alpha)\cdot 100\%$, the actual expected value x belongs to the random interval $[\bar{x}-\frac{\delta}{2}, \bar{x}+\frac{\delta}{2}]$.

In the case of CARMA, quantities of interest that can be queried using MultiQuaTEx expressions can involve the current time in the simulation, the value of 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 insights than what could be gained by simulation alone.

4 ANALYSIS OF THE ROBOT SEARCH MODEL

This model provides a rich parameter space for exploration. We now illustrate results obtained from the model using CARMA simulation and MultiVeStA statistical model checking.

The first experiment, whose results are shown in the left and central graphs of Figure 5, demonstrates how the counts of robots in each state change over the time of the simulation. The number of robots in state `R1` starts to decrease once the casualty is found. Only one robot is ever in state `R2`, because the casualty component no longer tries to communicate if it has been found. Few robots go into state `R3` to communicate the location, as this is quickly communicated to the base, after which the number of robots in state `R4` increases and then decreases as they receive the return-home signal and then slowly (and randomly) find their way home. This suggests that there are opportunities for improving the model to ensure faster return and we discuss this further below. This experiment was conducted with the following parameters: 1.5 for the search move rate, 10 for the non-search move rate, 15×15 city grid, 25 robots, and the average values reported were obtained from 500 simulations.

The left and central graph in Figure 5 compare how this evolution of the system differs depending on whether `RadioRange` is defined as a Von Neumann neighborhood of size one or a Moore neighborhood of size one. As might be expected, the use of a Moore neighborhood (which is larger) results in slightly faster evolution. More noticeable is that the number of robots in state `R3` differs between the two scenarios. This state is the one in which the casualty location is broadcasted with the aim of communicating this information to the base. The larger neighborhood results in faster communication and, hence, there are fewer robots in this state, compared to the case of the smaller neighborhood, which reflects a slower diffusion of information.

The MultiVeStA experiments consider movement rates. In the first experiment, with results given on the right in Figure 5, the time taken to find the casualty is considered for different robot swarm sizes and different search movement rates. It can be seen that for larger swarms and high rates, the time to reach the

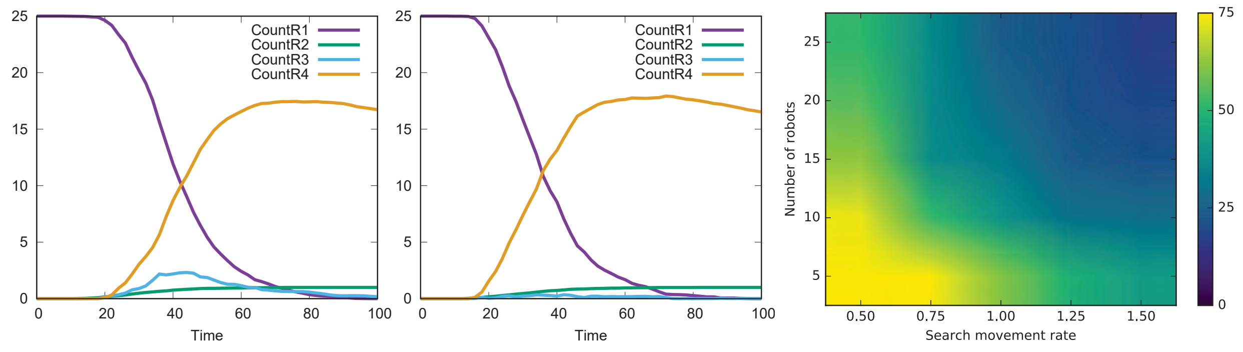


Figure 5: Count of robot states: Von Neumann `RadioRange` (left) and Moore `RadioRange` (centre). Time taken to find casualty (right).

casualty is under 20 time units, but then the time increases as the sizes and rates decrease. The data for this experiment were obtained with the MultiVeStA query

```
time_to_find() = if {s.rval("CasualtyFound")>0}
                     then {s.rval("time")} else #time_to_find() fi;
eval E[ time_to_find() ] ;
```

which considers if the casualty has been found. If it has been found, it returns the current simulated time (`s.rval()` queries observations on the current simulation state). If they haven't been found, then the query triggers the computation of a new simulation step (# is a one-step next operator), and then `time_to_find` will be recursively evaluated in the next simulation state. For this and the following experiment, an 11×11 city grid was used together with a Von Neumann radio range. In terms of MultiVeStA parameters, α was set at 0.1, δ was set at 10, meaning that the actual average value t for the number of time units to find the casualty is in the interval $[\bar{t}-5, \bar{t}+5]$ with statistical confidence of 90%, where $\bar{t}$ is the estimation of t computed by MultiVeStA. Hence, for each combination of robot number and movement rate, sufficient simulations were run to meet these constraints. The highest value for the number of simulations required was 900, with most of the remainder taking 200 simulations (the minimum permitted was 100).

It is also possible to consider the probability of events happening within a certain time. Figure 6 presents the results of the following query which is parameterised by different deadlines. Here, `20` and `60` are the first and last parameters, and `20` specifies the increment to be used to calculate the other parameters.

```
time_to_report(t) = if {s.rval("LocationReported")>0}
                         then {1}
                         else if {s.rval("time")>t}
                              then {0} else #time_to_report({t}) fi fi;
eval parametric( E[ time_to_report(x)],x,20,20,60) ;
```

The experiment shows the trade-off between different search and non-search movement rates. In the case of a 60-time-unit deadline, reporting the location to the base station is very likely to happen for any search rate of 1 or more. Even for the lower rates, there is still more than 50% chance of meeting the deadline. By contrast, for deadlines 20 and 40, a search rate of 0.50 cannot be paired with any non-search rate, if there is to be a reasonable chance of reporting the casualty's location before the deadline. In this experiment, the number of robots was fixed at 20, and the confidence interval was set at $\alpha = 0.1$ and $\delta = 0.05$. Hence, with statistical confidence of 90%, the actual probability of meeting the deadline p is in the interval $[\overline{p}-0.025, \overline{p}+0.025]$, with $\overline{p}$ the estimation of p computed by MultiVeStA.

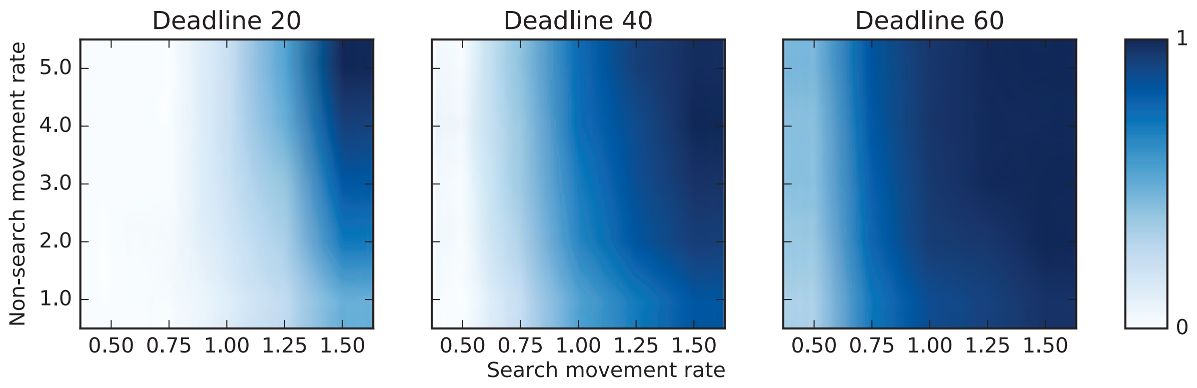


Figure 6: Probability of reporting location before 3 deadlines.

This model has been kept simple for the purpose of this tutorial; however, there are various modifications that could be made to improve performance and enhance realism, as suggested by the following partial list:

- A repulsion mechanism could be introduced to ensure that when the robots are searching, they move away from other robots.
- To ensure a faster return to base, an attraction mechanism could be added, but this could risk a clump of robots gathering in the furthest corner from the base.
- The robots could be specified to be smarter and have some ability to map as they move. This could then be used to move directly back to the base when the return-home signal is received, as well as communicate a direct route to the casualty.
- When the robot is searching, there could be a probability that is dependent on the speed of movement that affects whether the robot will hear the cry for help or not.
- Currently, the speed of the robot only depends on the task that the robot is undertaking. However, it is possible to have different speeds depending on the conditions of each street, with some streets blocked. This could be paired with mapping, and part of this task would be to report back the easiest route to the casualty.

5 CONCLUSIONS

In this advanced tutorial, we have shown how CARMA and its toolset can be used for the quantitative analysis of collective adaptive systems. CARMA, Collective Adaptive Resource-sharing Markovian Agents (Bortolussi et al. 2015; Loreti and Hillston 2016), is a modeling language based on stochastic process-algebra developed within the EU-funded QUANTICOL project.

We have used CARMA to model a scenario where robots move across city streets searching for casualties after a disaster. To illustrate the features of the CARMA toolset, statistical analysis of the provided model has been performed by using a Command Line Inteface and MultiVeStA (Sebastio and Vandin 2013).

ACKNOWLEDGEMENTS

This work is supported by the EU project QUANTICOL, 600708 and the EPSRC Platform Grant EP/N014758/1.

REFERENCES

Abd Alrahman, Y., R. De Nicola, and M. Loreti. 2016. "On the Power of Attribute-Based Communication". In *Proceedings of FORTE 2016*, edited by E. Albert and I. Lanese, LNCS 9688, 1–18. Cham: Springer.

Agha, G., and K. Palmskog. 2018. "A Survey of Statistical Model Checking". *ACM Transactions on Modeling and Computer Simulation* 28:6:1–6:39.

Agha, G. A., J. Meseguer, and K. Sen. 2006. "PMaude: Rewrite-based Specification Language for Probabilistic Object Systems". *Electronic Notes in Theoretical Computer Science* 153(2):213–239.

AlTurki, M., and J. Meseguer. 2011. "PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool". In *Proceedings of CALCO 2011*, edited by A. Corradini et al., LNCS 6859, 386–392. Cham: Springer.

Baier, C., B. Haverkort, H. Hermanns, and J.-P. Katoen. 2003. "Model-checking Algorithms for Continuous-time Markov Chains". *IEEE TSE* 29(6):524–541.

Belzner, L., R. De Nicola, A. Vandin, and M. Wirsing. 2014. "Reasoning (on) Service Component Ensembles in Rewriting Logic". In *Specification, Algebra, and Software – Essays Dedicated to Kokichi Futatsugi*, edited by S. Iida et al., LNCS 8373, 188–211. Cham: Springer.

Bernardo, M., and R. Gorrieri. 1998. "A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time". *Theoretical Computer Science* 202:1–54.

Bortolussi, L., R. De Nicola, V. Galpin, S. Gilmore, J. Hillston, D. Latella, M. Loreti, and M. Massink. 2015. "CARMA: Collective Adaptive Resource-sharing Markovian Agents". In *Proceedings of QAPL 2015*, edited by N. Bertrand and M. Tribastone, April 11th–12th, London, UK, 16–31.

Bortolussi, L., and A. Policriti. 2010. "Hybrid Dynamics of Stochastic Programs". *Theoretical Computer Science* 411(20):2052–2077.

Cardelli, L., and A. Gordon. 2000. "Mobile Ambients". *Theoretical Computer Science* 240(1):177–213.

Chen, W., Y. Lin, V. Galpin, V. Nigam, M. Lee, and D. Aspinall. 2018. "Formal Analysis of Sneak-Peek: A Data Centre Attack and its Mitigations". In *Proceedings of IFIP SEC 2018,* to appear.

Ciancia, V., D. Latella, M. Massink, R. Paskauskas, and A. Vandin. 2016. "A Tool-Chain for Statistical Spatio-Temporal Model Checking of Bike Sharing Systems". In *Proceedings of ISoLA 2016*, edited by T. Margaria and B. Steffen, LNCS 9952, 657–673. Cham: Springer.

De Nicola, R., M. Loreti, R. Pugliese, and F. Tiezzi. 2014. "A Formal Approach to Autonomic Systems Programming: The SCEL Language". *ACM TAAS* 9(2):7.

Feng, C., and J. Hillston. 2014. "PALOMA: A Process Algebra for Located Markovian Agents". In *Proceedings of QEST 2014*, edited by G. Norman and W. H. Sanders, LNCS 8657, 265–280. Cham: Springer.

Galpin, V. 2016. "Modelling Ambulance Deployment with CARMA". In *Proceedings of COORDINATION 2016*, edited by A. Lluch-Lafuente and J. Proença, LNCS 9686, 121–137. Cham: Springer.

Galpin, V., N. Zon, P. Wilsdorf, and S. Gilmore. 2018. "Mesoscopic Modelling of Pedestrian Movement Using CARMA and Its Tools". *ACM TOMACS* 28, article 11.

Gilmore, S., D. Reijsbergen, and A. Vandin. 2017. "Transient and Steady-State Statistical Analysis for Discrete Event Simulators". In *Proceedings of IFM 2017*, edited by N. Polikarpova and S. Schneider, LNCS 10510, 145–160. Cham: Springer.

Gilmore, S., M. Tribastone, and A. Vandin. 2014. "An Analysis Pathway for the Quantitative Evaluation of Public Transport Systems". In *Proceedings of IFM 2014*, edited by E. Albert and E. Sekerinski, LNCS 8739, 71–86. Cham: Springer.

Haas, Z., J. Halpern, and L. Li. 2002. "Gossip-based Ad Hoc Routing". In *Proceedings of IEEE INFOCOM 2002*, 1707–1716. Piscataway, New Jersey: IEEE.

Hansson, H., and B. Jonsson. 1994. "A Logic for Reasoning about Time and Reliability". *Formal Aspects of Computing* 6(5):512–535.

Hermanns, H., U. Herzog, and J. Katoen. 2002. "Process Algebra for Performance Evaluation". *Theoretical Computer Science* 274(1-2):43–87.

Hillston, J. 1995. *A Compositional Approach to Performance Modelling*. Cambridge: CUP.

Hillston, J., and M. Loreti. 2015. "Specification and Analysis of Open-Ended Systems with CARMA". In *Proceedings of E4MAS 2014*, edited by D. Weyns and F. Michel, LNCS 9068, 95–116. Cham: Springer.

John, M., C. Lhoussaine, J. Niehren, and A. Uhrmacher. 2008. "The Attributed Pi Calculus". In *Proceedings of CMSB 2008*, edited by M. Heiner and A. M. Uhrmacher, LNCS 5307, 83–102. Cham: Springer.

Loreti, M., and J. Hillston. 2016. "Modelling and Analysis of Collective Adaptive Systems with CARMA and its Tools". In *SFM 2016*, edited by M. Bernardo et al., LNCS 9700, 83–119. Cham: Springer.

Parvu, O., D. Gilbert, M. Heiner, F. Liu, N. Saunders, and S. Shaw. 2015. "Spatial-Temporal Modelling and Analysis of Bacterial Colonies with Phase Variable Genes". *ACM Transaction on Modeling and Computer Simulation* 25, article 13.

Pianini, D., S. Sebastio, and A. Vandin. 2014. "Distributed Statistical Analysis of Complex Systems Modeled Through a Chemical Metaphor". In *Proceedings of HPCS 2014*, 416–423. Piscataway, New Jersey: IEEE.

Priami, C. 1995. "Stochastic π-calculus". *The Computer Journal* 38(7):578–589.

Reinhardt, O., and A. Uhrmacher. 2017. "An Efficient Simulation Algorithm for Continuous-time Agent-based Linked Lives Models". In *Proceedings of the 50th Annual Simulation Symposium*, edited by S. Jafer et al., article 9. New York: Society for Computer Simulation International/ACM.

Sebastio, S., and A. Vandin. 2013. "MultiVeStA: Statistical Model Checking for Discrete Event Simulators". In *Proceedings of ValueTools 2013*, edited by A. Horvath et al., 310–315. New York: ICST/ACM.

Sen, K., M. Viswanathan, and G. Agha. 2005, Sept. "VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems". In *Proceedings of QEST'05*, 251–252. Piscataway, New Jersey: IEEE.

ter Beek, M. H., A. Legay, A. Lluch-Lafuente, and A. Vandin. 2015. "Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints". In *Proceeding of SPLC 2015*, edited by D. C. Schmidt, 11–15. New York: ACM.

ter Beek, M. H., A. Legay, A. Lluch Lafuente, and A. Vandin. 2018. "A Framework for Quantitative Modeling and Analysis of Highly (Re)configurable Systems". *IEEE Transactions in Software Engineering*. to appear.

Younes, H. L. 2005. "Probabilistic Verification for Black-box Systems". In *Proceedings of CAV 2005*, edited by K. Etessami and S. K. Rajamani, LNCS 3576, 253–265. Cham: Springer.

Zoń, N., S. Gilmore, and J. Hillston. 2016. "Rigorous Graphical Modelling of Movement in Collective Adaptive Systems". In *Proceedings of ISoLA 2016*, edited by T. Margaria and B. Steffen, LNCS 9952. Cham: Springer.

AUTHOR BIOGRAPHIES

VASHTI GALPIN is a researcher in the School of Informatics at the University of Edinburgh, where she also obtained her PhD. Her research interests include formal quantitative modeling of systems and process algebra. Her email address is Vashti.Galpin@ed.ac.uk.

ANASTASIS GEORGOULAS is a Research Software Engineer at University College London. His research interests focus on modeling of stochastic systems, and the interplay between formal methods and machine learning. His email address is a.georgoulas@ucl.ac.uk.

MICHELE LORETI is Associate Professor at the University of Camerino and was before (until 2017) Researcher Associate at the University of Firenze. His main research interests are formal methods for supporting design, deployment, and runtime monitoring of large-scaled adaptive systems. His email address is michele.loreti@unicam.it.

ANDREA VANDIN is Assistant Professor at DTU, the Technical University of Denmark. Before that he was an Assistant Professor at the IMT School for Advanced Studies Lucca, Italy, until 2017, and Senior Research Assistant at the University of Southampton, UK, until 2015. His main research interests are in the development of scalable techniques for formal quantitative system analysis. His email address is anvan@dtu.dk.