QUANTICOL

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

http://www.quanticol.eu

D5.2

A CAS-SCEL implementation for smart-city modelling

http://quanticol.sourceforge.net/

Revision: 1.0

Author(s): Cheng Feng, Diego Latella, Michele Loreti, Mieke Massink, Daniël Reijsbergen, Farshad Shams, Mirco Tribastone, Andrea Vandin, and Chris Williams

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

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

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

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

COOPERATION

Executive Summary

This deliverable presents the first release of the CARMA Eclipse plug-in, the software tool supporting the CARMA process algebra and specification language developed in WP4 and discussed in D4.2. We show CARMA in practice, applied to the analysis of a smart-city model. More specifically, we present an executable CARMA implementation of a distributed protocol based on coalitional game theory for power trading and coordination in a smart grid, recently proposed in [ST15] and also discussed in D1.3. In this respect, this contribution complements D4.2. While in D4.2 a model of a public transportation system has been specifically developed to showcase the features of CARMA, here the goal is to demonstrate how CARMA is expressive enough to describe a realistic smart-grid scenario that has been originally studied *independently from* the development of the language. The model is presented in a tutorial style, which serves as a guide to the usage of the CARMA Eclipse plug-in.

In addition, the deliverable discusses a number of *satellite tools* that have been developed during the course of the second reporting period. While the development activities have been mostly decoupled from those of the CARMA Eclipse plug-in, these tools implement techniques that have the potential to be integrated in future releases to enable further forms of analysis than those currently supported for CAS modelling. Specifically, the *Bus Data Visualizer* implements real-data visualisation methods as well as techniques for model validation using measurements; *FlyFast* is a basis for the integration of efficient model-checking techniques; *CRNReducer* implements model-reduction algorithms; finally, the *PALOMA Eclipse plug-in* features deterministic approximation techniques based on moment closures.

All the reported tools can be reached from `http://quanticol.sourceforge.net/`.

Contents

1 Introduction

The goal of WP5 is to provide software support for the techniques developed in the project in order to be able to design and analyse models of smart cities. During the course of the second reporting period, this task has been accomplished along two main directions of research. First, we have developed an initial release of the CARMA Eclipse plug-in. This has been done in close collaboration with WP4, where the theoretical underpinnings of the CARMA language have been investigated. The plug-in uses state-of-the-art Eclipse-based technology such as the Xtext framework for the development of domain specific languages [Xte]. It provides a fully-fledged text editor with syntax highlighting and static analysis as well as support for stochastic simulations and facilities for the visualisation of results.

The CARMA specification language is discussed in detail in D4.2, together with a model that exhibits all the major features of the language. Instead, in this deliverable we present, in a tutorial fashion, the more concrete workflow to be followed with the tool in order to develop and analyse CARMA models. To offer more complementarity with respect to D4.2, which uses a model of smart public transport, here we focus on the other case study envisaged in the project, namely a smart-grid scenario consisting of a mechanism of power trading coordination among micro-grids. Another difference is that while the case study of D4.2 has been written directly in CARMA, the model presented here is reported in D1.3, taken from [ST15] where it has been developed in the mathematical framework of coalitional game theory, thus independently from high-level specification languages such as CARMA. We believe that this *unanticipated* ability to capture such a case study in a concise manner represents a significant added value for CARMA.

The second direction of research, in line with the plan set out in the project's Description of Work, concerns activities focussed on the development of further "satellite" applications and software libraries for the analysis and verification of smart-city models. In this deliverable we report on four software tools that provide different analysis techniques. The *Bus Data Visualizer* has been developed to support the automatic derivation of *patch-based* models from real data (specifically, GPS measurements), following the procedure described in [RG14; RGH14]. *FlyFast* offers an efficient *on-the-fly* algorithm for model checking PCTL formulae, as described in [LLM15], including scalable mean field approximation. *CRNReducer* provides algorithms for the exact reduction of ordinary differential equations (ODEs) arising from chemical reaction networks with mass-action semantics, implementing the results of [Car+15] (and reported in D3.2). Finally, the *PALOMA Eclipse plug-in* supports the recently proposed PALOMA process algebra [FH14] with a novel ODE approximation technique based on moment closures [FHG] (also reported in D3.2).

Although the development of these applications has proceeded separately from that on the CARMA Eclipse plug-in, since often disjoint teams of developers have been involved, all the work in WP5 has followed the overall common strategy of planning for the potential of interoperability and integration. To accomplish this, both FlyFast and the PALOMA Eclipse plug-in have been developed within the Eclipse framework. The user can already work with them in the same development environment; furthermore, the plug-in communication facilities of Eclipse can be leveraged in the future to allow for a tighter integration between the tools. The Bus Data Visualizer can be accessed from the Web, which allows for straightforward integration with any other tool. Finally, CRNReducer is a pure-Java application, hence it can be seamlessly included (for instance, as a library) in any Java project.

This deliverable is organised as follows. Section 2 summarises the original model of power trading coordination in smart grids presented in [ST15]. Section 3 offers a tutorial on how to encode the model in CARMA using the Eclipse plug-in. The theoretical results in [ST15] of convergence of the model to a stable state and of non-uniqueness of such stable state are backed by numerical results provided by the plug-in. Section 4 provides an overview of the satellite software tools. Finally, Section 5 concludes with some final remarks and an outlook for future work. All the reported tools can be reached from `http://quanticol.sourceforge.net/`.

```
// Outer algorithm:  Power loss minimization dynamic learning process:
repeat
   // Inner algorithm:  Coalition formation:
   repeat
     ...
   until (stable state);
   //The set of formed coalitions = power distribution conditions are satisfied;
until (stable state);
//Power loss is minimized
```

Tab. 1: The global algorithm of power trading strategy.

2 Case Study: Power Trading Coordination in Smart Grids

Motivation. In [ST15], the problem of power trading coordination among "micro-grids" (MGs) (e.g., solar panels, wind turbines, PHEVs, etc.) is investigated (see also D1.3). The motivation is that in traditional power distribution models, consumers acquire power from the central distribution unit (CDU), while MGs in a smart power grid can also trade power between themselves. Hence, whenever some MGs have an excess of power while others have a need for power, it might be beneficial for these MGs to exchange energy with one another instead of requesting it from the main CDU.

The advantage of such a local exchange is twofold:

- The energy exchange between nearby MGs can significantly reduce the amount of power that is wasted during the transmission over the distribution lines;
- Performing a local exchange of energy contributes further to the autonomy of the MG system while reducing the demand and reliance on the main electric grid.

This problem fits well the CAS framework because the scenario involves collectives (MGs) which need to perform adaptations since the excess/demand for power can depend on the current weather conditions as well as the usage profile.

The algorithm in [ST15] aims at minimising the amount of dissipated power during generation and transfer using an approach that combines coalitional game theory and dynamic learning. In particular, the coalitional game theory-based algorithm finds a set of *coalitions* of MGs that achieves a feasible power distribution condition where all the surplus power from MGs is absorbed by the network and, dually, all the deficit power of MGs is provided by the network. The dynamic learning algorithm, instead, modifies the set of coalitions in order to minimize the power loss. The two approaches are nested, as illustrated in Table 1. Here we briefly sketch the main ingredients that are relevant for the presentation of this approach as a CARMA implementation. In particular, for the sake of simplicity the CARMA model presents only the coalitional game theory-based algorithm (inner step) that leads to a feasible power distribution among MGs in a distributed fashion.

Model. Consider a distribution network composed of a single CDU that is connected to the main grid as well as to M MGs, denoted by the set $\mathcal{M} = \{1, \ldots, M\}$. We denote the CDU by the index 0. At any given time point, each MG m has a residual power load Q_m which is defined as the difference between the generated power and the overall demand (of its own area). A positive quantity represents the surplus power that the MG can transfer to other MGs or to the CDU, whereas a negative quantity represents the deficit power the MG needs to acquire from other MGs or from the CDU. MGs that are able to exactly meet their demand, do not participate in any energy exchange. We denote *suppliers* ($Q_m > 0$) and *demanders* ($Q_m < 0$) with $\mathcal{M}^+$ and $\mathcal{M}^-$, respectively, such that $\mathcal{M} = \mathcal{M}^+ \cup \mathcal{M}^-$.

Problem statement. The objective is to form a set of not necessarily disjoint and possibly singleton coalitions of MGs denoted by $\boldsymbol{\mathcal{K}} = \{\mathcal{K}_1, \ldots, \mathcal{K}_K\}$. MGs in each coalition can trade power between themselves and with the CDU. Each non-singleton coalition consists of *one* supplier which serves the assigned demanders. On the other hand, the MG in a singleton coalition only trades directly with the CDU. A coalition structure will be formed only if the following two conditions are satisfied:

i) surplus power quantities of all suppliers are completely loaded;

ii) deficit power quantities of all demanders are completely served.

Let us denote by P_{ij} the quantity of power the MG i loads and transfers over the transmission line $i \to j$, and by L_{ij} the amount of dissipated power over the transmission line $i \to j$. So, $P_{ij} - L_{ij}$ is the amount of power the MG j receives from the transmission line $i \to j$. Using these quantities, the above requirements can be encoded in the following "power-distribution conditions":

$$\begin{cases} \sum_{n \in \mathcal{M}^-} P_{mn} + P_{m0} = Q_m \quad \forall m \in \mathcal{M}^+ & \text{(1a)} \\ \sum_{m \in \mathcal{M}^+} (P_{mn} - L_{mn}) \;+\; (P_{0n} - L_{0n}) = -Q_n \quad \forall n \in \mathcal{M}^- & \text{(1b)} \end{cases}$$

Coalition formation. For coalition formation, the goal is to determine the best amount of loaded power at all the uni-directional transmission lines. The algorithm is a dynamic learning process where each P_{mn} is a *learner* that individually decides the amount of loaded power at the sending-end terminal. By doing so:

- A supplier $i \in \mathcal{M}^+$ and a demander $j \in \mathcal{M}^-$ will be in the same coalition if the supplier i loads some power over the transmission line $i \to j$;
- An MG $m \in \mathcal{M}$ forms a singleton coalition if it trades its whole power quantity with the CDU.

A coalition structure will be formed if and only if the power-distribution conditions in (1) are satisfied. As can be seen, the value of P_{m0} appears only in the first condition (1a), the value of P_{n0} in the second condition (1b), and the value of P_{mn} in both conditions of (1). From the first condition (1a), the following equality is derived:

$$P_{m0} = Q_m - \sum_{n \in \mathcal{M}^-} P_{mn} =: C_{m0} \quad \forall m \in \mathcal{M}^+ \tag{2}$$

Essentially, the equation is rewritten in such a way that each transmission-line power is expressed as a function of the others. The use of the symbols C_{i0} plays a role in the dynamic learning process. The interpretation of these is that they are *the constants* that satisfy the power-distribution conditions.

In a similar fashion we express these constants for (1b), after simple algebraic manipulations:

$$\begin{cases} C_{0n} = -Q_n - \sum_{m \in \mathcal{M}^+} P_{mn} + L_{0n} + \sum_{m \in \mathcal{M}^+} L_{mn} \qquad \forall n \in \mathcal{M}^-; \\ C_{mn} = 0.5 \left(Q_m - Q_n - \sum_{n \neq n' \in \mathcal{M}^-} P_{mn'} - \sum_{m \neq m' \in \mathcal{M}^+} P_{m'n} - P_{m0} - P_{0n} + L_{0n} + \sum_{m' \in \mathcal{M}^+} L_{m'n} \right) \forall m \in \mathcal{M}^+, n \in \mathcal{M}^-. \end{cases} \tag{3}$$

These values must be learnt iteratively by the algorithm, which updates the *tentative powers* P_{ij}. So, the power-distribution conditions are satisfied if all equalities $P_{ij} = C_{ij}$ hold.

For each learner to verify these conditions, the temporary power values of all other learners must be known. The rationality of each learner is to update its own value P_{ij} in order to achieve the value

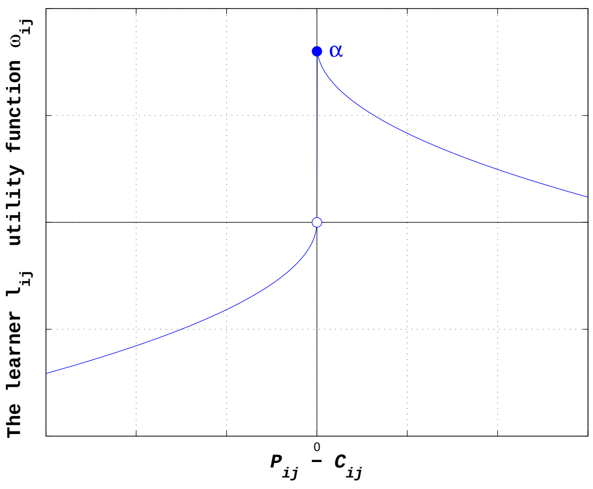


Fig. 1: Learner utility, ω_{ij}, as a function of $P_{ij} - C_{ij}$ with $\alpha \gg 0$.

of C_{ij} *exactly*. At each iteration of the learning process, each learner updates its power value according to the following *learning utility function*:

$$\omega_{ij} = -\sqrt{|P_{ij} - C_{ij}|} + \alpha \cdot \mathrm{u}\left(P_{ij} - C_{ij}\right) \tag{4}$$

where $\mathrm{u}\left(\cdot\right)$ is the step function, and α is a sufficiently large positive constant. If $P_{ij} = C_{ij}$, the learner earns the highest possible payoff, $\omega_{ij} = \alpha$ (see Fig. 1). The learners can distinguish whether the value of P_{ij} is either smaller or larger than C_{ij} only by knowing their own payoffs.

During the coalition formation learning process each learner individually makes its decision as:

$$\tilde{P}_{ij} = P_{ij} - \mathrm{sign}\left(\omega_{ij}\right) \cdot \Delta\tilde{P}_{ij} \tag{5}$$

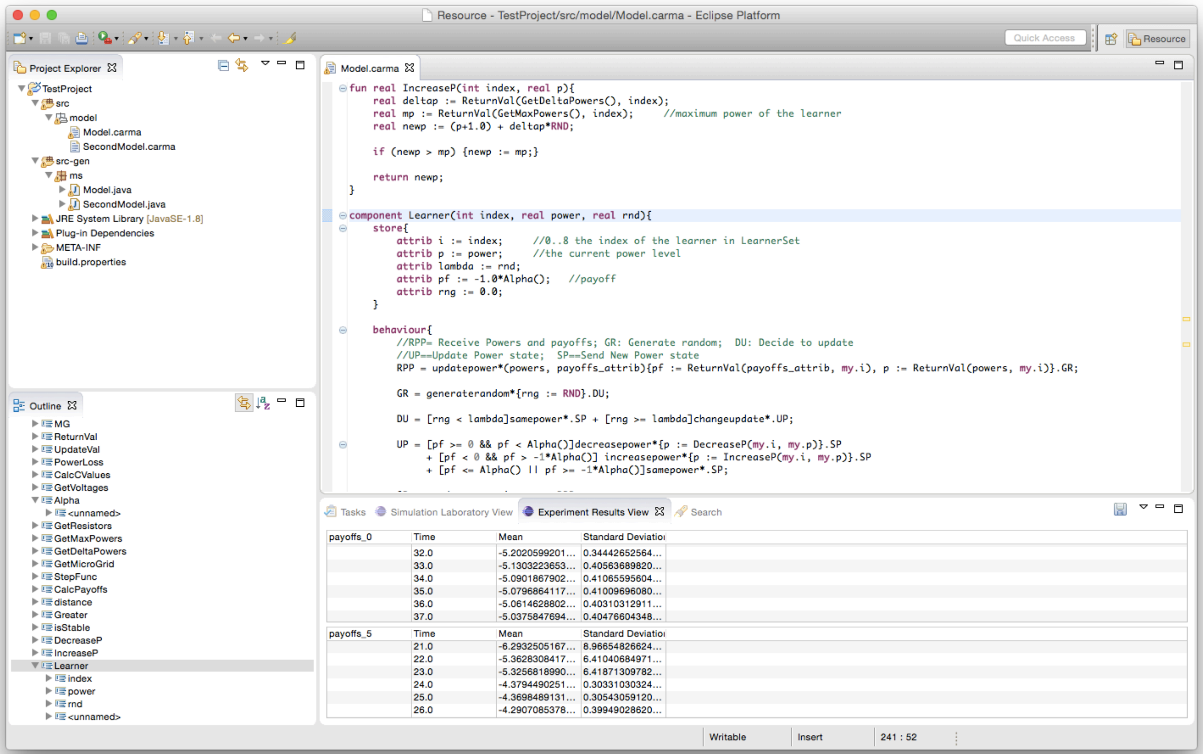
where sign$(\cdot)$ is the sign function, ω_{ij} is the learner's payoff, $\tilde{P}_{ij}$ is the tentative power, and the power step $\Delta\tilde{P}_{ij}$ is the particular outcome (value) of a random variable. If $\omega_{ij} < 0$, then $P_{ij} < C_{ij}$, and the best strategy for the learner is to increase its power so as to increase its payoff. If $\omega_{ij} \geq 0$ the best strategy is on the contrary to decrease its power.

At each iteration, all learners simultaneously and distributively adjust the tentative powers $\tilde{P}_{ij}$'s. If the tentative powers increase all learners' payoffs, then each learner accepts its tentative power and sets $P_{ij} := \tilde{P}_{ij}$. Otherwise the learners discard the tentative powers and keep the old value. Adjusting P_{ij} changes all values of C_{m0} and C_{0n} $\forall m \in \mathcal{M}^+$ and $\forall n \in \mathcal{M}^-$. To reduce the number of occurrences of this event, we modify our algorithm by requesting each learner *not* to update its power value at one iteration with probability $\lambda_{ij} \in [0, 1]$.

In [ST15] it is proven that the algorithm reaches a fixed point at which $P_{ij} = C_{ij}$, for all transmission lines. Here we implement this algorithm as a CARMA model and back this proof of convergence by means of numerical examples.

3 Tutorial on the CARMA Eclipse Plug-in

Using the case study of Section 2, in this section we present a tutorial on the CARMA Eclipse Plug-in. A brief overview of the development environment is provided in Section 3.1, followed by a presentation of the CARMA model of the case study in Section 3.2. Numerical experiments by means of stochastic simulation are reported in Section 3.3.



```
fun real IncreaseP(int index, real p){
    real deltap := ReturnVal(GetDeltaPowers(), index);
    real mp := ReturnVal(GetMaxPowers(), index);     //maximum power of the learner
    real newp := (p+1.0) + deltap*RND;

    if (newp > mp) {newp := mp;}

    return newp;
}

component Learner(int index, real power, real rnd){
    store{
        attrib i := index;     //0..8 the index of the learner in LearnerSet
        attrib p := power;     //the current power level
        attrib lambda := rnd;
        attrib pf := -1.0*Alpha();   //payoff
        attrib rng := 0.0;
    }

    behaviour{
        //RPP= Receive Powers and payoffs; GR: Generate random;  DU: Decide to update
        //UP==Update Power state;  SP==Send New Power state
        RPP = updatepower*(powers, payoffs_attrib){pf := ReturnVal(payoffs_attrib, my.i), p := ReturnVal(powers, my.i)}.GR;

        GR = generaterandom*{rng := RND}.DU;

        DU = [rng < lambda]samepower*.SP + [rng >= lambda]changeupdate*.UP;

        UP = [pf >= 0 && pf < Alpha()]decreasepower*{p := DecreaseP(my.i, my.p)}.SP
            + [pf < 0 && pf > -1*Alpha()] increasepower*{p := IncreaseP(my.i, my.p)}.SP
            + [pf <= Alpha() || pf >= -1*Alpha()]samepower*.SP;
```

payoffs_0	Time	Mean	Standard Deviatio
	32.0	-5.2020599201...	0.34442652564...
	33.0	-5.1303223653...	0.40563689820...
	34.0	-5.0901867902...	0.41065595604...
	35.0	-5.0796864117...	0.41009696080...
	36.0	-5.0614628802...	0.40310312911...
	37.0	-5.0375847694...	0.40476604348...

payoffs_5	Time	Mean	Standard Deviatio
	21.0	-6.2932505167...	8.96654826624...
	22.0	-5.3628308417...	6.41040684971...
	23.0	-5.3256818990...	6.41871309782...
	24.0	-4.3794490251...	0.30331030324...
	25.0	-4.3698489131...	0.30543059120...
	26.0	-4.2907085378...	0.39949028620...

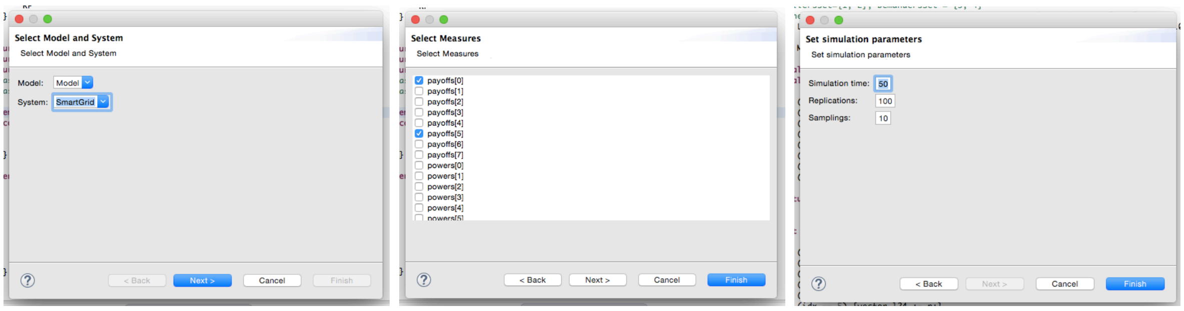
Fig. 2: A screenshot of the CARMA Eclipse plug-in.

3.1 Development Environment

The CARMA specification language is implemented as an Eclipse plug-in using the Xtext framework. It can be downloaded using the standard procedure in Eclipse by pointing to the update site at `http://quanticol.sourceforge.net/updates/`[1]. After the installation, the CARMA editor will open any file in the workspace with the `carma` extension. In addition to the source code, the Sourceforge repository at `https://sourceforge.net/projects/quanticol/` contains sample CARMA model files. The model at `http://sourceforge.net/projects/quanticol/files/EXAMPLES/SmartGrid.carma` is the CARMA model of the case study of Section 2, discussed in the next section. Upon opening the model in the workspace, the user will find a familiar Eclipse text editor featuring syntax highlighting. A screenshot is given in Fig. 2.

Given a CARMA specification, the CARMA Eclipse Plug-in automatically generates the Java classes providing the machinery to simulate the model. This generation procedure can be specialised to enable the use of different kind of simulators. Currently, a simple ad-hoc simulator, is used. The simulator provides generic classes for representing simulated systems (named here *models*). 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 updated in the simulation model and the simulation process continues until a given simulation time is reached. In the classes generated from a CARMA specification, these activities correspond to the *actions* that can be executed by processes located in the system components. Each of these activities in fact mimics the execution of a transition of the CARMA operational semantics (see D4.2). Specific *measure*

[1]Detailed installation instructions can be found at `http://quanticol.sourceforge.net`



Select Model and System

Select Model and System

Model: Model

System: SmartGrid

< Back | Next > | Cancel | Finish

Select Measures

Select Measures

- [✓] payoffs[0]
- [ ] payoffs[1]
- [ ] payoffs[2]
- [ ] payoffs[3]
- [ ] payoffs[4]
- [✓] payoffs[5]
- [ ] payoffs[6]
- [ ] payoffs[7]
- [ ] powers[0]
- [ ] powers[1]
- [ ] powers[2]
- [ ] powers[3]
- [ ] powers[4]
- [ ] powers[5]

< Back | Next > | Cancel | Finish

Set simulation parameters

Set simulation parameters

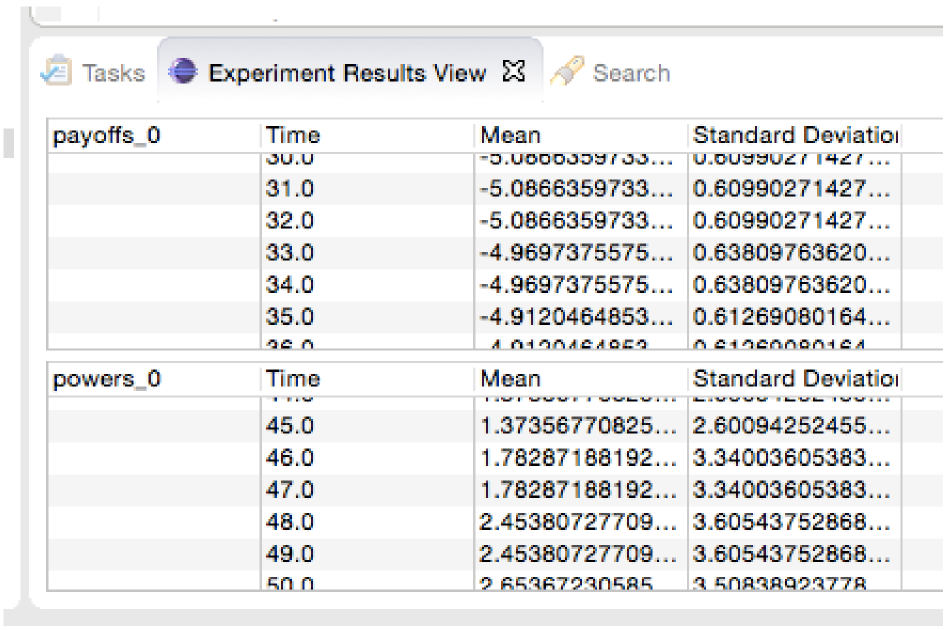
Simulation time: 50

Replications: 100

Samplings: 10

< Back | Next > | Cancel | Finish

Fig. 3: CARMA Eclipse Plug-In: simulation wizard.



payoffs_0	Time	Mean	Standard Deviatio
	30.0	-5.0866359733...	0.60990271427...
	31.0	-5.0866359733...	0.60990271427...
	32.0	-5.0866359733...	0.60990271427...
	33.0	-4.9697375575...	0.63809763620...
	34.0	-4.9697375575...	0.63809763620...
	35.0	-4.9120464853...	0.61269080164...
	36.0	-4.9120464853...	0.61269080164...

powers_0	Time	Mean	Standard Deviatio
	[illegible]	[illegible]	[illegible]
	45.0	1.37356770825...	2.60094252455...
	46.0	1.78287188192...	3.34003605383...
	47.0	1.78287188192...	3.34003605383...
	48.0	2.45380727709...	3.60543752868...
	49.0	2.45380727709...	3.60543752868...
	50.0	2.65367230585...	3.50838923778...

Fig. 4: CARMA Eclipse Plug-In: Experiment Results View.

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[2]. Finally, the framework relies on a *factory-pattern* approach to allow replicated execution of simulations.

To access the simulation features, a user can select the menu *Carma*$\rightarrow$*Simulation*. When this menu is selected, a dialogue box pops up to choose the simulation parameters (see Figure 3). This dialogue box is automatically populated with appropriate values from the model. When the selection of the simulation parameters is completed, the simulation is started. The results are reported within the *Experiment Results View* (see Figure 4). There, the average and standard deviation of the collected values, which correspond to the *measures* selected during the simulation set-up, are reported in a tabular form. These values can then be exported in CSV format and used to build suitable plots in the preferred application.

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

```
record LSet = [ real l10 , real l13 , real l14 , real l20 ,
                real l23 , real l24 , real l03 , real l04 ];

record MG = [ real m1 , real m2 , real m3 , real m4 ];
```

Fig. 5: The block of records.

3.2 CARMA Model

Here, we present the algorithm in Sec. 2 for power trading coordination in smart grids using CARMA. For ease of presentation we consider a simple smart grid network consisting of only $M = 4$ MGs with $\mathcal{M}^+ = \{1, 2\}$ and $\mathcal{M}^- = \{3, 4\}$. The model, however, can be straightforwardly generalised to larger topologies. Here we remark that the possibility of describing the autonomous behaviour of the learners to satisfy the power-distribution conditions (1) is an interesting feature of CARMA.

In Fig. 5 two records are declared:

1) **record** `LSet` declares the set of learners, i.e., the transmission lines. Each field name `lij` refers to the link from the terminal i to the terminal j. The set of learners is ordered as [`l10`, `l13`, `l14`, `l20`, `l23`, `l24`, `l03`, `l04`] with corresponding indices 0 : 7 (from 0 to 7). In the CARMA model, we will declare different variables with type of `LSet` record in order to assign various parameters to each learner, such as power loaded, payoff, value of C_{ij}, and voltage.

2) **record** `MG` declares the set of MGs. Each field name `mi` with $i = 1..4$ refers to MG i and the attribute `qi` determines its quantity of power.

The CARMA model consists of two components. Component `Learner` presents the behaviour of the learner (cf. Fig. 6). Each `Learner` is initialised with the index `i` according to its position in the record `LSet`. The attributes `power` and `rnd` represent the initial power value and the probability value $\lambda_{ij} \in [0, 1]$ of the learner, respectively. The component `ComputingServer` models the central computing unit, i.e., the CDU (cf. Fig. 7).

Each `Learner` starts in state `SP`, as specified in line 30. Thus, at the initial time point, it performs the unicast action `sendpower` (Fig. 6, line 26) to send out its initial power value and its index. This action is synchronised with `ComputingServer`, which starts in state `RP` (Fig. 7, line 46). `ComputingServer` waits for the power values of all learners by counting how many messages it has received in any round (incrementing the attribute `my.rcvnum`). This is possible because the model ensures that no message can be lost when transmitted. Once all messages have been received (by checking the condition `my.rcvnum == 8`), `ComputingServer` computes the values of C_{ij} (according to Eq. 3) and the payoffs of all learners (line 29). The function which performs these computations, `CalcPayoffs`, is not shown for the sake of brevity. It is worthwhile to note that the values C_{ij} are computed at a central unit that has the whole information of the network, since each C_{ij} is a function of some other power values and parameters. Still, each learner updates its current power based on a randomised local decision.

At line 33, the `ComputingServer` checks for the stability of the payoffs, i.e., if all learning payoffs are equal to the highest possible value, setting the flag `a` accordingly. If stability is reached `ComputingServer` transits to the state **nil** and the process terminates (line 35). Otherwise, it decides whether to accept or ignore new power values (line 38). The new power values are accepted if the norm of their corresponding payoffs is strictly greater than the previous values. Finally, state `DC` performs the broadcast output action `updatepower` to send new payoffs and powers to all `Learner`s. After this, it transits to the initial state `RP` where it is ready for a new iteration.

Once each `Learner` receives the new learning payoffs and powers from the `ComputingServer`, it extracts its payoff and power value (Fig. 6, line 16). Then, each `Learner` individually decides whether to update its power value in the next time step, with probability `rng` (Fig. 6, line 20). If it does, it

```
component Learner(int index, real power, real rnd){
  store{
    attrib i := index;      //0..8 the index of the learner in LearnerSet
    attrib p := power;      //the current power level
    attrib lambda := rnd;
    attrib pf := -1.0*Alpha();    //payoff
    attrib rng := 0.0;
  }

  behaviour{
    // RPP: Receive powers and payoffs
    // GR: Generate random
    // DU: Decide to update
    // UP: Update power state
    // SP: Send new power state
    RPP = updatepower*(powers, payoffs_attrib){pf := ReturnVal(payoffs_attrib, my
        .i), p := ReturnVal(powers, my.i)}.GR;

    GR = generaterandom*{rng := RND}.DU;

    DU = [rng < lambda]samepower*.SP + [rng >= lambda]changeupdate*.UP;

    UP = [pf >= 0 && pf < Alpha()]decreasepower*{p := DecreaseP(my.i, my.p)}.SP
       + [pf < 0 && pf > -1*Alpha()] increasepower*{p := IncreaseP(my.i, my.p)}.
         SP
       + [pf <= Alpha() || pf >= -1*Alpha()]samepower*.SP;

    SP = sendpower<my.i, my.p>.RPP;
  }

  init{
    SP
  }
}
```

Fig. 6: The component Learner.

```
component ComputingServer(){
  store{
    attrib a := 0;
    attrib rcvnum := 0;
    attrib stable := 0;
    attrib alphas := [ l10:=Alpha(), l13:=Alpha(), l14:=Alpha(), l20:=Alpha(),
              l23:=Alpha(), l24:=Alpha(), l03:=Alpha(), l04:=Alpha() ];
    attrib newpowers := [ l10:=0.0, l13:=0.0, l14:=0.0, l20:=0.0, l23:=0.0,
              l24:=0.0, l03:=0.0, l04:=0.0 ];
    attrib powers :=[ l10:=0.0, l13:=0.0, l14:=0.0, l20:=0.0, l23:=0.0, l24:=0.0,
              l03:=0.0, l04:=0.0 ];
    attrib newpayoffs := [ l10:=0.0, l13:=0.0, l14:=0.0, l20:=0.0, l23:=0.0,
              l24:=0.0, l03:=0.0, l04:=0.0 ];
    attrib payoffs := [ l10:=-Alpha(), l13:=-Alpha(), l14:=-Alpha(),
              l20:=-Alpha(), l23:=-Alpha(), l24:=-Alpha(), l03:=-Alpha(),
              l04:=-Alpha() ];
  }

  behaviour{
        // RP : Receive the new power of a learner
        // CP : Calculate Payoffs
        // EA : Checks payoffs equal to alpha
        // FS : Finishing state
        // PP : Payoff check
        // UP : Update powers and payoffs
        RP = sendpower(i, p){a := UpdateVal(my.newpowers, i, p),
                 rcvnum := rcvnum + 1, step := 1}.CP;

        CP = [my.rcvnum < 8]waitalllearners*.RP
                 + [my.rcvnum == 8] calcpayoffs*{
                       newpayoffs := CalcPayoffs(my.newpowers), rcvnum := 0}.EA;

        EA = payoffAlphas*{a := isStable(my.newpayoffs)}.FS;

        FS = [a == 1]stablestate*{payoffs := newpayoffs, powers := newpowers}.nil
                 + [a == 0]continue*.PP;

        PP = payoff*{a := Greater(my.newpayoffs, payoffs)}.UP;

        UP = [a == 1]updatepower*<newpowers, newpayoffs>{payoffs := newpayoffs,
                 powers := newpowers}.RP
                       + [a == 0]updatepower*<powers, payoffs>.RP;
  }

  init{
    RP
  }
}
```

Fig. 7: The component ComputingServer.

```
system SmartGrid{
  collective{
    new Learner(0:7, 0.0, 0.2);
    new ComputingServer();
  }

  environment{
    prob{
      default: 1.0; // no message loss
    }

    rate{
      default: 1.0; // models discrete time
    }
  }
}
```

Fig. 8: The block system.

```
fun LSet PowerLoss(LSet newpowersvect){
  LSet pL := new LSet(0, 0, 0, 0, 0, 0, 0, 0); //Power Loss
  LSet resistors := GetResistors();
  LSet voltages := GetVoltages();

  pL.l10 := resistors.l10*Pow(newpowersvect.l10/voltages.l10, 2);
  pL.l13 := resistors.l13*Pow(newpowersvect.l13/voltages.l13, 2);
  pL.l14 := resistors.l14*Pow(newpowersvect.l14/voltages.l14, 2);
  pL.l20 := resistors.l20*Pow(newpowersvect.l20/voltages.l20, 2);
  pL.l23 := resistors.l23*Pow(newpowersvect.l23/voltages.l23, 2);
  pL.l24 := resistors.l24*Pow(newpowersvect.l24/voltages.l24, 2);
  pL.l03 := resistors.l03*Pow(newpowersvect.l03/voltages.l03, 2);
  pL.l04 := resistors.l04*Pow(newpowersvect.l04/voltages.l04, 2);

  return pL;
}
```

Fig. 9: The function power loss.

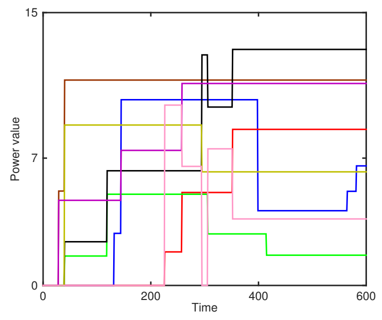
individually updates its power value according to (5): that is, if the payoff is negative then the learner increases the power; otherwise, the power is decreased (lines 22–24). Then it performs the unicast action `sendpower` to send the new power value and its index to `ComputingServer`. The interaction between `Learners` and `ComputingServer` proceeds until `ComputingServer` transits to the state **nil**.

The block **system**, in Fig. 8, initialises the model. It creates eight learners (with indices from `0` to `7`), all with initial power value `0.0` and probability of updating power updating equal to 0.2. The block `environment` states that the probability of receiving messages is `1.0` and that the rates of the actions are equal to `1.0`. This is used to describe a discrete time model using the continuous time semantics of CARMA.

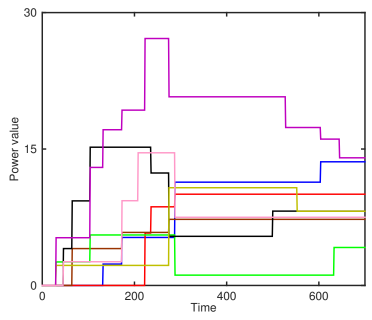
According to the power-distribution conditions (1), to compute the values of C_{ij} the component `ComputingServer` needs to calculate the amounts of power loss L_{ij} across all transmission lines (learners). The function `PowerLoss` (Fig. 9) receives the powers over transmission lines as input and returns a variable with `LSet` record type each of whose attributes determines the amount of power loss over the corresponding transmission line. The function `GetVoltages` (Fig. 10) returns the voltage at sending-end of each transmission line as a variable with `LSet` record type. Similarly, the function `GetResistors` (not shown for brevity) returns resistors of the transmission lines.

```
fun LSet GetVoltages(){
  return [ l10:=20.0, l13:=10.0, l14:=10.0, l20:=20.0, l23:=10.0, l24:=10.0, l03:
     =20.0, l04:=20.0 ];
}
```

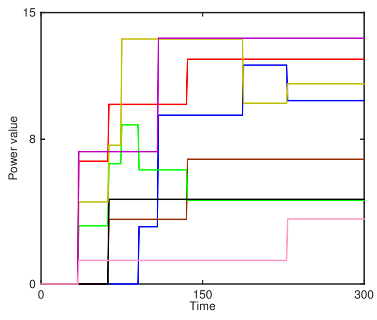
Fig. 10: The function GetVoltages.



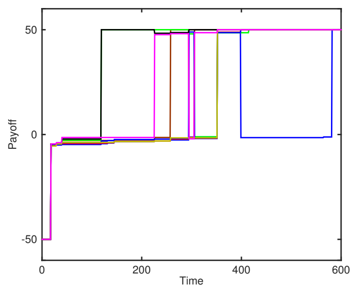
(a) Power values - Simulation 1



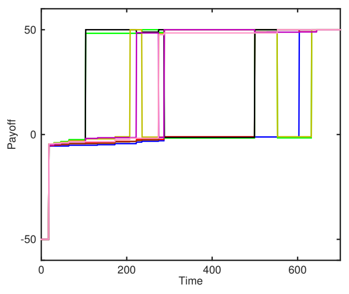
(b) Power values - Simulation 2



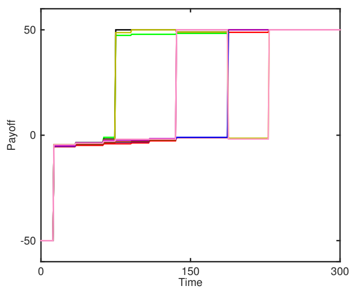
(c) Power values - Simulation 3



(d) Payoff values - Simulation 1



(e) Payoff values - Simulation 2



(f) Payoff values - Simulation 3

Fig. 11: The behaviour of learners during the coalition formation learning process. Each learner is associated with a different line colour in the plots.

3.3 Stochastic Analysis

In this section we illustrate the stochastic simulation features of the plugin. As an interesting application to our case study, we confirm the theoretical results of convergence established in [ST15]. In particular, we show that each simulation runs converges to a "stable state", i.e., a state where all learners have agreed on their output power. Formally, in CARMA this is a state of the underlying continuous-time Markov chain where no outgoing transition is allowed. Furthermore, we show that the stable state is not unique — different runs may lead to different output powers. The convergence results of [ST15] are provided by reducing the algorithm to the run of a discrete time Markov chain. Hence, in our analysis we set all the rates to the same value (as shown in Fig. 8). We remark however, that the continuous-time semantics of CARMA allows us to study further scenarios with different rates among the learners. This could model communication delays in a possible deployment. The numerical results are collectively reported in Fig. 11. Figures 11(a)–(c) show the outputs of the learners as a function of the simulated time for three different runs. Below these figures, for completeness, we show the dynamics of the pay-offs, which all converge to the maximum value, as expected.

As discussed, the final power values are different across the runs. Interestingly, in all cases the total power loss in the network is significantly lower compared to a traditional non-cooperative scheme. This is the scheme where each micro-grid trades the whole quantity of residual power load with the CDU, i.e. $P_{m0} = Q_m \; \forall \; m \in \mathcal{M}^+$ and $P_{0n} - L_{0n} = -Q_n \; \forall \; n \in \mathcal{M}^-$. We measure such improvement by defining the *power loss ratio* as the ratio between the overall amount of dissipated power in the proposed cooperative scheme and that in the traditional non-cooperative distribution model. This ratio is equal to about 13%, 22%, and 18% in the three simulation runs shown above, respectively.

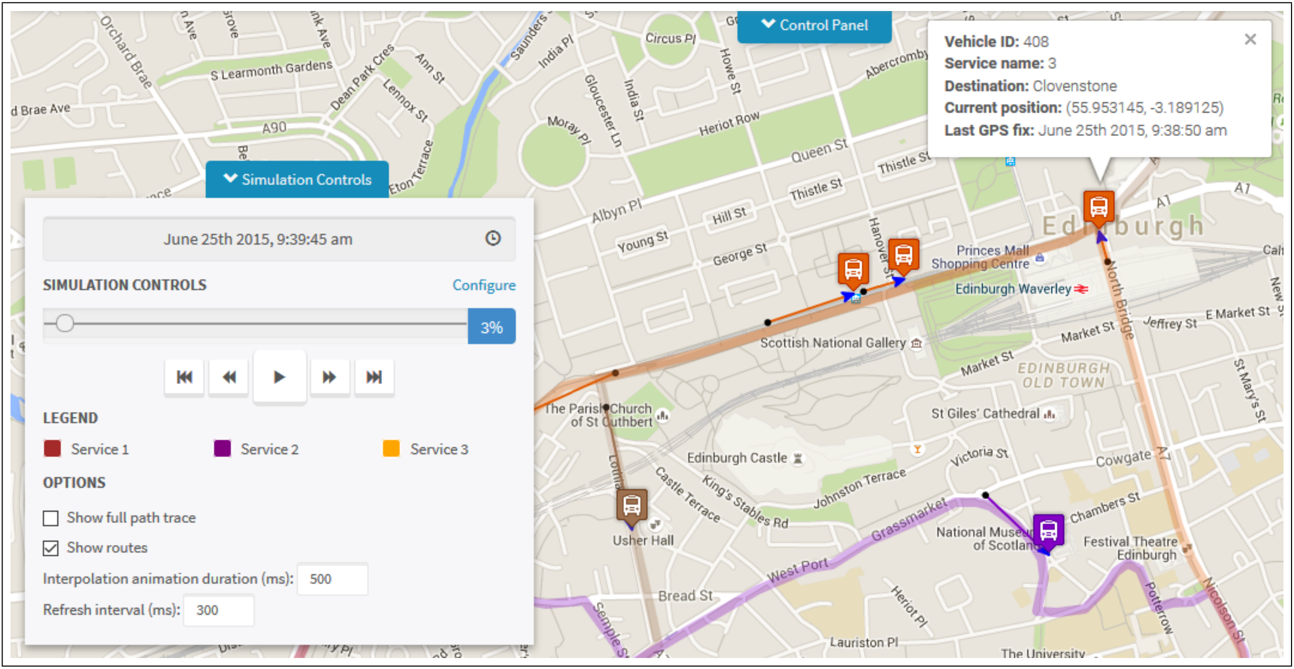
4 Satellite Tools

The CARMA Eclipse plug-in currently offers model analysis by stochastic simulation only. We now present other tools that implement techniques with the potential to be integrated in future releases. The *Bus Data Visualizer* implements real-data visualisation methods as well as techniques for model validation using measurements; *FlyFast* is a basis for the integration of efficient model-checking techniques; *CRNReducer* implements model-reduction algorithms; finally, the *PALOMA Eclipse plug-in* features deterministic approximation techniques based on moment closures.

4.1 Bus Data and Bus Data Visualizer

We have developed a tool for the parameterisation and visualisation of bus models. The tool is based on an API made publicly available by Lothian Buses (at `http://www.mybustracker.co.uk/?page=API%20Key`) that allows developers to access live bus GPS data. It can be made to collect data, which is then stored in a database so that it can be visualised at any time. In addition, it is also able to visualise live data. Unfortunately, there often seem to be 2-3 minutes between GPS updates, in contrast to the 30-40 seconds of the datasets used previously and discussed in Deliverable 5.1.

A screen shot of the tool is displayed in Figure 12. After selecting the desired services, vehicles, and time period using the control panel (which is hidden in Figure 12), the tool allows the user to create animations of the data or to scroll through it manually. For the latter, the user has two options: use the slider directly under 'Simulation Controls' to move quickly to a time point between the first and last entries of the selected dataset, or use the fast forward/rewind buttons below the slider (next to the 'play' button). Step sizes of forward/rewind steps can be configured in the 'Configure' menu to the top-right of the slider. The bus icons on the main visualisation are coloured according to their route, and can be clicked to create a pop-up window with more information.



Vehicle ID: 408
Service name: 3
Destination: Clovenstone
Current position: (55.953145, -3.189125)
Last GPS fix: June 25th 2015, 9:38:50 am

June 25th 2015, 9:39:45 am

SIMULATION CONTROLS Configure

3%

LEGEND

🟥 Service 1 🟪 Service 2 🟧 Service 3

OPTIONS

☐ Show full path trace
☑ Show routes

Interpolation animation duration (ms): 500
Refresh interval (ms): 300

Fig. 12: Screenshot of the Bus Data Visualizer

The tool can also visualise traces of patch-based bus models as described in [VCG14] and [RGH14]. This model type involves one or several buses moving through 'patches' (i.e., segments of the route) such that the sequence of patches is fixed for each bus (i.e., its route). The time spent in each patch is modelled using the phase-type distribution. This type of modelling also allows for the computation of interesting quantities such as the probability of being "on time" or one of the punctuality criteria for frequent services discussed in [RG14].

Parameterisation of a patch-based model involves several stages, of which the most important ones are (1) patch identification and (2) estimation of the parameters of the phase-type distributions. Given a patch structure, the procedure for the latter phase is as follows: we use the dataset to construct a set of observations of the amount of time buses spend in each patch, and then execute a phase-type fitting algorithm implemented, for instance, in the tool HyperStar [RKW13] or the well-known statistical package R. Development of techniques for automated patch identification is ongoing and will be presented as part of Deliverable 2.2.

Implementation details. The visualisation tool is implemented as a web application in JavaScript using jQuery and Bootstrap, with a back-end NoSQL database using Node.js and MongoDB. The tool is currently hosted at an Amazon server at `http://ec2-52-28-155-29.eu-central-1.compute.amazonaws.com:3000/#/tool`. Documentation for installation procedure on a local machine is available at `http://ec2-52-28-155-29.eu-central-1.compute.amazonaws.com:3000/#/doc`.

4.2 FlyFast

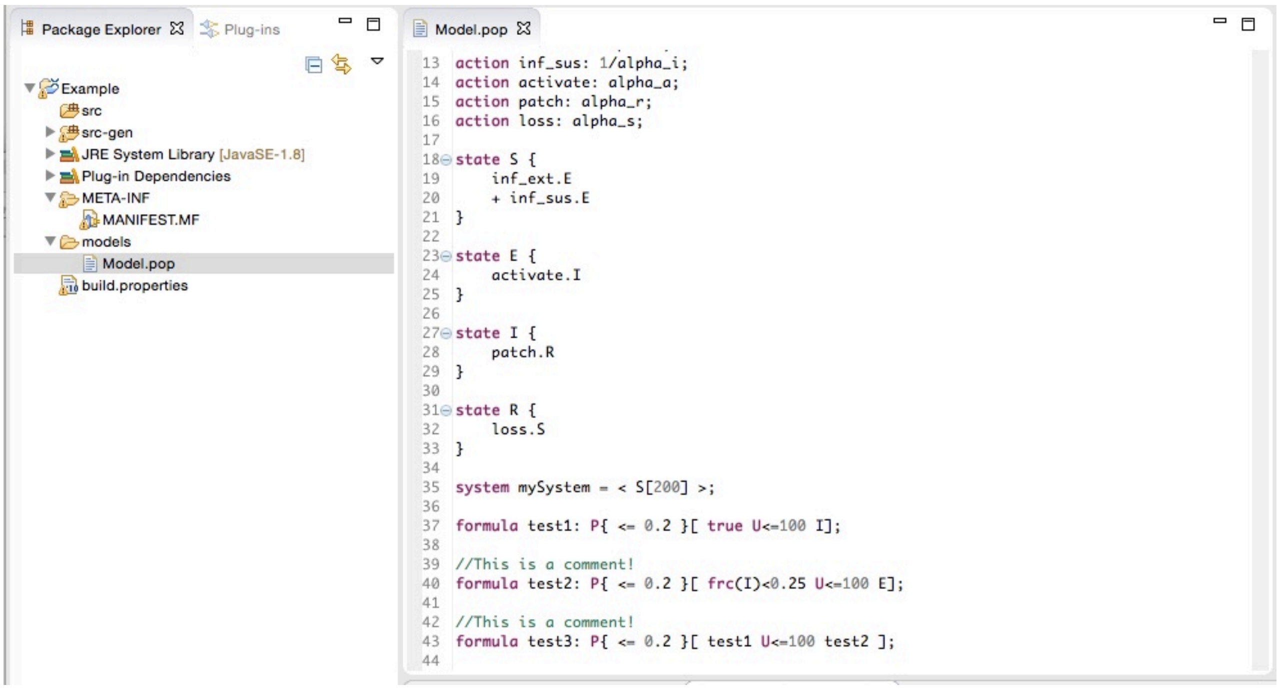
In [LLM15] a new analysis technique that *combines* on-the-fly model-checking and mean-field approximation techniques [BMM07] has been proposed. This proposed technique can be used to verify bounded PCTL (Probabilistic Computation Tree Logic) [HJ94] properties of *selected individuals* in the context of systems consisting of a large number of similar, but independent, interacting objects; a limited form of *global* system properties can be treated as well. The procedure is scalable in the sense that it can be used with huge population sizes, typical of analysis techniques based on mean-field approximation. The proposed technique has been included in a prototype implementation, named FlyFast, and provided within the jSAM framework[3]. The proposed model checker has been also used to study a selection of simple and more elaborate case studies from the field of computer epidemic (also discussed in [Bor+13]) and public transportation [LLM15], in particular bike sharing.

Systems are formalised in FlyFast via a *Population Language*. This is used to describe model composed of N identical interacting objects. At any point in time, each object can be in any of its finitely many states and the evolution of the system proceeds in a clock-synchronous fashion: at each clock tick, each member of the population either executes one of the transitions that are enabled in its current state, or remains in that state (see [LLM15] for all the formal details).

An example of a specification is reported in Figure 13. The provided model describes a network of computers that can be infected by a worm. Each node in the network can acquire infection from two sources, i.e. by the activity of a worm of an infected node (`inf_sus`) or by an external source (`inf_ext`). Once a computer is infected, the worm remains latent for a while, and then activates (`act`). When the worm is active, it tries to propagate over the network by sending messages to other nodes. After some time, an infected computer can be patched (`patch`), so that the infection is recovered. New versions of the worm can appear; for this reason, recovered computers can become susceptible to infection again, after a while (`loss`). The described automaton is reported in Figure 14.

FlyFast provides simulation tools that can be used to analyse the system's behaviour. Two kinds of simulations are available: one based on standard probabilistic simulation and one based on fast simulation. The latter uses a mean-field approximation to simulate the behaviour of a single object in the context of a large class of elements. Moreover, using FlyFast one can check whether a given

[3] `http://j-sam.sourceforge.net/`



```
action inf_sus: 1/alpha_i;
action activate: alpha_a;
action patch: alpha_r;
action loss: alpha_s;

state S {
    inf_ext.E
    + inf_sus.E
}

state E {
    activate.I
}

state I {
    patch.R
}

state R {
    loss.S
}

system mySystem = < S[200] >;

formula test1: P{ <= 0.2 }[ true U<=100 I];

//This is a comment!
formula test2: P{ <= 0.2 }[ frc(I)<0.25 U<=100 E];

//This is a comment!
formula test3: P{ <= 0.2 }[ test1 U<=100 test2 ];

```

Fig. 13: FlyFast population model.

element satisfies a given property when it is located in an environment with a given structure. The FlyFast model checking view (top of Figure 15) can be used to select the states of interest of the single elements and the initial state of the global system represented as the fraction of individuals in each of their local states (initial limit occupancy measure). For example, FlyFast can be used to estimate the probability that an object in state S can get infected within the next t steps. Analysis results are then displayed within the appropriate view integrated in FlyFast (bottom of Figure 15). The population language is also suitable to model systems consisting of multiple classes of objects and several forms of interaction or cooperation between objects can be expressed.

4.3 CRNReducer

Chemical reaction networks (CRN) are a classical model of system dynamics in many natural sciences including biology, chemistry, and ecology. More recently, they have been proposed as a foundational

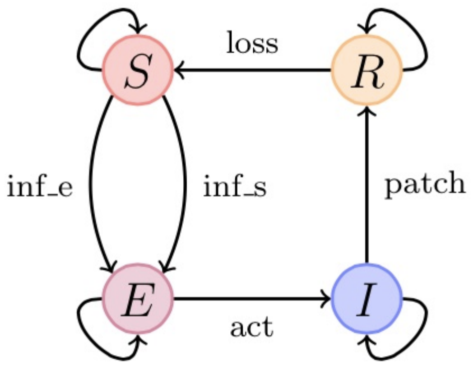
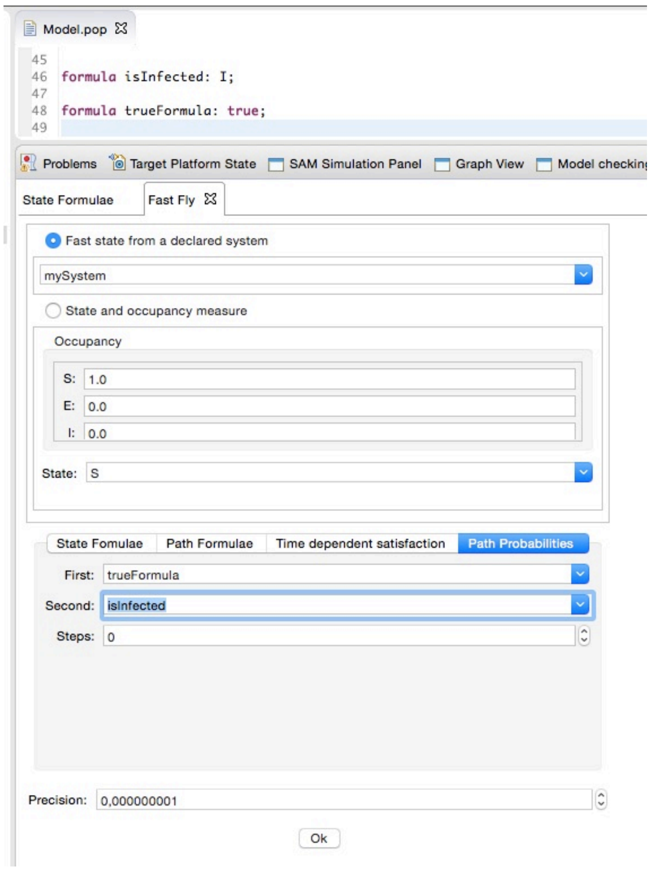


Fig. 14: FlyFast population model.



Model.pop

```
formula isInfected: I;

formula trueFormula: true;
```

Fast state from a declared system: mySystem

State and occupancy measure

Occupancy

S: 1.0

E: 0.0

I: 0.0

State: S

First: trueFormula

Second: isInfected

Steps: 0

Precision: 0,000000001

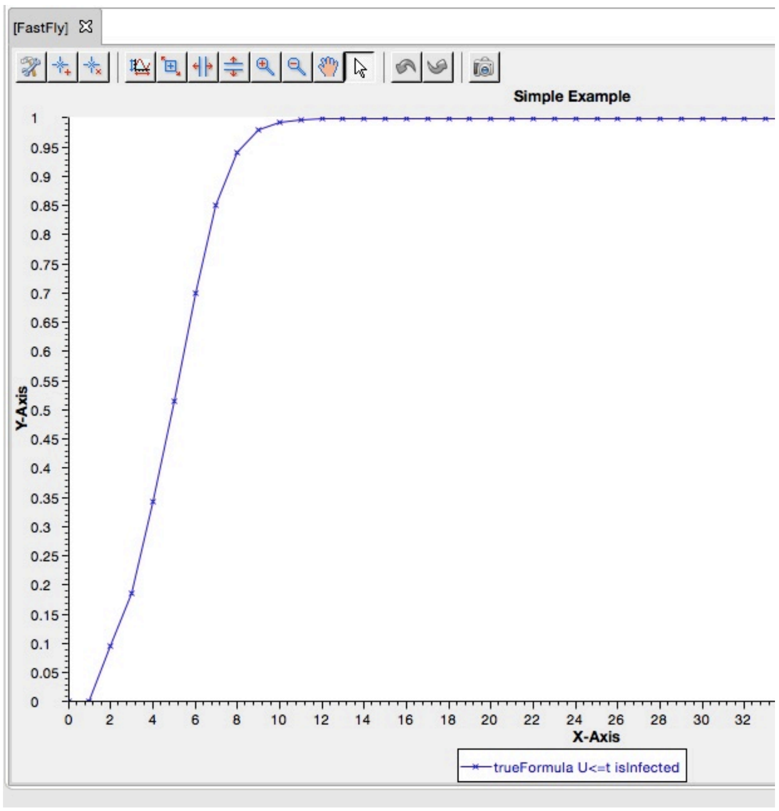


Fig. 15: FlyFast model checking and graph views.

model of concurrent computation, where agents are interpreted as *chemical species* and the patterns of synchronisation, interaction, and communications are represented by *reactions* [RS02]. The well-known kinetic law of mass action, which has been proven formally correct for a class of CRNs [Gil77], is based on a general argument of random encounters between *populations* of different types, which has found applications in the study of engineered systems such as computer networks [Zha+07]. In light of this analogy, we see CRNs as a basic quantitative model for CAS.

CRNReducer is a Java tool for the automatic reduction of CRNs. The tool supports the continuous-state semantics of CRNs which provides a system of ordinary differential equations (ODEs) for a CRN. In particular, CRNReducer exploits two novel reduction techniques defined for such semantics, the Forward and Backward Bisimulations for CRNs [Car+15]. These techniques, discussed in the Deliverable D3.2, bring to the chemical context well-established reduction techniques for models of computation based on the notion of bisimulation.

A Forward CRN bisimulation (FB) is an equivalence among species of a CRN inducing an ordinary lumpable partition of its ODEs. An FB-reduced CRN can be automatically computed, containing a species per block of FB-equivalent species of the original CRN. The solution of the ODEs of such reduced CRN gives the exact sum of the concentrations of the species of each equivalence class. A Backward CRN bisimulation (BB) is similar, but equivalent species have the same solution at all time points; in other words, a BB relates species whose ODE solutions are equal whenever they start from identical initial conditions. A BB-reduced CRN can also be automatically computed, containing a representative species per block of BB-equivalent species of the original CRN.

The tool, available at `sysma.imtlucca.it/crnreducer/`, currently supports CRNs given in the ".net" format generated with the well-established tool BioNetGen [Bli+04], version 2.2.5-stable. We use this format because it allows us to access a rich repository of large-scale CRNs, which we use as benchmarks for our reduction techniques. Listing 1 exemplifies a simple CRN in .net format. Lines 2-9 provide the set of five species of the CRN: $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}$. As sketched in Line 3, each species definition is composed of the identifier of the species, followed by its name and its initial concentration. Lines 12-19 depict the reactions of our example CRN. As sketched in Line 13, each reaction definition contains the identifier of the reaction, followed by two comma-separated multisets of species identifiers, and by the rate of the reaction. The two multi-sets of species identifiers specify the *reagents* and *products* species of the reaction, respectively. For example, the reaction of Line 14 is a *unary* reaction $\mathtt{A} \xrightarrow{6.0} \mathtt{B}$, specifying that instances of species `A` spontaneously evolve into instances of species `B` (modelling, for example changes of its internal state like phosphorylation). Instead, the reaction of Line 18 is a *binary* reaction $\mathtt{A+B} \xrightarrow{3.0} \mathtt{C}$. The latter reaction specifies that, upon collision, a pair of instances of species `A` and `B` might become an instance of species `C` (modelling, for example binding or complexification operations).

Our CRN bisimulations can be applied to any .net model by just running the following command

java -Xmx4068M -jar CRNReducer.jar *technique fileIn fileOut*

where *technique* is either FB or BB to apply our Forward or Backward CRN reduction, respectively. When the BB technique is used, then species are additionally pre-partitioned in blocks of species with the same initial conditions.

The parameter *fileIn* specifies the .net file containing the CRN to be reduced up to our CRN bisimulations, while *fileOut* is the file in which to store the reduced CRN. The latter file also contains information about the computed partition of species, preceding each species with a comment listing all the species of the corresponding block.

Usage example. Let us assume that the CRN shown in Listing 1 is stored in the file `./BNGNetworks/runningExample.net`, then we can reduce it using our Forward CRN reduction by

Listing 1: A simple CRN

```
//List of species
begin species
 //(id, name, initial concentration)
 1 A 1.0
 2 B 1.0
 3 C 1.0
 4 D 1.0
 5 E 1.0
end species

//List of reactions
begin reactions
 //(id, comma-separated reagents ids, comma-separated products ids, rate)
 1 1    2     6.0
 2 2    1     6.0
 3 3,4  3,3,4 5.0
 4 4,5  4,5,5 5.0
 5 1,2  3     3.0
end reactions
```

Listing 2: Output printed by CRNReducer while FB-reducing the CRN in Listing 1

```
java -Xmx4068M -jar CRNReducer.jar FB ./BNGNetworks/runningExample.net
                                      ./BNGNetworks/runningExampleFB.net

Importing: ./BNGNetworks/runningExample.net
Read parameters: 9, read species: 5, read CRNreactions: 5.

FB partitioning ./BNGNetworks/runningExample.net ... completed.
  From 5 species to 4 blocks. Time necessary: 0.007 (s).
Reducing the CRN with respect to the obtained partition ... completed.
  Time necessary 0.003 (s).

The original CRN: ./BNGNetworks/runningExample.net: 5 species, 5 reactions.
The FB reduced CRN: 4 species, 4 reactions.

The current CRN is updated with the reduced one.

Writing the CRN in file ./BNGNetworks/runningExampleFB.net ... completed
```

running

```
java -Xmx4068M -jar CRNReducer.jar FB ./BNGNetworks/runningExample.net
                                      ./BNGNetworks/runningExampleFB.net
```

From the output, provided in Listing 2, we know that the coarsest partition of FB-equivalent species of the CRN has four blocks, and that its computation required 0.007 seconds. The corresponding reduced CRN has been instead computed in 0.003 seconds, and contains four species and four reactions. In particular, the `species` section of the automatically generated FB-reduced CRN stored in `./BNGNetworks/runningExampleFB.net` is provided in Listing 3. Here, each reduced species is preceded by a comment block (lines starting with "//") specifying the represented block of species. In particular, we note that the original species `C` and `E` are FB-equivalent and have been collapsed in the reduced species `C`.

We remark that the FB- and BB-reduced CRNs created by CRNReducer are also provided in the .net format, and are thus amenable to any analysis technique supported by BioNetGen. In other

Listing 3: FB-Reduced species of the CRN in Listing 1

```
//List of FB-reduced species
begin species
//Representative of block (with 1 species):
//   A
1 A 1.0
//Representative of block (with 1 species):
//   B
2 B 1.0
//Representative of block (with 2 species):
//   C
//   E
3 C 2.0
//Representative of block (with 1 species):
//   D
4 D 1.0
end species
```

Original model			*Forward reduction*				*Backward reduction*			
Id	$\|R\|$	$\|S\|$	*Red.(s)*	$\|R\|$	$\|S\|$	*Speed-up*	*Red.(s)*	$\|R\|$	$\|S\|$	*Speed-up*
M1	3538944	262146	4.61E+4	990	222	—	7.65E+4	2708	222	—
M2	786432	65538	1.92E+3	720	167	—	3.68E+3	1950	167	—
M3	172032	16386	8.15E+1	504	122	1.16E+3	1.77E+2	1348	122	5.34E+2
M4	48	18	1.00E–3	24	12	1.00E+0	2.00E–3	45	12	1.00E+0
M5	194054	14531	3.72E+1	142165	10855	1.03E+0	1.32E+3	93033	6634	1.03E+0
M6	187468	10734	3.07E+1	57508	3744	1.92E+1	2.71E+2	144473	5575	3.53E+0
M7	32776	2506	1.26E+0	16481	1281	6.23E+0	1.66E+1	32776	2506	x
M8	41233	2562	1.12E+0	33075	1897	1.12E+0	1.89E+1	41233	2562	x
M9	5033	471	1.91E–1	4068	345	1.04E+0	4.35E–1	5033	471	x
M10	5797	796	1.61E–1	4210	503	1.47E+0	7.37E–1	5797	796	x
M11	5832	730	3.89E–1	1296	217	1.32E+1	6.00E–1	2434	217	7.55E+0
M12	487	85	2.00E–3	264	56	1.88E+0	6.00E–3	426	56	1.31E+0
M13	24	18	1.20E–2	24	18	x	7.00E–3	6	3	1.00E+0

Tab. 2: Forward and backward reductions and corresponding speed-ups in ODE analysis. Speed-up entries "—" indicate that the original model could not be solved; entries "x" indicate that the coarsest bisimulation did not reduce the original model.

words, we provided a backward-compatible extension of BioNetGen with model reduction capabilities.

Tool evaluation. The support for the .net format of BioNetGen [Bli+04] allowed us to evaluate the CRN bisimulations and their implementation in CRNReducer against a wide set of existing models in the literature. We now study their effectiveness in reducing the ODEs of a number of biochemical models from the literature. As discussed, here we use them as numerical benchmarks. We refer to [Car+15] for a biological interpretation of the reductions.

Table 2 lists our case studies: four synthetic benchmarks to obtain combinatorially larger CRNs by varying the number of phosphorylation sites (M1–M4) [SFE11]; a model of pheromone signalling (M5, [SD13]); two signalling pathways through the Fcε complex (M6–M7, [Fae+03; SFE11]); two models of enzyme activation (M8–M9, [BFH09]); a model of a tumor suppressor protein (M10, [BH13]); a model of tyrosine phosphorylation and adaptor protein binding (M11, [Col+09; Col+10]); a MAPK model (M12, [KFL12]); and an *influence network* (M13, [Car14]).

The headings $|R|$ and $|S|$ give the number of reactions and species of the CRN (and of its reductions), respectively. The reduction times (*Red.*) account also for the computation of the quotient CRNs. The speed-up is the ratio between the time to solve the ODEs of the original CRN and that of the reduced one including the time to reduce the CRN. Measurements were taken on a 2.6 GHz Intel Core i5 with 4 GB of RAM. The time interval of the ODE solution was taken from the original papers; for M1–M4, where this data was not available, time point 50.0 was used as an estimate of steady state. The initial conditions for the ODEs were also taken from the original papers. The initial partition for FB was chosen to be the trivial one containing the singleton block $\{S\}$ (i.e., no species was singled out). Instead, the initial partition for BB was chosen consistently with the ODE initial conditions; that is, two species may be equivalent only if they have the same initial conditions in the original CRN. This ensured that the backward reduced CRN was a lossless aggregation of the original CRN.

We make three main observations: (i) FB and BB can reduce a significant number of models. In the two largest models of our case studies, M1 and M2, the bisimulations were able to provide a compact aggregated ODE system which could be straightforwardly analyzed, while the solutions of the original models did not terminate due to out-of-memory errors, consistently with [SFE11]. (ii) FB and BB are not comparable in general. For instance, both reduce M5 to 10855 and 6634 species, respectively, while M6 is reduced to 3744 species by FB, and to 5574 by BB. Also, FB was able to reduce M7–M10, while BB did not aggregate. The influence network M13 shows the opposite; in fact, none of the influence networks presented in [Car14] can be reduced up to FB (here we showed M13, which is the largest one from [Car14]). (iii) Models M1–M4 and M12 show that the intersection between FB and BB is nonempty.

4.4 The PALOMA Eclipse plug-in

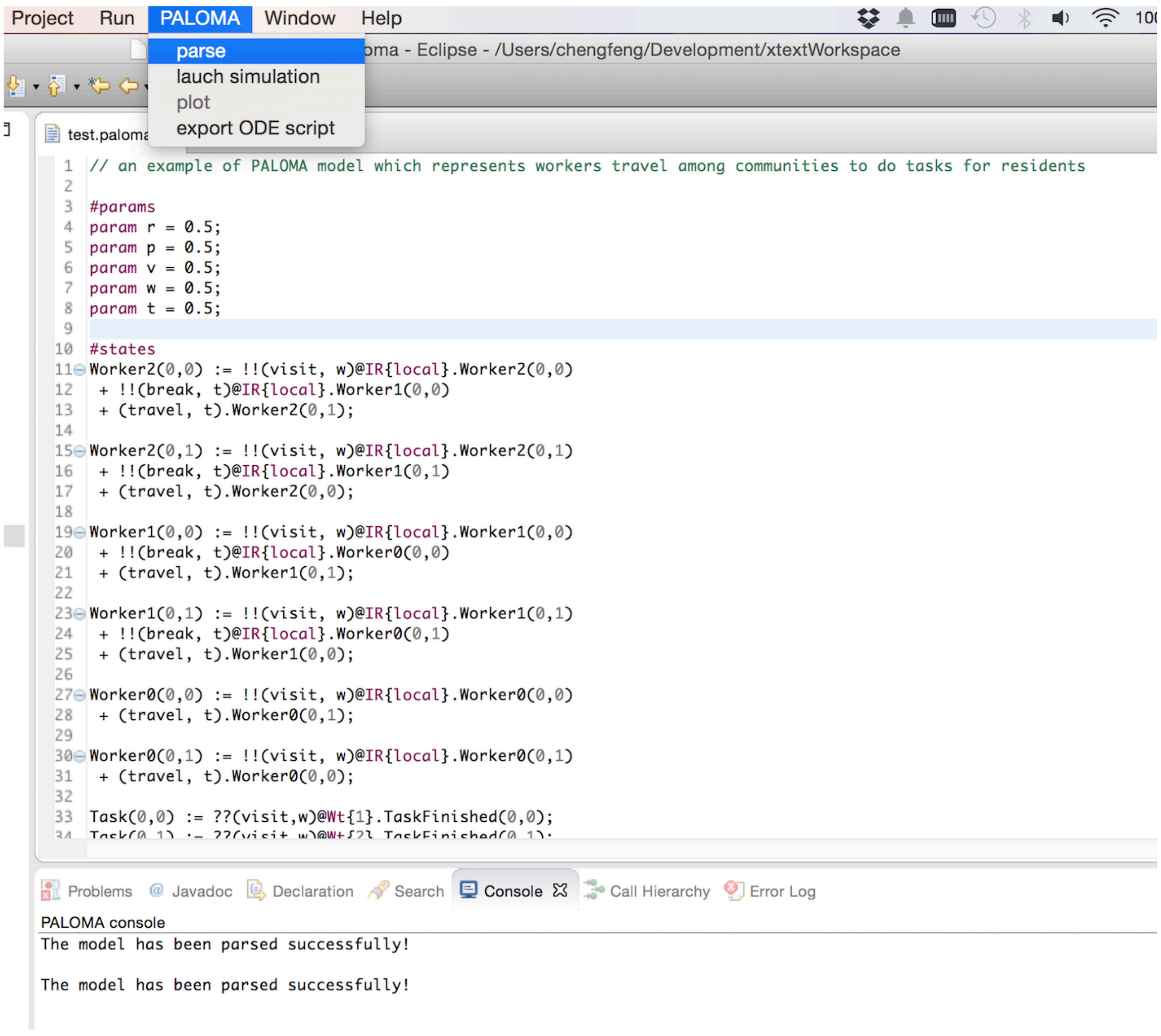
The PALOMA Eclipse plug-in provides a fully-featured development environment for modelling with the recently proposed PALOMA process algebra [FH14], which was discussed in deliverable D4.1. The plug-in consists of:

- An editor for PALOMA models with syntax highlighting functions;
- A simulator which supports population-level stochastic simulation of PALOMA models using Gillespie's algorithm [Gil77];
- Plotting facilities for simulation results;
- A generator which can translate a PALOMA model to directly runnable Matlab scripts for moment-closure analysis using ODEs [FHG].

The source code of the plug-in is available in `https://github.com/cfeng783/paloma`. A user manual about how to install and use the plug-in can be found in `http://groups.inf.ed.ac.uk/paloma/usermanual.pdf`. Once the plug-in is installed, one can create a PALOMA model, parse it and then do time-series analysis of the model by stochastic simulation and moment-closure approximation. In the following we show the typical workflow supported by the tool.

Creating a PALOMA model. To create a PALOMA model, the user only needs to create a file whose name ends in *.paloma*. Three sample PALOMA models can be found in `https://github.com/cfeng783/paloma/wiki#example-models`. In the forthcoming screenshots, we show the concrete syntax for one of these, which can be downloaded from `http://groups.inf.ed.ac.uk/paloma/example.paloma`.

Parsing a PALOMA model. To parse a PALOMA model, the user simply needs to get the mouse's focus on the PALOMA file. Then a *PALOMA* menu will appear on the Eclipse menu bar. By clicking the *PALOMA* menu, a drop-down command list will show up. Clicking the *parse* command, the model will be parsed and the parsing result will show on the console view (see Figure 16).



```
// an example of PALOMA model which represents workers travel among communities to do tasks for residents

#params
param r = 0.5;
param p = 0.5;
param v = 0.5;
param w = 0.5;
param t = 0.5;

#states
Worker2(0,0) := !!(visit, w)@IR{local}.Worker2(0,0)
 + !!(break, t)@IR{local}.Worker1(0,0)
 + (travel, t).Worker2(0,1);

Worker2(0,1) := !!(visit, w)@IR{local}.Worker2(0,1)
 + !!(break, t)@IR{local}.Worker1(0,1)
 + (travel, t).Worker2(0,0);

Worker1(0,0) := !!(visit, w)@IR{local}.Worker1(0,0)
 + !!(break, t)@IR{local}.Worker0(0,0)
 + (travel, t).Worker1(0,1);

Worker1(0,1) := !!(visit, w)@IR{local}.Worker1(0,1)
 + !!(break, t)@IR{local}.Worker0(0,1)
 + (travel, t).Worker1(0,0);

Worker0(0,0) := !!(visit, w)@IR{local}.Worker0(0,0)
 + (travel, t).Worker0(0,1);

Worker0(0,1) := !!(visit, w)@IR{local}.Worker0(0,1)
 + (travel, t).Worker0(0,0);

Task(0,0) := ??(visit,w)@Wt{1}.TaskFinished(0,0);
Task(0,1) := ??(visit,w)@Wt{2}.TaskFinished(0,1);
```

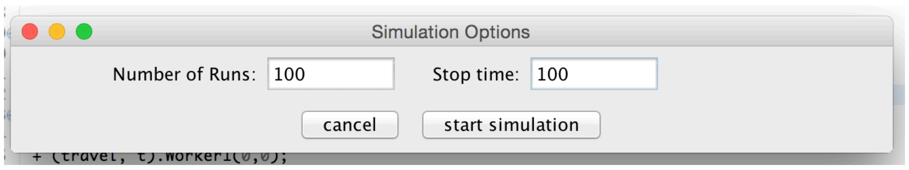
```
The model has been parsed successfully!

The model has been parsed successfully!
```

Fig. 16: Parsing a PALOMA model

Time-series analysis. Once the model has been parsed successfully, it is possible to perform time-series analysis by stochastic simulation using Gillespie's algorithm, and moment-closure approximation by exporting Matlab ODE scripts.

Stochastic simulation is performed by clicking the *launch simulation* command (see Figure 16). A dialogue box allows the user to set the number of simulation runs and the time length of each run, as shown in Figure 17.



Number of Runs: 100 Stop time: 100

Fig. 17: Simulation options

Once the simulation is finished, the *plot* command under the *PALOMA* menu allows the user to choose which variables to plot (see Figure 18).

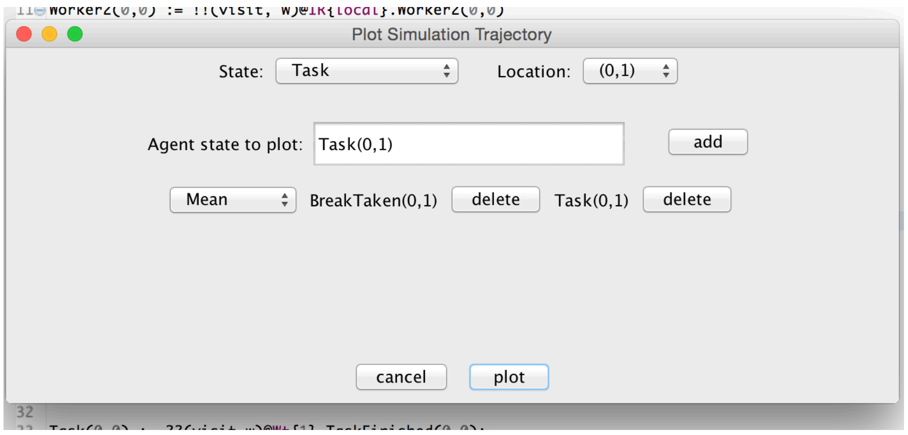


Fig. 18: Choose variables to plot

Clicking *plot* in the pop-up dialogue generates the plots. The simulation results will be displayed in a pop-up graph, which can be saved as a PNG picture (see Figure 19).

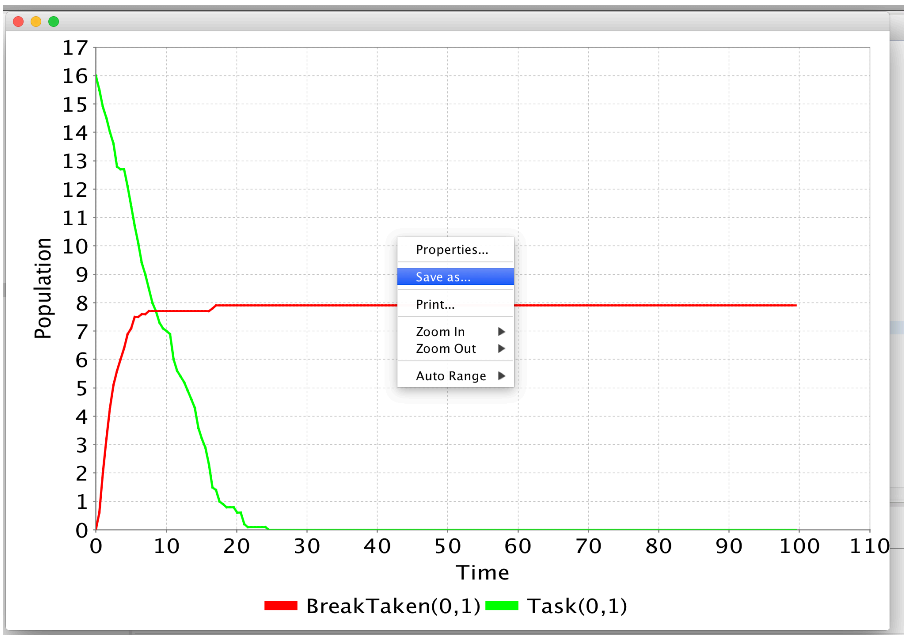


Fig. 19: Plot result

To export Matlab scripts for moment-closure approximation, the user clicks the *export ODE script* command under the *PALOMA* menu. Then, she chooses the variables to analyse, and clicks *export* in the pop-up dialogue (see Figure 20). The generated scripts are directly runnable in Matlab.

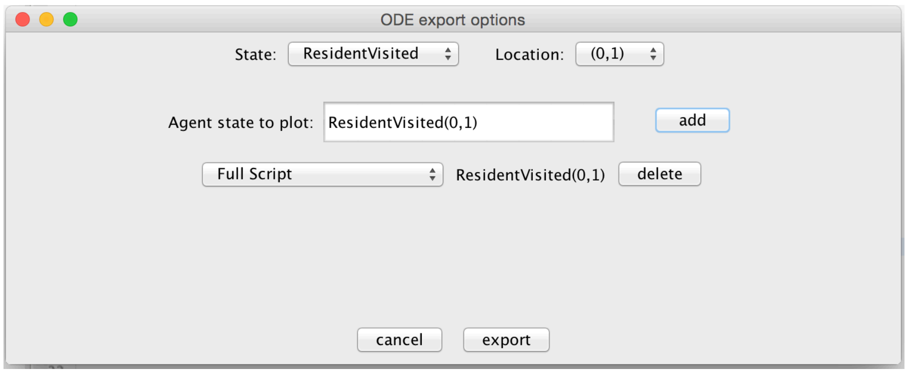


Fig. 20: ODE export options

5 Conclusions and Roadmap

This deliverable has discussed the first release of the CARMA Eclipse plugin, the software tool that supports the CARMA process algebra, developed in close collaboration with WP4. The case study that was used as a running example to show the features of the tool has also allowed us to highlight useful characteristics of the language. Indeed, it was possible to encode a distributed algorithm for power trading coordination in smart grid which has been originally developed in the context of WP1 without having in mind a high-level specification language. This encoding, we believe, brings the algorithm closer to a possible implementation/deployment because it more clearly separates the components involved in the protocol and the communication mechanisms to be adopted. It is interesting to remark that such an encoding is not possible with any other stochastic process algebra we are aware of, because they all focus on behaviour only and lack the possibility of handling (real-valued) data. This feature is clearly crucial in order to properly handle adaptation in this context, whereby a stopping criterion for the algorithm is based on the current values of the power on the transmission lines.

CARMA also offers the opportunity to test variants of the protocol in a rather straightforward manner. For instance, it would be possible to study the speed of convergence under the assumption of imperfect communication links. Interestingly, this can be done without "invasive" changes to the model: only the environment would be modified while leaving the learners' behaviour unaltered; the model description is mostly independent from coalition under study, which allows for easy generalisation to other, and larger, smart-grid topologies.

The satellite tools presented in this deliverable have covered different aspects of smart-city modelling that were originally planned for this project, including data visualisation and model parameter fitting using measurements (in the context of WP2), model order reduction, scalable analysis through ordinary differential equations, and support for model checking (in collaboration with WP3).

Future work in WP5 will build upon what has been presented in this deliverable and will be focussed on augmenting the feature set of existing tools as well as tightening integration between them. In particular we highlight the following priority tasks:

- The bus data visualisation tool will be used as a platform to demonstrate the work in WP2 (specifically from Task 2.3) on automatic patch identification.
- Automatic model reduction is currently available for chemical reaction networks with ODE semantics. There is a plan to integrate with CARMA according to one of the following routes: i) explicit declaration of CARMA models with mass-action type semantics, which are automatically translated into a chemical reaction network; ii) extension of the tool to other forms of semantics. This second route will likely require significant advances in the theory, to be developed in the context of WP3.

- Implementation work will also aim to improve the efficiency of stochastic simulation. In particular, a future extension of the PALOMA plug-in will incorporate the recently proposed model reduction technique for Markov population processes presented in [FH15]. Furthermore we will also identify a dialect of CARMA which is equivalent to PALOMA and so widen the applicability of these techniques to CARMA models also.
- We plan to support ODE-based fluid-flow analysis of CARMA within the plug-in, in collaboration with the research work underway in WP4. This will significantly extend the current analysis supported which is limited to stochastic simulation, and build on results form WP1.

References (from the Quanticol project within the reporting period)

[Car+15] L. Cardelli, M. Tribastone, M. Tschaikowski, and A. Vandin. “Forward and Backward Bisimulations for Chemical Reaction Networks”. In: *CONCUR*. 2015, pp. 226–239.

[FH14] C. Feng and J. Hillston. “PALOMA: A Process Algebra for Located Markovian Agents”. In: *QEST*. 2014, pp. 265–280.

[FH15] C. Feng and J. Hillston. “Speed-Up of Stochastic Simulation of PCTMC Models by Statistical Model Reduction”. In: *EPEW*. 2015, pp. 291–305.

[FHG] C. Feng, J. Hillston, and V. Galpin. *Automatic moment-closure approximation of spatially distributed collective adaptive systems*. Tech. rep. Submitted.

[LLM15] D. Latella, M. Loreti, and M. Massink. “On-the-fly PCTL fast mean-field approximated model-checking for self-organising coordination”. In: *Science of Computer Programming* 110 (2015), pp. 23–50.

[RG14] D. Reijsbergen and S. Gilmore. “Formal punctuality analysis of frequent bus services using headway data”. In: *Computer Performance Engineering*. Springer, 2014, pp. 164–178.

[RGH14] D. Reijsbergen, S. Gilmore, and J. Hillston. “Patch-based modelling of city-centre bus movement with phase-type distributions”. In: *Proceedings of the Seventh International Workshop on Practical Applications of Stochastic Modelling (PASM 2014), Newcastle, England (May 2014)*. 2014.

[ST15] F. Shams and M. Tribastone. “Power Trading Coordination in Smart Grids Using Dynamic Learning and Coalitional Game Theory”. In: *QEST*. 2015.

[VCG14] L. Vissat, A. Clark, and S. Gilmore. “Finding optimal timetables for Edinburgh bus routes”. In: *Proceedings of the Seventh International Workshop on Practical Applications of Stochastic Modelling (PASM 2014), Newcastle, England (May 2014)*. 2014.

References

[BFH09] D. Barua, J. R. Faeder, and J. M. Haugh. “A Bipolar Clamp Mechanism for Activation of Jak-Family Protein Tyrosine Kinases”. In: *PLoS Computational Biology* 5.4 (2009).

[BH13] D. Barua and W. S. Hlavacek. “Modeling the Effect of APC Truncation on Destruction Complex Function in Colorectal Cancer Cells”. In: *PLoS Comput Biol* 9.9 (Sept. 2013), e1003217.

[Bli+04] M. L. Blinov, J. R. Faeder, B. Goldstein, and W. S. Hlavacek. “BioNetGen: software for rule-based modeling of signal transduction based on the interactions of molecular domains”. In: *Bioinformatics* 20.17 (2004), pp. 3289–3291.

[BMM07] J.-Y. L. Boudec, D. McDonald, and J. Mundinger. “A Generic Mean Field Convergence Result for Systems of Interacting Objects”. In: *Fourth International Conference on the Quantitative Evaluaiton of Systems (QEST 2007), 17-19 September 2007, Edinburgh, Scotland, UK*. IEEE Computer Society, 2007, pp. 3–18.

[Bor+13] L. Bortolussi, J. Hillston, D. Latella, and M. Massink. “Continuous approximation of collective system behaviour: A tutorial”. In: *Performance Evaluation* 70.5 (2013), pp. 317–349.

[Car14] L. Cardelli. “Morphisms of reaction networks that couple structure to function”. In: *BMC Systems Biology* 8.1 (2014), p. 84.

[Col+09] J. Colvin, M. I. Monine, J. R. Faeder, W. S. Hlavacek, D. D. V. Hoff, and R. G. Posner. “Simulation of large-scale rule-based models”. In: *Bioinformatics* 25.7 (2009), pp. 910–917.

[Col+10] J. Colvin, M. I. Monine, R. N. Gutenkunst, W. S. Hlavacek, D. D. V. Hoff, and R. G. Posner. “RuleMonkey: software for stochastic simulation of rule-based models”. In: *BMC Bioinformatics* 11 (2010), p. 404.

[Fae+03] J. R. Faeder, W. S. Hlavacek, I. Reischl, M. L. Blinov, H. Metzger, A. Redondo, C. Wofsy, and B. Goldstein. “Investigation of Early Events in FcεRI-Mediated Signaling Using a Detailed Mathematical Model”. In: *The Journal of Immunology* 170.7 (2003), pp. 3769–3781.

[Gil77] D. Gillespie. “Exact Stochastic Simulation of Coupled Chemical Reactions”. In: *Journal of Physical Chemistry* 81.25 (1977), pp. 2340–2361.

[HJ94] H. Hansson and B. Jonsson. “A Logic for Reasoning about Time and Reliability”. In: *Formal Aspects of Computing* 6 (1994), pp. 512–535.

[KFL12] P. Kocieniewski, J. R. Faeder, and T. Lipniacki. “The interplay of double phosphorylation and scaffolding in MAPK pathways”. In: *Journal of Theoretical Biology* 295 (2012), pp. 116–124.

[RKW13] P. Reinecke, T. Kraussand, and K. Wolter. “Phase-Type Fitting Using HyperStar”. In: *Computer Performance Engineering*. Vol. 8168. LNCS. Springer Berlin Heidelberg, 2013, pp. 164–175.

[RS02] A. Regev and E. Shapiro. “Cellular abstractions: Cells as computation”. In: *Nature* 419.6905 (2002), pp. 343–343.

[SD13] R. Suderman and E. J. Deeds. “Machines vs. Ensembles: Effective MAPK Signaling through Heterogeneous Sets of Protein Complexes”. In: *PLoS Comput Biol* 9.10 (2013).

[SFE11] M. W. Sneddon, J. R. Faeder, and T. Emonet. “Efficient modeling, simulation and coarse-graining of biological complexity with NFsim”. In: *Nature Methods* 8.2 (Feb. 2011), pp. 177–183.

[Xte] *Xtext Website*. `https://www.eclipse.org/Xtext/`. Aug. 2014.

[Zha+07] X. Zhang, G. Neglia, J. Kurose, and D. Towsley. “Performance modeling of epidemic routing”. In: *Computer Networks* 51.10 (2007), pp. 2867–2891. ISSN: 1389-1286.