Modelling movement for collective adaptive systems with CARMA

Natalia Zoń Vashti Galpin Stephen Gilmore

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh

s1478292@sms.ed.ac.uk Vashti.Galpin@ed.ac.uk Stephen.Gilmore@ed.ac.uk

Space and movement through space play an important role in many collective adaptive systems (CAS). CAS consist of multiple components interacting to achieve some goal in a system or environment that can change over time. When these components operate in space, then their behaviour can be affected by where they are located in that space. Examples include the possibility of communication between two components located at different points, and rates of movement of a component that may be affected by location. The CARMA language and its associated software tools can be used to model such systems. In particular, a graphical editor for CARMA allows for the specification of spatial structure and generation of templates that can be used in a CARMA model with space. We demonstrate the use of this tool to experiment with a model of pedestrian movement over a network of paths.

1 Introduction

Collective adaptive systems consist of multiple components or agents that interact collaboratively on common goals, and compete to achieve individual goals. They 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. These systems are called *collective* because of the interaction of many components, and *adaptive* because they respond to changes in the environment in which they operate. They are often characterised as having emergent behaviour, that is behaviour which cannot be predicted in advance by considering the individual components in isolation from each other. Modelling of CAS is crucial because it is difficult to understand the behaviour of the overall system just by inspecting the behaviour of the components. Modelling allows us to experiment with the system before implementation and deployment. This paper considers the CARMA language which has been developed specifically for the modelling of CAS [12] with a particular focus on smart city concerns such as smart transport and smart energy grids.

The use of *local* versus *global* above suggests that space may play an important role in CAS. While this is not necessarily true of all CAS since the distinction between local and global may be logical or virtual rather than physical, it is true for many CAS, and hence this is an important part of understanding their behaviour. This paper focusses on a spatial example and illustrates how CARMA can be used to model this example.

We consider the example of 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 essentially two groups of pedestrians 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. During rush hour in the morning and afternoon, people want to traverse the park or lanes as fast as possible so that they are not late, and we wish to investigate what features enable these pedestrians to pass through the network efficiently. If there are multiple paths, it would seem in advance that it makes sense to use some paths for one direction and other paths for the other direction. This raises the question of what routing or information such as signs is sufficient for the two groups of pedestrians to separate out onto different paths. This paper presents an initial investigation into the modelling of this scenario, and we demonstrate how this can be achieved using CARMA, its Eclipse Plug-In and its Graphical Eclipse Plug-in, considering different possibilities for the network.

The paper starts with a brief discussion of the CARMA language before describing the modelling of the scenario in more detail and presentation of our experiments and results from various networks, followed by conclusions and discussion of future work.

2 CARMA

The CARMA process calculus has been developed specifically for the modelling of collective adaptive systems and a full description of the language can be found in [12]. Here, we give a brief outline. A CARMA model consists of a collective N and the environment $\mathscr{E}$ in which it operates, using the syntax N **in** $\mathscr{E}$. A collective is either a component C or collectives in parallel $N \parallel N$. Each component is either null, $\mathbf{0}$, or a combination of behaviour described by a process P and a store of attributes γ, denoted by (P, γ). We use function notation to denote store access, thus if $\gamma = \{x \mapsto v\}$ then $\gamma(x) = v$.

Prefix, constants, choice and parallel composition can be expressed in the standard manner by defining P appropriately. Additionally, there is the **nil** process which does nothing, the **kill** process which results in the component being removed from the collective, and the option of prefixing a process with a predicate $[\pi]P$, in which case the process P can only proceed if the predicate π evaluates to *true* using the values of the attributes in the component's store γ. To improve readability we sometimes parenthesise the process expression P, writing this term as $[\pi](P)$. The meaning is unchanged.

Process prefixes are rich and permit actions that provide value-passing unicast and broadcast communication using predicates on the attributes in the store of the sending and receiving component. Communication between components will only take place if the predicates evaluate to *true*. The value *false* indicates that no communication partner is needed. Furthermore, attribute values can be updated (probabilistically) on completion of an action.

Unicast communication is blocking; the sender cannot output values unless there is a matching input action which can be performed by another component. In contrast, broadcast is not blocking, and we can use a specific form with a constant *false* predicate (written as $\perp$ here) to allow components to act without interaction with other components, as seen in the example to follow.

The syntax of a non-blocking broadcast on name α is $\alpha^\star[\pi]\langle\vec{v}\rangle\sigma$ where π is a predicate which must be satisfied by all processes wishing to receive this broadcast. The vector $\vec{v}$ is a vector of values to be communicated; this vector may be empty. The suffix σ is an *update* of variables in the local store of a component. A component refers to its local store with the prefix my (similar to this in Java) so an update to store the value of x as the new value of $\mathsf{my}.x$ is written as $\{\mathsf{my}.x \leftarrow x\}$. As an example, the prefix process term $\mathsf{move}^\star_{ij}[\perp]\langle\rangle\{\mathsf{my}.x \leftarrow i, \mathsf{my}.y \leftarrow j\}.\mathit{Ped}$ broadcasts that it is performing a move_{ij} activity, updates its local x and y values, and continues as the process *Ped*.

The environment contains both the global store and an evolution rule which returns a tuple of four functions $(\mu_p, \mu_w, \mu_r, \mu_u)$ known as the *evaluation context*. Communication between sender s and receiver r on activity α has both an associated *probability* (determined by μ_p) and a *weight* (determined by μ_w). These functions depend on activity α and both the attribute values of the sender (in the store γ_s)

and the attribute values of the receiver (in the store γ_r). The activity *rate* however depends on only the attribute values in the store of the sender (γ_s); the attribute values of the receiver do not affect the rate at which a communication activity is performed. Thus the first three functions in the evaluation context determine probabilities, weights and rates that supply quantitative information about the behaviour of actions. The fourth function μ_u performs global updates, either of the attributes in the global store or of the collective by adding new components. These updates include the usual initialisation of variables, incrementing counters, or accumulating totals.

The operational semantics of CARMA are defined in FuTS style [3] and define for each model a time-inhomogeneous continuous-time Markov chain (ICTMC). The behaviour of these ICTMCs can be simulated and the CARMA Eclipse Plug-in provides this functionality. The software tools for processing CARMA models are available from `http://quanticol.sourceforge.net`. More information about the QUANTICOL project which created these tools is available from `http://www.quanticol.eu`.

3 Automatic code generation

The CARMA Graphical Editor allows the user to specify the structure of movement in a CAS model by laying out graphical symbols on a plane [15]. The editor generates CARMA code from the graph which the user has defined. In addition to normal attributes, CARMA components which are defined in this way have a set of distinguished attributes to specify their current location in space.

Each CARMA component in the model may further have its mobility restricted to a given set of paths through the graph defined by the user. Paths in this context are subgraphs of the user-defined graph consisting of a set of uniform vertices connected by directed, coloured edges. At any given time in the system's evolution, the location attributes of a component instance must be equal to the location of one of the nodes belonging to the subgraph where that component is restricted. A component can change its location attributes only if there exists a path from its current node to the new node, and if this path belongs to the subgraph where the mobility of this component is restricted.

In CARMA, functions are used for storing the information about each subgraph's topology. Component actions query these functions during the execution of their predicates, and can modify the component's location attributes accordingly, in the update block. For each node which can be accessed by a particular component type, a movement action must be included in the component's behaviour. If the node can be accessed by a component in more than one state, the action must be specified separately for each state. In systems with complex mobility restriction graphs, topology-defining functions, as well as component behaviour blocks may require a large number of lines of code. This code is automatically generated by the CARMA Graphical Editor, freeing the modeller from the task of manually producing this CARMA model code.

4 Pedestrian model

The CARMA model is illustrated in Figures 1 and 2. It assumes that there are two types of pedestrians, A and B, and that P and Q are variables of type pedestrian. The two *Arrival* components generate pedestrians at two different locations (on opposite sides of the graph), and the pedestrians move from their origin side to the opposite side. Once a pedestrian has reached its goal, the count for that type of pedestrian is incremented and the time taken for traversal is added to the total time so that the average traversal time can be calculated for each pedestrian type.

Store of *Pedestrian* component:

P	pedestrian type — an enumeration with values A and B
x	current x coordinate
y	current y coordinate
stime	time of arrival

Behaviour of *Pedestrian* component:

$$\begin{aligned} Ped \overset{\text{def}}{=} & \textstyle\sum_{(i,j)\in V} \big[\mathbf{ExistsPath}(P,x,y,i,j)\big] \big(\mathsf{move}^{\star}_{ij}[\bot]\langle\rangle\{\mathsf{my}.x \leftarrow i, \mathsf{my}.y \leftarrow j\}.Ped\big) \\ & + \big[\mathbf{AtGoal}(P,x,y)\big] \big(\mathsf{fin}^{\star}[\bot]\langle\rangle.\mathbf{nil}\big) \end{aligned}$$

Initial state of *Pedestrian* component: *Ped*

Store of *Arrival* component:

P	pedestrian type

Behaviour of *Arrival* component:

$$Arr \overset{\text{def}}{=} \mathsf{arrive}^{\star}[\bot]\langle\rangle.Arr$$

Initial state of *Arrival* component: *Arr*

Figure 1: The *Pedestrian* and *Arrival* components

The model is parameterised by a number of functions that capture the graph information and are generated automatically as described above.

- $\mathbf{ExistsPath}(P,x,y,i,j)$ is a Boolean function that determines if an edge exists between a pedestrian's current position and another node in the graph, hence a $\mathsf{move}^{\star}_{ij}$ action can only occur when such an edge exists.
- $\mathbf{AtGoal}(P,x,y)$ is a Boolean function that checks if the pedestrian has reached its goal, hence $\mathsf{fin}^{\star}$ can only occur once the destination has been reached. After this the pedestrian does not move any more.
- $\mathbf{ArrivalRate}(P)$ determines the arrival rate for each type of pedestrian.
- $\mathbf{Start}_x(P)$ and $\mathbf{Start}_y(P)$ define the initial location of a new pedestrian depending on its type.

A function that is not directly related to the graph structure is $\mathbf{MoveRate}(P,x,y,i,j,\ldots)$ which determines the rate of movement along a particular edge, and can take additional parameters that can affect this rate such as the current count of other pedestrians of the same or different type. We use the following definition that uses the numbers of pedestrians of the other type at the target node to reduce the movement rate.

$$\mathbf{MoveRate}(P,x,y,i,j,A_{ij},B_{ij}) = \begin{cases} move_A/(B_{ij}+1) & \text{if } P = A \\ move_B/(A_{ij}+1) & \text{if } P = B \end{cases}$$

where A_{ij} are the number of A pedestrians at the target node and B_{ij} are the number of B pedestrians at the target node, and $move_Q$ is a basic movement rate for each pedestrian type.

Constants:

V	set of coordinate pairs representing nodes in the graph

Measures:

$average_P$	average time for traversal by pedestrians of type P

Global store:

$count_P$	number of P pedestrians to complete traversal
$total_P$	total time for all completed P pedestrian traversals

Evolution rule functions:

$$\mu_p(\gamma_s, \gamma_r, \alpha) = 1$$
$$\mu_w(\gamma_s, \gamma_r, \alpha) = 1$$

$$\mu_r(\gamma_s, \alpha) = \begin{cases} \textbf{ArrivalRate}\big(\gamma_s(P)\big) & \text{if } \alpha = \mathsf{arrive}^\star \\ \textbf{MoveRate}\big(\gamma_s(P), \gamma_s(x), \gamma_s(y), i, j, \ldots\big) & \text{if } \alpha = \mathsf{move}^\star_{ij} \\ \lambda_{fast} & \text{otherwise} \end{cases}$$

$$\mu_u(\gamma_s, \alpha) = \begin{cases} \{\}, \big(Pedestrian, \{P \leftarrow \gamma_s(P), x \leftarrow \textbf{Start}_x(\gamma_s(P)), y \leftarrow \textbf{Start}_y(\gamma_s(P)), stime \leftarrow \mathsf{now}\}\big) \\ \qquad \text{if } \alpha = \mathsf{arrive}^\star \\ \{count_{\gamma_s(P)} \leftarrow count_{\gamma_s(P)} + 1, total_{\gamma_s(P)} \leftarrow total_{\gamma_s(P)} + (\mathsf{now} - \gamma_s(stime))\}, 0 \\ \qquad \text{if } \alpha = \mathsf{fin}^\star \\ \{\}, 0 \qquad \text{otherwise} \end{cases}$$

Collective:

$$PedAB \overset{\text{def}}{=} \big(Arrival, \{P \mapsto A\}\big) \parallel \big(Arrival, \{P \mapsto B\}\big)$$

Figure 2: Environment and collective

Figure 2 specifies the four functions $(\mu_p, \mu_w, \mu_r, \mu_u)$ known as the evaluation context. Probabilities and weights on activities are not used in this model so the μ_p and μ_w functions are trivially constant functions.

5 Model instances

Four instances of this CARMA model are shown in Figure 3. These show instantiations of the general CARMA model from the previous section with increasing size and shape complexity. The central repeating features of the path network are the cross-bars in the centre of the network. In the simplest instance we have only one cross-bar and we describe this instance as having height 1 and width 1, representing it as instance 1×1. As the cross-bar structure is repeated we have instances 1×2, 2×1, and 2×2, depending where the additional structure is added into the network.

An increase in the width of the network has the obvious consequence that journeys across the network take longer. An increase in the height of the network has the consequence that pedestrians are offered an increased choice of routes, with the implicit consequence that individual paths are less congested (because there are more of them on offer). The edges are equally long and thus the time to traverse them is the same under comparable conditions.

In each instance of the network of paths there are two sub-networks which restrain the movement of

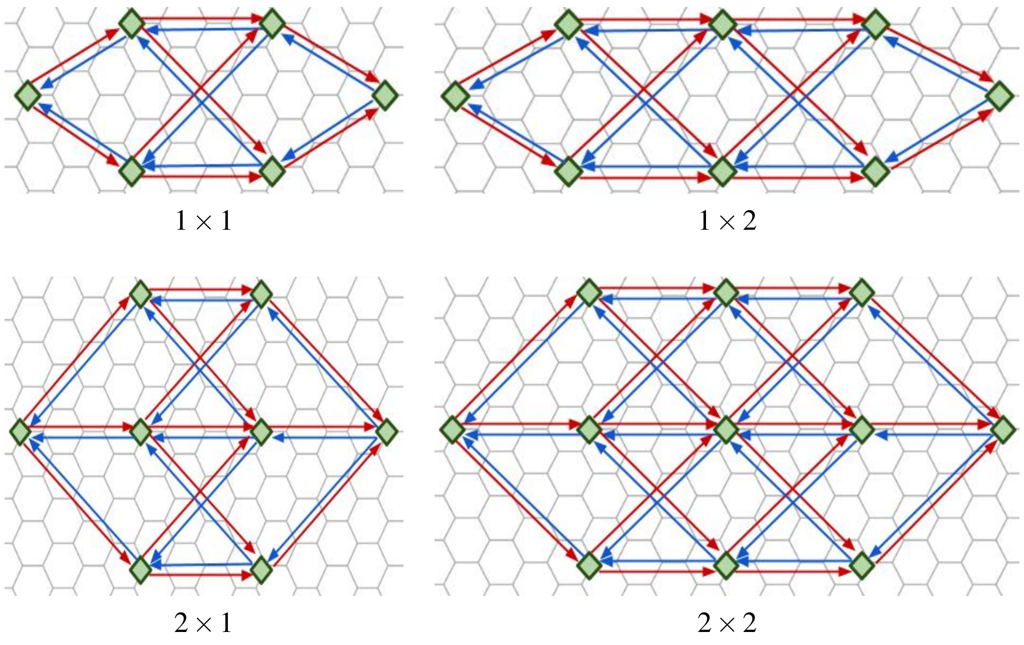

Figure 3: Four model instances of increasing size and complexity.

the pedestrians of type A and type B. Pedestrians of type A are restricted to the red sub-network and must cross the network from left to right. Pedestrians of type B are restricted to the blue sub-network and must cross the network from right to left. The networks illustrated in Figure 3 are symmetric but this is of no particular significance and it would pose no difficulty to work with networks which were not symmetric.

These graphs were drawn in the CARMA Graphical Editor and CARMA code was generated from it, including all necessary instances of the **ExistsPath**, **AtGoal**, and **ArrivalRate** functions and applications of these in predicate guards on processes.

6 Analysis and results

We analysed our CARMA model using the CARMA Eclipse Plugin [12]. The CARMA Eclipse Plugin provides a helpful syntax-aware editor for the CARMA *Specification Language*, implemented in the XText editor framework. The CARMA Specification Language provides a wrapper around the CARMA process calculus adding non-essential (but useful) features such as data types and data structures, functions, and the ability to specify real-valued *measures* of interest over the model. In some modelling languages measures of interest or Markov reward structures are defined externally to the model but in CARMA and languages such as CASPA [10], PRISM [11] and ProPPA [6], the specification of measures of interest and reward structures is included in the modelling language itself.

Given a CARMA specification, the CARMA Eclipse Plug-in compiles the model into a set of Java classes which are linked with the CARMA simulator classes to provide a custom simulator for this model. The compiled Java code is executed to compute the measures of interest from an ensemble of simulation runs. The CARMA simulator uses a *kinetic Monte-Carlo* algorithm to select the next simulation event to fire and draws from the appropriate weighted random number distribution to determine the duration of the event. The simulation state is updated as specified by the event which was fired and the simulation proceeds forward until a pre-specified simulation stop time is reached.

The measure functions defined by the modeller are passed into the simulation environment and provide a view onto the raw simulation results at intervals which are specified by the modeller. The Apache Commons Math Library is used within the Plug-in to perform statistical analysis of the data. The *Simulation Laboratory View* provided by the CARMA Eclipse Plug-in acts as an electronic laboratory notebook, recording details of the simulation studies which have been performed.

The CARMA Eclipse Plugin and the CARMA Graphical Editor are available from the SourceForge website at `http://quanticol.sourceforge.net`. After installation they can be kept up-to-date using the standard mechanism in Eclipse to check for updates.

6.1 Design of experiments

We designed a suite of experiments to explore the behaviour of the model. To provide a baseline for average travel time we investigated the travel time in the presence of only one type of pedestrian (thereby giving a model which has no congestion). Thereafter we investigated the models with congestion in the presence or absence of pedestrian routing. When routing is present, only one starting route has a non-zero rate, and the non-zero rate is assigned in order to direct pedestrians away from each other.

We used the CARMA Graphical Editor to automatically create CARMA code models. Fig. 4 shows how the number of lines of CARMA code grows with model structure complexity.

Model	Nodes	Connections	LoC
1x1	6	8	208
1x2	8	12	248
1x3	10	16	288
2x1	8	13	258
2x2	11	20	328
2x3	14	27	398
3x1	10	18	308
3x2	14	28	408
3x3	18	38	508

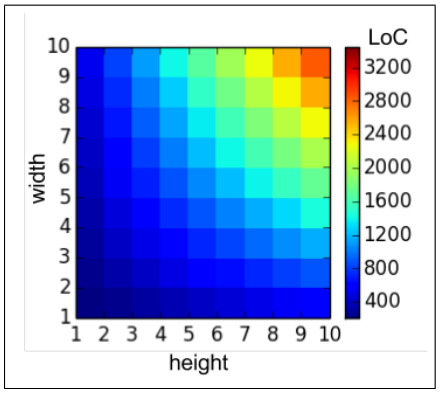

Figure 4: The number of lines of CARMA code per model structure. Left, for small values of width and height. Right, for larger values of width and height.

6.2 Analysis

The results from our experiments are presented in Figure 5. We have three results (no congestion, routing, and no routing) for each of the four model instances considered (1×1, 1×2, 2×1, and 2×2). 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 narrow networks where congestion in experienced most (i.e. in the 1×1 structure and the 1×2 structure).

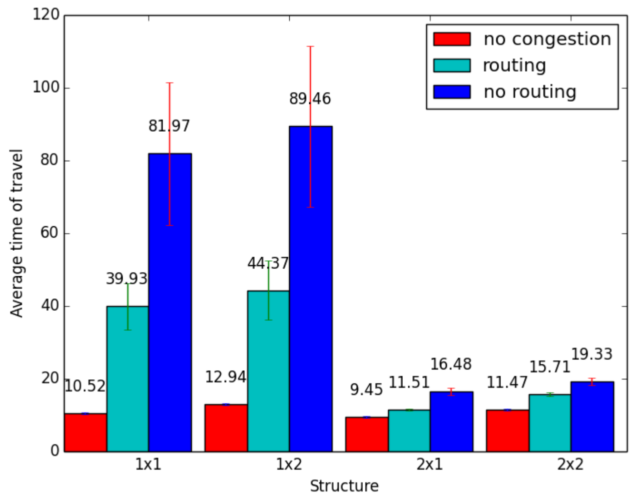

Figure 5: Average travel time results from the experiments on structure and network usage.

7 Related work

The CARMA language provides high-level language constructs for describing communicating processes. The language has a stochastic semantics expressed in terms of continuous-time Markov chains. The language contains some features which are familiar from languages such as Bio-PEPA [2], PRISM [11] and the Attributed π-calculus [8]. In this section we compare CARMA to these established modelling languages and highlight differences in approach between them.

Each of the languages considered here has the potential to be used to model stochastic systems with mobile populations of individuals but language design decisions, the choice of language features, and underlying analysis mechanisms can make one of the languages better-suited for a particular modelling problem than the others. As examples of modelled systems, Bio-PEPA has been used to model scenarios where safe movement of people is an important factor in systems including *emergency egress* [13] and *crowd formation and movement* [1]. PRISM has been used to model *dynamic power management controllers* [14] and *human-in-the-loop UAV mission planning* [4]. The Attributed π-calculus has been used to model *spatial movement in phototaxis* [8], and *cooperative protein binding in gene regulation* [9]. CARMA has been used to model a number of spatial CAS including *carpooling* [15], *taxi movement* [7] and *ambulance deployment* [5].

Inter-process communication in CARMA is *attribute-based*; communication partners are determined dynamically as the model evolves through state-to-state transitions. Communication in the Attributed π-calculus is similarly dynamic. In contrast, the communication partners of Bio-PEPA and PRISM components are determined statically, and do not change as state-to-state transitions occur. Additionally, CARMA and the Attributed π-calculus support value-passing communication whereas the Bio-PEPA and PRISM languages do not.

The primary analysis method for CARMA models is simulation. This is also the case for Bio-PEPA and the Attributed π-calculus whereas analysis of PRISM models is typically through probabilistic model-checking.

The CARMA language and the PRISM language are explicitly-typed. Types such as boolean, integer and real are ascribed to variables in the language by the modeller, or inferred by the language type-checker. In contrast, types in Bio-PEPA and the Attributed π calculus are implicit. Explicitly-typed languages can make the modeller's intentions more obvious, when, for example, expecting to receive an initial integer value instead of a real value.

CARMA provides guarded process definitions (used in a similar way to the guarded commands found in the PRISM language; the Attributed π-calculus does not support these directly; Bio-PEPA has no boolean expressions at all). Guarded process definitions allow declarative descriptions of the relationships between locations in a network and we have used this description mechanism comprehensively here.

In common with PRISM, CARMA provides strong support for encapsulation, with variable declarations being local to an enclosing structure (in PRISM this is a module, whereas in CARMA it is a component). A structuring mechanism such as this is not found in Bio-PEPA or the Attributed π-calculus where declarations of rate functions, channel names, process definitions or species definitions have global scope.

Differently from the other languages considered here, CARMA treats location and space as an aspect of a model which can be described separately from the detailed model dynamics. Through the provision of a graphical editor for CARMA, space, location, and connectivity can be treated separately from logic, communication, and synchronisation. This separation of concerns may make it easier to maintain a model of a system when the spatial structure of the system changes.

8 Conclusions and future work

We have demonstrated a simple model of pedestrian movement over a number of different graphs, to illustrate the modelling of spatial aspects of CAS. The CARMA Graphical Editor allowed us to automatically generate the CARMA code for different networks which simplified the task, and allowed our pedestrian components to be generic in nature. Our initial experiments have considered situations with and without congestion as well as with and without explicit routing of pedestrians as they enter the network.

There are many directions for future work. For example, another group of pedestrians could also be introduced, namely, tourists, and the focus would be on efficient traversal of the network during afternoon rush hours when commuters want to get home quickly and tourists wish to sightsee, and hence move slowly. We are also interested in identifying when the model shows emergent behaviour, in the sense that different groups of pedestrian use different paths through the network in response to environmental cues such as information about congestion or routing suggestions (rather than explicit routing).

Acknowledgements: This work is supported by the EU project QUANTICOL, 600708. We thank the anonymous reviewers for many helpful comments which encouraged us to improve the paper.

References

[1] L. Bortolussi, D. Latella & M. Massink (2013): *Stochastic Process Algebra and Stability Analysis of Collective Systems*. In R. De Nicola & C. Julien, editors: *Coordination Models and Languages, 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, *Lecture Notes in Computer Science* 7890, Springer, pp. 1–15, doi:10.1007/978-3-642-38493-6_1.

[2] F. Ciocchetta & J. Hillston (2009): *Bio-PEPA: A framework for the modelling and analysis of biological systems*. Theor. Comput. Sci. 410(33-34), pp. 3065–3084, doi:10.1016/j.tcs.2009.02.037.

[3] R. De Nicola, D. Latella, M. Loreti & M. Massink (2013): *A uniform definition of stochastic process calculi*. ACM Computing Surveys 46, p. 5, doi:10.1145/2522968.2522973.

[4] L. Feng, C. Wiltsche, L. Humphrey & U. Topcu (2015): *Controller Synthesis for Autonomous Systems Interacting with Human Operators*. In: Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems, ICCPS '15, ACM, New York, NY, USA, pp. 70–79, doi:10.1145/2735960.2735973.

[5] V. Galpin (2016): *Modelling Ambulance Deployment with CARMA*. In A. Lluch Lafuente & 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, Springer, pp. 121–137, doi:10.1007/978-3-319-39519-7_8.

[6] A. Georgoulas, J. Hillston, D. Milios & G. Sanguinetti (2014): *Probabilistic Programming Process Algebra*. In G. Norman & William Sanders, editors: Quantitative Evaluation of Systems: 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings, Springer, pp. 249–264, doi:10.1007/978-3-319-10696-0_21.

[7] J. Hillston & M. Loreti (2015): *Specification and Analysis of Open-Ended Systems with* CARMA. In: Fourth International Workshop on Agent Environments for Multi-Agent Systems, Revised Selected and Invited Papers (E4MAS 2014), LNCS 9068, Springer, pp. 95–116, doi:10.1007/978-3-319-23850-0_7.

[8] M. John, C. Lhoussaine, J. Niehren & A.M. Uhrmacher (2008): *The Attributed Pi Calculus*. In M. Heiner & A.M. Uhrmacher, editors: Computational Methods in Systems Biology: 6th International Conference CMSB 2008, Rostock, Germany, October 12-15, 2008. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 83–102, doi:10.1007/978-3-540-88562-7_10.

[9] M. John, C. Lhoussaine, J. Niehren & A.M. Uhrmacher (2010): *The Attributed Pi-calculus with Priorities*. In C. Priami, R. Breitling, D. Gilbert, M. Heiner & A.M. Uhrmacher, editors: Transactions on Computational Systems Biology XII, Springer-Verlag, Berlin, Heidelberg, pp. 13–76, doi:10.1007/978-3-642-11712-1_2.

[10] M. Kuntz, M. Siegle & E. Werner (2004): *Symbolic Performance and Dependability Evaluation with the Tool CASPA*. In M. Núñez, Z. Maamar, F.L. Pelayo, K. Pousttchi & F. Rubio, editors: Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops The FormEMC, EPEW, ITM, Toledo, Spain, October 1-2, 2004, LNCS 3236, Springer, pp. 293–307, doi:10.1007/978-3-540-30233-9_22.

[11] M. Kwiatkowska, G. Norman & D. Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In G. Gopalakrishnan & S. Qadeer, editors: Proc. 23rd International Conference on Computer Aided Verification (CAV'11), LNCS 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[12] M. Loreti & J. Hillston (2016): *Modelling and Analysis of Collective Adaptive Systems with CARMA and its Tools*. In M. Bernardo, R. De Nicola & 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, Springer International Publishing, Cham, pp. 83–119, doi:10.1007/978-3-319-34096-8_4.

[13] M. Massink, D. Latella, A. Bracciali, M. D. Harrison & J. Hillston (2012): *Scalable context-dependent analysis of emergency egress models*. Formal Asp. Comput. 24(2), pp. 267–302, doi:10.1007/s00165-011-0188-1.

[14] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla & R. Gupta (2002): *Formal Analysis and Validation of Continuous Time Markov Chain Based System Level Power Management Strategies*. In W. Rosenstiel, editor: Proc. 7th Annual IEEE International Workshop on High Level Design Validation and Test (HLDVT'02), IEEE Computer Society Press, pp. 45–50.

[15] N. Zoń, S. Gilmore & J. Hillston (2016): *Rigorous graphical modelling of movement in Collective Adaptive Systems*. In Proceedings of ISoLA 2016. To appear.