Towards a Goal-oriented Agent-based Simulation framework for High-Performance Computing

Dmitry GNATYSHAK [a], Luis OLIVA-FELIPE [a], Sergio ÁLVAREZ-NAPAGAO [b], Julian PADGET [c], Javier VÁZQUEZ-SALCEDA [a], Dario GARCIA-GASULLA [b] and Ulises CORTÉS [a,b]

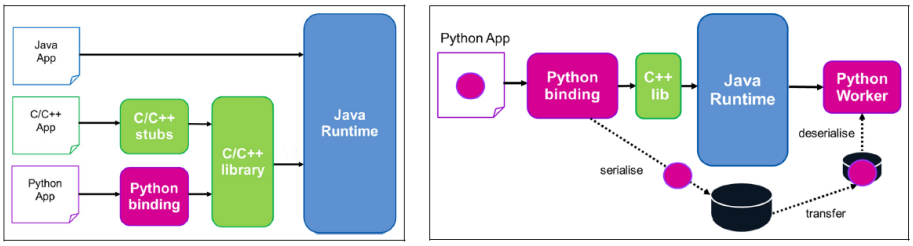
[a] *Universitat Politècnica de Catalunya - BarcelonaTECH (UPC)*
C/Jordi Girona 1-3, E-08034 , Barcelona, Spain
[b] *Barcelona Supercomputer Centre (BSC)*
C/Jordi Girona 1-3, E-08034 , Barcelona, Spain
[c] *Department of Computer Science*
University of Bath, BATH BA2 7AY , United Kingdom

Abstract. Currently, agent-based simulation frameworks force the user to choose between simulations involving a large number of agents (at the expense of limited agent reasoning capability) or simulations including agents with increased reasoning capabilities (at the expense of a limited number of agents per simulation). This paper describes a first attempt at putting goal-oriented agents into large agent-based (micro-)simulations. We discuss a model for goal-oriented agents in High-Performance Computing (HPC) and then briefly discuss its implementation in PyCOMPSs (a library that eases the parallelisation of tasks) to build such a platform that benefits from a large number of agents with the capacity to execute complex cognitive agents.

Keywords. agent-based simulation, high-performance computing, agent platform, multi-agent system, goal-oriented agent

1. Introduction

Agent-based simulation (ABS) faces the conflicting demands of providing larger simulations (in terms of the number of agents) and providing agents with better cognitive and decision-making capabilities (which tend not to scale well in large simulations). This paper is a first attempt at addressing this friction, through the use of High-Performance Computing (HPC), with the aim of enabling scenarios in which large populations of individuals (e.g. traffic simulation, industrial and urban areas along geographical areas) have the ability to perform normative reasoning, planning or even BDI-like behaviour, in order to explore the analysis of phenomena resulting from more detailed behavioural modelling. On the one hand, current HPC-based approaches[12] seem to focus on large simulations with limited interaction or perception or with reactive-like agents. In some cases, planning is even hard-coded. On the other hand, [2] lists a wide range of ABS



(a) PyCOMPSs runtime structure

(b) PyCOMPSs object handling scheme

Figure 1. Aspects of PYCOMPSs

which seem to hardly offer scalability and agents with high cognitive capabilities. This is an indication of an existing trade-off between scalability and agents with complex deliberation processes. In Section 2, we discuss a model for goal-oriented agents in HPC and briefly outline its implementation in PyCOMPSs (a library that eases task parallelisation through annotation) to build such a platform. In Section 3, we present the test scenarios used to evaluate the performance of the platform and demonstrate its capabilities and potential. In Section 4, we summarise the main results, the contributions and we identify the next steps.

2. Proposed Model

This section provides background about the COMPSs HPC framework and the corresponding Python package, and then presents the proposed agent-based simulation model.

2.1. COMPSs and PyCOMPSs

COMPSs [15] is a framework based on the Grid Component Model (GCM)[1] and ServiceSs model[17] developed by the Barcelona Supercomputing Center. Its main purpose is to allow the development of distributed cloud- or grid-based applications without the need to deal with the specifics of underlying execution systems. Thus, it provides an abstraction layer, allowing the development of hardware-configuration-agnostic applications. These applications can be distributed automatically, saving effort in accounting for low-level aspects of the target hardware. COMPSs analyses the data dependencies among the user-specified functions (called *tasks* as in the GCM model) of a sequential program and runs as many of those tasks in parallel as it is safe to do. A set of additional commands and parameters can be used to fine-tune the runtime execution and parallelisation. The core of this runtime is written in Java, but C/C++ and Python interfaces are also available. As all of these commands are language-specific, for the purpose of this work we choose to focus on the Python version: PyCOMPSs[16].

From a coding perspective, the COMPSs interface takes the form of annotations (decorators) in the Python code and several API functions and procedures. The main annotation is `@Task(...)`, where the user specifies the return type (a type or class name) and the data direction of inputs (in, out) for mutable types and classes.The following code shows an example of a PyCOMPSs task annotation:

```
@Task(par1=INOUT, par3=OUT, returns=list)
def my_func(par1, par2, par3):
```

By means of such annotations, we may mark global functions, class methods, and instance methods as tasks to be automatically parallelised.

2.2. Conceptual Model: Multi-agent system elements

We define a **Multi-agent System** $\mathscr{M}$ as the tuple $\mathscr{M} = \{E, \mathscr{A}^+, \mathscr{C}\}$ where:

1. E is an **environment**, in which the agents are situated, that they can perceive, and on which they can act
2. $\mathscr{A}^+$ is a non-empty set of agents
3. $\mathscr{C}$ is a controller that maintains the environment and handles communications between agents

An **Agent** $\mathscr{A}_i$ is defined as $\mathscr{A}_i = \{ID, msgQs, Bh, \mathbb{B}, \mathbb{G}, \mathscr{P}, outAcs\}$ where:

1. $ID = \{AgID, AgDesc\}$ is $\mathscr{A}_i$'s identity data
   (a) $AgID$ is the unique identifier for $\mathscr{A}_i$
   (b) $AgDesc$ is an arbitrary description of $\mathscr{A}_i$
2. $msgQs = \{\mathscr{I}, \mathscr{O}\}$ is a set of the $\mathscr{A}_i$'s message queues
   (a) $\mathscr{I} = \{\ldots, msg_i, \ldots\}$ is the inbox, a set of messages sent to $\mathscr{A}_i$
   (b) $\mathscr{O} = \{\ldots, msg_i, \ldots\}$ is the outbox, a set of messages sent by $\mathscr{A}_i$
   (c) $msg_i = \{AgID_s, AgID_r, performative, content, priority\}$ is a **message** from agent Ag_s to agent Ag_r with the given performative, content, and priority. $performatives$ are FIPA-compliant.
3. $Bh = \{MendR, \mathbb{RG}\}$ is $\mathscr{A}_i$'s **role behaviour**
   (a) $MendR$ is a **means-ends reasoner** used to generate plans (see Section 2.5)
   (b) $\mathbb{RG}$ is a set of goals associated with this role behaviour
4. $\mathbb{B}$ is the set of $\mathscr{A}_i$'s **beliefs**
5. $\mathbb{G}$ is the set of $\mathscr{A}_i$'s **goals**
6. $\mathscr{P} = \{\ldots, ab_i, \ldots\}$ is the **current plan** of $\mathscr{A}_i$
7. $ab_i = \{\ldots, a_{ij}, \ldots\}$ is an **action block**, an ordered set of agent actions, for which we distinguish three types: **internal actions** executed by the agents and intended to change their beliefs **external actions** sent to the controller to be executed by it on the environment, and **message actions** to generate messages to other agents
8. $outAcs$ is a set of external actions to be executed on the environment
   (a) $outAcs_i = \{senderID, a^e\}$ is a tuple of sender ID and an external action

The **Controller** $\mathscr{C}$ is defined by the tuple $\mathscr{C} = \{\mathscr{I}, inAcs\}$ where $\mathscr{I}$ is an inbox for the reception of messages from all the agents and $inAcs$ is the set of all the actions to be applied on the environment.

2.3. Operational semantics of the Agents' deliberation cycle: agent transition rules

In the previous section we have defined the different elements that compose our framework. Using these definitions, we can introduce the operational semantics of our model by means of a set of transition rules. The model assumes that $\mathscr{M}$ evolves as a set of simulation steps. These transition rules describe how the agents' internal states are trans-

$$\frac{\mathbb{B} \xrightarrow{perceive(E)} \mathbb{B}'}{\mathscr{A}_i\{\cdot,\cdot,\cdot,\mathbb{B},\cdot,\cdot,\cdot\},E \rightarrow \mathscr{A}_i\{\cdot,\cdot,\cdot,\mathbb{B}',\cdot,\cdot,\cdot\},E} \quad (1)$$

$$\frac{\mathbb{B},\mathbb{G},\mathscr{O} \xrightarrow{process(h)} \mathbb{B}',\mathbb{G}',\mathscr{O}'}{\mathscr{A}_i\{\cdot,\{\{h,t\},\mathscr{O}\},\cdot,\mathbb{B},\mathbb{G},\cdot,\cdot\},\cdot \rightarrow \mathscr{A}_i\{\cdot,\{\{t\},\mathscr{O}'\},\cdot,\mathbb{B}',\mathbb{G}',\cdot,\cdot\},\cdot} \quad (2)$$

$$\frac{\mathbb{G} \xrightarrow{goal\_check(\mathbb{B},\mathbb{G})} \mathbb{G}'}{\mathscr{A}_i\{\cdot,\cdot,\cdot,\cdot,\mathbb{G},\cdot,\cdot\},\cdot \rightarrow \mathscr{A}_i\{\cdot,\cdot,\cdot,\cdot,\mathbb{G}',\cdot,\cdot\},\cdot} \quad (3)$$

$$\frac{\mathscr{P} = \varnothing \vee \mathscr{P} = Fail \xrightarrow{MendR(\mathbb{B},\mathbb{G},\mathbb{RG})} \mathscr{P}'}{\mathscr{A}_i\{\cdot,\cdot,\cdot,\cdot,\cdot,\mathscr{P},\cdot\},\cdot \rightarrow \mathscr{A}_i\{\cdot,\cdot,\cdot,\cdot,\cdot,\mathscr{P}',\cdot\},\cdot} \quad (4)$$

$$\frac{\mathscr{P}(h,t) \xrightarrow{execute(h)} \mathscr{P}'(t)}{\mathscr{A}_i\{\cdot,\{\mathscr{I},\mathscr{O}\},\cdot,\mathbb{B},\cdot,\mathscr{P},outAcs\},\cdot \rightarrow \mathscr{A}_i\{\cdot,(\mathscr{I},\mathscr{O}'),\cdot,\mathbb{B}',\cdot,\mathscr{P}',outAcs'\},\cdot} \quad (5)$$

$$\frac{Bh \xrightarrow{role\_check(\mathbb{B},\mathbb{G},\mathbb{RG})} Bh'}{\mathscr{A}_i\{\cdot,\cdot,Bh,\cdot,\cdot,\cdot,\cdot\},\cdot \rightarrow \mathscr{A}_i\{\cdot,\cdot,Bh',\cdot,\cdot,\cdot,\cdot\},\cdot} \quad (6)$$

Figure 2. Definitions for the agent transition (rules 1 to 6). We use $\cdot$ to denote elements in the agent tuple that are not modified in the transition.

formed over each single step of the simulation. They show the part of the state that is changed, the function that changes it, and a summary of the transformation.

Agents in our model are goal-oriented deliberative agents inspired by the Beliefs-Desires-Intentions (BDI) Model[11]. Therefore, the agents' reasoning cycle is split into a *perception* phase (where beliefs are updated), a *deliberation* phase (where goals are updated and prioritised), a *means-ends reasoning* phase (where plans are selected/built) and an *actuation* phase. As in many modern BDI-inspired agent languages (e.g., JADEX, 2APL, GOAL), in our model plans are selected directly from beliefs and goals without the need for intermediate intention selection (which forces, in the theoretical BDI model, to only pursue one goal at a time).

The first sub-step in the agent deliberation cycle happens when the agent perceives the current state of the environment. This perception only modifies the agent's beliefs which are then used in the following sub-steps. Formally, this sub-step is formalised in Rule 1 (formulae are provided in Figure 2):

Rule 1: Perceiving the environment, where $perceive(\dots)$ is a user-defined function that transforms the agent $\mathscr{A}$'s beliefs based on the environment state E.

During the second sub-step, the agent processes the incoming messages. This is done sequentially, in an arbitrary order. Message processing may modify the agent's beliefs, as well as its goals. This is formalised in Rule 2:

Rule 2 Message processing, where $process(\dots)$ is a user-defined message processing function that can modify the agent $\mathscr{A}$'s beliefs, goals, and outbox.

In the third sub-step of the deliberation cycle, the agent gets an opportunity to reevaluate and change its own goals. The decision is based on its beliefs and current goals, and only the latter is changed during the process. This transition is defined in Rule 3:

Rule 3 Goal check, where $goal\_check(\dots)$ is a user-defined function that may change the agent $\mathscr{A}$'s goals based on its beliefs.

One of the most important sub-steps is the actual reasoning. In this step, the agent relies on a means-ends reasoner to generate a plan that will be followed by itself afterwards. This step is only performed if the current plan has failed or finished.[1] Formally, this step is defined in Rule 4:

Rule 4 Means-ends reasoner, where $MendR(\dots)$ is a means-ends reasoner that generates plans based on the agent $\mathscr{A}$'s beliefs and goals.

After the plan has been updated, the agent proceeds to execute it. The next action block is extracted from the plan and the set of its actions are run one after another.

Rule 5 Action execution, where $execute(\dots)$ is the action execution function for action h. If h is an internal action, $execute$ runs h to modify the set of agent $\mathscr{A}$'s beliefs; if h is an external action, $execute$ appends h to the set of outgoing external actions; if h is a message action, $execute$ runs h to generate messages to send.

Finally, the agent has an optional choice to reevaluate its role behaviour (i.e. Bh). This sub-step is similar to the goal check and its result can be either a new role, or no changes.

Rule 6 Role check (optional), where $role\_check(\dots)$ is a user-defined function that may modify the agent $\mathscr{A}$'s role behaviour and role goals, based on its beliefs and goals.

2.4. Operational semantics of Controller and Environment: MAS transition rules

After formally defining the internal deliberation process of agents in our framework, we now describe how the Controller changes the environment on each simulation step.

First of all, there is an optional sub-step during which the controller can modify the environment before agents' actions are applied to it. These actions are considered to be commutative. Formally, we can express it in the following way:

Rule 7 Pre-action execution step (optional), where $pre\_step()$ is a user-defined function that modifies the environment before the agents' actions are collectively applied to it.

After that, the controller acts as a message dispatcher, processing all the outgoing messages and passing them to the corresponding recipients in a sequential manner.

Rule 8 Message forwarding, where $fwd_msh(\dots)$ is a function that moves the specified message from the sender's inbox, to the recipient's outbox.

The next sub-step executes, sequentially, all the actions sent by agents.

Rule 9 Action execution, where $execute\_ac(\dots)$ is a function that applies the action h to the environment.

Finally, there is another optional sub-step to modify the environment. It is similar to the first sub-step, differing only in the function used.

Rule 10 Post-action execution step (optional), where $post\_step()$ is a user-defined function that modifies the environment after the agents' actions are applied to it.

[1] See § 2.5 for more details.

$$\frac{E \xrightarrow{pre\_step()} E'}{\mathscr{M}\{E,\cdot,\cdot\} \rightarrow \mathscr{M}\{E',\cdot,\cdot\}} \quad (7)$$

$$\frac{\mathscr{A}_r\{\cdot,\{\{x\},\mathscr{O}\},\cdot,\cdot,\cdot,\cdot,\cdot\} \xrightarrow{fwd\_msg(h,\mathscr{A}_r)} \mathscr{A}'_r\{\cdot,\{\{x,h\},\mathscr{O}\},\cdot,\cdot,\cdot,\cdot,\cdot\}}{\mathscr{M}\{\cdot,\{\ldots,\mathscr{A}_r,\ldots\},\{\{h,t\},\cdot\}\} \rightarrow \mathscr{M}\{\cdot,\{\ldots,\mathscr{A}'_r,\ldots\},\{\{t\},\cdot\}\}} \quad (8)$$

$$\frac{E \xrightarrow{execute\_ac(a)} E'}{\mathscr{M}\{E,\cdot,\{\cdot,\{a,t\}\}\} \rightarrow \mathscr{M}\{E',\cdot,\{\cdot,\{t\}\}\}} \quad (9)$$

$$\frac{E \xrightarrow{post\_step()} E'}{\mathscr{M}\{E,\cdot,\cdot\} \rightarrow \mathscr{M}\{E',\cdot,\cdot\}} \quad (10)$$

Figure 3. Definitions for the MAS transition rules (rules 7 to 10)

2.5. Means-ends reasoning: Hierarchical Task Networks planner

In our base model, we choose not to specify any fixed means-to-ends reasoning model ($MendR(\mathbb{B},\mathbb{G},\mathbb{RG})$ in agent **Rule 4**), leaving it to the user to select one (and to provide the required specification). However, for our implementation of the base model, we choose a planner based on Hierarchical Task Networks (HTN)[8] to instantiate the *MendR* function, on the grounds that HTN planners are a good fit to the means-end stage of the BDI reasoning process[14].There are several HTN-based models, with different representation and planning procedures. As most of these models deal only with high-level planning, at a distance from the practical planning and following task execution, we considered HTN planner models used for actual computer games. These models have been adapted not only for effective abstract-to-concrete facts mapping, but also for re-planning during execution.

We slightly modified [9] to suit both our needs and the COMPSs requirements. These modifications cover the difference in the perception models (the original algorithm used active sensors), and allow the partial sets of method sub-tasks to be accepted as parts of the plan to gain flexibility. In this model, abstract tasks are divided into compound tasks and methods. A compound task consists only of an ordered set of methods while each method consists of an ordered set of compound or primitive tasks, conditions and some additional information. Primitive tasks contain action blocks to be executed, consisting of action of various types, and also some preconditions.

To connect the output of the deliberation sub-step with the HTN planner, we consider the goals provided by the *goal_check* function (**rule 3**) as tasks to be searched in the HTN. By following the model planning rules associated with the (abstract) task [9], we can easily obtain a sequence of actions to be executed based on the current beliefs and goals. After the current plan has finished, we generate a new one, as per **rule 4**. Thus, the only explicit goal type supported currently is the maintain goal[7], but other types of goals (for example, achieve goals or perform goals) can be simulated by the appropriate use of the *goal_check* function. In the case of HTN as a means-ends reasoner, *goal_check*() can easily switch between different compound tasks in HTN (or select a new HTN) as

a method of goal revision. This can be done either by traversing the graph of the HTN, or by generating a new HTN and replacing the old one, although one should always be careful about the belief sets and ontologies used by each planner.

3. Implementation and Results

In the previous section, we introduced the agent-based simulation model and described its design features. This section describes an instantiation of the framework in a concrete implementation and provides some preliminary test results.

3.1. Implementation

The whole system is implemented in Python. It is organised as a package with several modules and a list of main classes that are easily accessible by the user:

Controller: defines the `Controller` class, the main utility entity in the system, responsible for containing the MAS, running PyCOMPSs tasks, forwarding messages, and executing all the utility methods.

Agent: defines the `Agent` class that is a container for the agent description.

Behaviour: defines the `Behaviour` class, that implicitly provides the agent's workflow; the user may subclass this for complex behaviours.

Directory: defines the `Directory` and `DirectoryEntry` classes that are used by the built-in directory facilitator.

HTN: defines all the HTN-related classes, as well as some related functionality; the full list is: `CompoundTask`, `Method`, `PrimitiveTask`, `Effect`, `Action`, `ActionBlock`, `HTNPlanner`, `BeliefSet` and `Conditions`.

Messaging: defines the `Message` and `Messagebox` classes used for agent messaging.

State: defines the `State` structure, used to transfer and work with persistent states.

The `Behaviour` class follows the model presented in Section 2.3 with its instance functions corresponding to the functions in the rules. In complex scenarios, it is expected that the user redefines the following functions:

```
1 state.beliefs = self.perceive(environment, state.beliefs)
2 for message in inbox:
3     state.beliefs, state.planner, reply = self.process(message, state
          .beliefs, state.planner)
4 state.planner = self.goal_check(state.beliefs, state.planner)
5 role = self.role_check(state.beliefs)
```

A simulation is run by initialising a Controller object, registering agents in it, and calling the `run` method. An example of the latter two:

```
1 controller.generate_agent("Test agent", behavior=MyBehavior(),
      beliefs=myBeliefs, init_block=None, planner=myPlanner,
      default_block=None, services=["testing"], register=True)
2 controller.run(num_iter=10, performance=False)
```

An HTN planner can be generated following the structure defined in Section 2.5. The user needs to create a structure of `PrimitiveTasks`, `Methods`, and `CompoundTasks` and use the root `CompoundTask` as a parameter for the planner's initialisation.

Number of agents	Time, s	
	Total	Tasks
10	29.5460	28.5788
50	75.2455	72.3717
100	132.4631	128.8426
200	308.3213	302.9179
300	539.4989	530.3990
500	972.8603	959.3281
1000	1775.0650	1747.6440
2000	2872.2980	2834.4890

Table 1. Results for the agent count test. All the results are averages of 5 replications.

3.2. Performance experiments

In order to test the system's performance, we have designed a number of experiment sets to test different aspects of the system. The overall time performance of the system was evaluated in accordance to 4 parameters: number of agents (100), number of requested processing units (246), number of messages sent (10), and size of messages sent (0Kb). For each series of tests one of the parameters was modified while the others were fixed at default values (the ones inside the brackets above). Due to space limitations we will only focus on the first two.

The Agents' behaviours are guided by a trivial HTN with a single compound task, single method, and single primitive task with 2 actions: send messages and increment step counter belief. On each turn, each agent sent a specified number of messages containing lists of zeroes of specified length to random agents and incremented its "step" counter belief. Upon the reception of a message, each agent incremented its "counter" belief.

All of these experiments, with two exceptions, were performed on the BSC NordIII cluster using the default setting of 256 processors cores per test.

In the case of the test of number of agents (Table 1), the behaviour of the chart fits the expectations. The chart increases linearly w.r.t the increase of the total number of agents (and, correspondingly, the number of PyCOMPSs tasks). Also, even if it is not clear enough from the plot, the controller time noticeably increases due to the fact that the controller has to process additional messages and handle these new agents.

But the most interesting results come from the experiments on the number of processes (Table 2). For them we have two sets of columns: the left ones show the results for the default tests with a specific agent behaviour, the right ones are for the slightly modified version of it, where on each step we have added a one-second delay. We can see that, in the case without delays, there is only a slight dependency between the number of processes and the execution time. But as soon as we imitate the harder tasks that take more time to execute, even just adding one second, the difference becomes massive as we get a clear exponential dependency.

Also please notice that after the number of processes exceeds the number of tasks, the changes in the computation time are almost non-existent, as COMPSs has enough resources to get all the tasks distributed with a minimal delay.

Besides the results on the tested metrics, we have obtained insights about both possible weaknesses of the system and directions for future research. The main issue we faced

Number of processes	Number of nodes	Time, s (w/o delay)		Time, s (with delay)	
		Tasks	Total	Tasks	Total
2	32	223.4435	220.9405	829.7442	826.4027
4	64	159.7769	157.7344	373.3008	370.3567
6	96	161.5091	159.1821	252.5065	250.0795
8	128	179.6115	176.8729	194.2150	192.0736
10	160	164.2693	159.5374	195.0013	191.9677
12	192	125.6789	122.7972	195.0007	192.0937
14	224	128.8578	125.1992	196.3293	193.1141

Table 2. Results for the test for the number of processes. 1 second delays were introduced to each task in the second case.

was disk space usage. As COMPSs transforms custom objects into files for transfer, this puts a strain on the data transfer infrastructure. This may result in exceeding cluster disk space quotas. This limits the scale of the results we are able to achieve (although they still exceed the standard capabilities of the non-HPC BDI platforms).

4. Conclusions

This paper has presented a model for agent-based simulation in HPC and its implementation in PyCOMPSs. Our simulation model is driven by the controller, which can be effectively considered as the synchronisation point for the simulation, following the Bulk Synchronous Parallel model[18]. The main advantage of our model when compared to the closest works in literature[4,6,5,13] is that our model would benefit from simulation domains that require goal-driven agents being able to perform more complex reasoning or planning. Another limitation of these frameworks (except for [4]) is that the agent communication is very limited, based on direct method calls. This kind of communication only works properly within agents executed on the same processor or in the overlapping zones, but not with agents in other processors. We are starting to test our model with a real scenario base on wastewater management of a full river basin with hundreds of pollutant producers and dozens of pollutant processors coordinating their effort to ensure pollution levels are law-compliant. Future research lines include other simulation problems and implementing normative reasoning by incorporating a norm monitor [3] or deontic sensors [10], which would also facilitate the introduction of normative contexts.

Acknowledgements

This work is partially supported by the BSC-IBM Deep Learning Center agreement, the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), the 11th call on the Severo Ochoa Mobility Program in BSC, the Spanish Ministry of Science and Technology through TIN2015-65316-P project and the Generalitat de Catalunya (contract 2017-SGR-1414).

References

[1] Basic features of the grid component model (assessed). CoreGRID Deliverable D.PM.04, 2007.

[2] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, 24:13 – 33, 2017.

[3] Sergio Álvarez Napagao. *Bringing social reality to multiagent and service architectures: practical reductions for monitoring of deontic-logic and constitutive norms*. PhD thesis, Universitat Politècnica de Catalunya, 2016.

[4] S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough. Exploitation of high performance computing in the flame agent-based simulation framework. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 538–545, June 2012.

[5] Nicholson Collier and Michael North. Repast hpc: A platform for large-scale agent-based modeling. *Large-Scale Computing*, pages 81–109, 04 2012.

[6] Gennaro Cordasco et al. A framework for distributing agent-based simulations. In Michael Alexander et al., editor, *Euro-Par 2011: Parallel Processing Workshops*, pages 460–470, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Goal types in agent programming. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06, pages 1285–1287, New York, NY, USA, 2006. ACM.

[8] Kutluhan Erol, James Hendler, and Dana Nau. Htn planning: Complexity and expressivity. *Proceedings of the National Conference on Artificial Intelligence*, 2, 05 1994.

[9] Troy Humphreys. *Exploring HTN Planners through Example*, chapter Architecture, pages 149–167. CRC Press, September 2013.

[10] Julian Padget, Marina De Vos, and Charlie Ann Page. Deontic sensors. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, pages 475–481. AAAI Press, 2018.

[11] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[12] Alban Rousset, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review*, 22:27 – 46, 2016.

[13] Xavier Rubio-Campillo. Pandora: A versatile agent-based modelling platform for social simulation. *Proceedings of SIMUL*, pages 29–34, 2014.

[14] Sebastian Sardina and Lin Padgham. A bdi agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23:18–70, 07 2011.

[15] Enric Tejedor and Rosa M. Badia. Comp superscalar: bringing grid superscalar and gcm together. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 185–193. IEEE, 06 2008.

[16] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. Pycompss: Parallel computational workflows in python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.

[17] Enric Tejedor, Jorge Ejarque, Francesc Lordan, Roger Rafanell, Javier Álvarez Cid-Fuentes, Daniele Lezzi, Raül Sirvent, and Rosa M. Badia. A cloud-unaware programming model for easy development of composite services. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 375–382. IEEE, 11 2011.

[18] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.