Editorial Manager(tm) for Journal of Autonomous Agents and Multi-Agent Systems
Manuscript Draft

Manuscript Number: AGNT-829

Title: A Multi-Agent Simulation-based Architecture for Self-Organizing Systems

Article Type: Manuscript

Keywords: Self-organization; Agent-based Simulation, Architecture, Validation

Corresponding Author: Maira Athanazio de Cerqueira Gatti, Ph.D.

Corresponding Author's Institution: PUC-Rio

First Author: Maira Athanazio de Cerqueira Gatti, Ph.D.

Order of Authors: Maira Athanazio de Cerqueira Gatti, Ph.D.;Carlos José Pereira de Lucena, Ph.D.;Simon Miles, Ph.D.;Michael Luck, Ph.D.

Noname manuscript No.
(will be inserted by the editor)

A Multi-Agent Simulation-based Architecture for Self-Organizing Systems

the date of receipt and acceptance should be inserted later

Abstract In this paper we propose a middle-out approach to the engineering of self-organizing systems achieved with the use of a multi-agent systems simulation-based architecture. The architecture helps on the microscopic analysis and design of self-organizing mechanisms and provides a set of interfaces that helps on the self-organization validation, by validation we mean to check that the required macroscopic properties of the self-organization hold true. Therefore, with this architecture, the self-organization engineer can design, simulate and validate self-organizing applications more easily since it provides an asset base engineers can draw from when developing self-organizing applications. We evaluated the architecture with different applications although we only describe one of them because it is a well known self-organizing application: the Automated Guided Vehicles (AGV) problem. We explain how to model the AGV self-organization on top of the main architecture elements, how to define the required properties of the AGV self-organization and how to instantiate the architecture's validation component. We also compare the performance of the architecture's instantiation with and without the validation component. With this comparison we can check the impact of validating the design of the self-organizing system while its simulation is running.

Keywords Self-organization · Agent-based Simulation · Architecture · Validation

Address(es) of author(s) should be given

Noname manuscript No.
(will be inserted by the editor)

A Multi-Agent Simulation-based Architecture for Self-Organizing Systems

Maira Gatti · Carlos J.P. de Lucena · Simon Miles · Michael Luck

the date of receipt and acceptance should be inserted later

Abstract In this paper we propose a middle-out approach to the engineering of self-organizing systems achieved with the use of a multi-agent systems simulation-based architecture. The architecture helps on the microscopic analysis and design of self-organizing mechanisms and provides a set of interfaces that helps on the self-organization validation, by validation we mean to check that the required macroscopic properties of the self-organization hold true. Therefore, with this architecture, the self-organization engineer can design, simulate and validate self-organizing applications more easily since it provides an asset base engineers can draw from when developing self-organizing applications. We evaluated the architecture with different applications although we only describe one of them because it is a well known self-organizing application: the Automated Guided Vehicles (AGV) problem. We explain how to model the AGV self-organization on top of the main architecture elements, how to define the required properties of the AGV self-organization and how to instantiate the architecture's validation component. We also compare the performance of the architecture's instantiation with and without the validation component. With this comparison we can check the impact of validating the design of the self-organizing system while its simulation is running.

Keywords Self-organization · Agent-based Simulation · Architecture · Validation

1 Introduction

Self-organizing system is a complex system that enables decentralized control. And during the last few years, several definitions and mechanisms have been examined in order to understand how software can be used to model self-organizing systems and

M. Gatti · C.J.P. de Lucena
Department of Informatics, PUC-Rio, Rio de Janeiro, Brazil
E-mail: mairagatti@gmail.com; lucena@inf.puc-rio.br

S. Miles · M. Luck
Department of Informatics, King's College London, London, UK
E-mail: drsimonmiles@gmail.com; michael.luck@kcl.ac.uk

how self-organizing systems can empower the computer science by adding robustness and adaptation.

That robustness and adaptation are achieved in self-organizing systems because self-organization is a process that results from the evolution of a system into an organized form in the absence of an external supervisor [1]. If we call each local entity of this process an agent, a system can be defined as a group of interacting agents that is functioning as a whole and distinguishable from its surroundings by its behavior and an organization is an arrangement of selected parts so as to promote a specific function.

Serugendo et al. also highlight the difference between strong and weak self-organizing systems [2]. Strong self-organizing systems are those systems where there is no explicit central control either internal or external; and weak self-organizing systems are those systems where, from an internal point of view, there is re-organization maybe under an internal (central) control or planning.

In this paper, when we mention self-organizing systems we mean strong self-organizing systems and in this case, due to its decentralized control, it is hard to understand how the local decisions impact on the global behavior of the system where it belongs. So far it is known that multi-agent systems can be used to model self-organization since they share the same properties. In a multi-agent system, we have the agents (local) sensing and acting in an environment (global) [3][4].

Moreover, there is a lack of a suitable architecture that address this problem by enabling design and validation. So far we have in literature sets of self-organizing patterns [5][6] detached from validation methods to check the required properties of the self-organizing system. There is also no integration to the system simulation nor reusable approach to different kinds of self-organizing systems.

The requirements for validation support in a simulation-based self-organizing architecture are related to the support for designing macroscopic properties (which are the desired guarantees of the self-organizing system to be achieved), monitoring and evaluation at the testing phase. By evaluation we mean checking if the self-organization is not producing a misbehavior with regard to the desired guarantees. Therefore interfaces have to be provided at the architecture level so the engineer is able to model the relation between the macro-micro levels and evaluate the macro level w.r.t the simulation at the micro level. In particular, this means that the architecture has to provide abstractions to enable an entity to perceive all the input and output events or actions from the agents and environments at the micro level. The impact of those actions on the environment at the macroscopic level has to be then evaluated during the simulation if possible in an autonomic way, by autonomic we mean without the human intervention.

In this context, this paper presents a multi-agent simulation-based architecture that helps on the design and validation of self-organizing systems. The architecture also addresses multi-environment structures [7] that is a characteristic of many self-organizing systems, such as clusters of networks or biological systems (for instance, extracellular is an environment where the cells live and each cell is an environment where proteins live and so on). The architecture also addresses validation issues by providing suitable interfaces and hooks to plug the validation strategies of the desired guarantees, and an internal mechanism during the simulation that can roll back the simulation if an undesired state is achieved. Undesired states can be achieved if the self-organization mechanism is not properly set. Therefore we can ensure that the design solution of the self-organization does not lead the system to undesired states or it (the design solution) will be rejected after a number of roll backs.

Therefore, the main contribution of this paper is the architecture itself integrated to the provided automated validation features. Along with this architecture, we propose a middle-out approach to the engineering of self-organizing systems.

In the next section, we describe a well known self-organizing problem in order to introduce not only the main issues but also the case study for which we present an evaluation and results later in Section 5. We describe the architecture requirements in Section 3. Section 4 presents in details the proposed architecture meta-model and dynamics. In Section 5, we describe the case study by explaining how to model the self-organization on top of the main architecture elements, how to define the desired guarantees of the self-organization and how to instantiate the validation component. After the case study description, we present the related work on Section 6. Finally, we conclude the paper and present the future work.

2 Problem Description: The Automated Guided Vehicles

The Automated Guided Vehicles (AGV) problem was chosen because it has been used in the existing self-organizing research [8], [9] as a centralized system that can be designed as a self-organizing system to achieve decentralized control and it is capable of evaluating the components of the proposed architecture.

An AGV transportation system uses multiple computer-guided vehicles in warehouses. Each vehicle, or AGV, moves loads (e.g. packets, materials) in a warehouse from the pickup to the drop stations and can only perform a limited set of local actions, such as move, pick up load, and drop load. The goal is to efficiently transport incoming loads to their destination.

The dispatching, which is the process of an AGV receiving a load, and routing, which is the process of an AGV carrying a load from the Pickup Station to the nearest Drop Station to avoid congestions, both require a mechanism that enables aggregation and calculation of extra information while flowing through the warehouses. In [9] the decentralized control for this problem is achieved through the use of the Gradient Fields pattern which takes its inspiration from physics and biology [10], [5]. In physics, the Gradient Fields mechanism can be found in the way masses and particles in our universe adaptively move and globally self-organize their movements accordingly to the locally perceived magnitude of gravitational/electromagnetic/potential fields. The fields create a "waveform" that the particles follow.

In this way, three gradient fields that indirectly guide the automated vehicles can be defined: (i) the Pickup Gradient, generated by the Pickup Stations to notify that there are loads to be dispatched. The farther the gradient is from the source station, the weaker it is, and this strength or weakness helps a vehicle to choose the nearest station because it will choose the strongest one, since it means that this is the closest station; (ii) the Drop Gradient, designed with the same mechanism of the Pickup Gradient, which helps a vehicle to choose the nearest Drop Station once it has a load to be delivered; and (iii) the Vehicle Gradient, a gradient propagated in the environment by all vehicles with the goal of avoiding collisions or congestion. Then if a vehicle perceives this gradient in the neighborhood, it will try to choose the next location with the lowest Vehicle Gradient (since the highest means more vehicles nearby that location).

The first issue that arises from this design solution is: how do we design its implementation? Do we have any architecture from where we can start, without having to reinvent all the basic mechanisms such as the propagation of the information and the

coordination as a result of the propagation? How do we design the entity-entity and entity-environment communications?

Then the second issue is: how do we evaluate this system? What guarantees can we get from our model? In this problem, we need more than guarantees about delivering all packets, we want it to be as fast as possible or at least in a reasonable time. Therefore, we need statements that assure the desired evolution of the average dispatching and routing time, and those statements represent the macro properties or the desired guarantees of the self-organizing system. For instance, suppose that the routing throughput measures the routing rate of a load from the moment a vehicle gets a load and delivers it at the drop station. If the routing throughput decreases it means that more packets are being dispatched than being delivered at the drop off station, on the other hand if it increases it means the opposite which is desired because shows efficiency. But how do we design this evaluation? We need an approach to specify those desired guarantees and they will be the validation method pillars: the self-organization can only be considered valid if the desired guarantees are satisfied.

Moreover, as a self-organizing system it has a non-deterministic behavior which makes even harder to find a good solution efficiently since we are not able to try every state. What kind of architecture and strategy should we use to tackle the non-determinism of self-organizing systems?

We need an architecture that provides an asset base engineers can draw from when developing and evaluating self-organizing applications. This architecture has to provide suitable interfaces and mechanisms that allows the simulation to be automated observable and tuned.

Moreover, this architecture has to help the development of not only this solution, but of any application that share self-organizing dynamics (decentralized control, propagation of information, coordination in response to propagations, etc) in any type of environment and structure. This paper describes such an architecture.

3 The Architecture Requirements

An architectural design helps on the development of a modular program structure and on the representation of the control relationships between modules [11]. It provides a software engineer with a picture of the program structure and behavior. An architecture encourages the software engineer to concentrate on architectural design before worrying about optimizations or code. The main goal of an agent-based architecture for self-organizing systems is to provide an architecture that helps on the design, simulation, and validation of self-organizing systems.

A multi-agent simulation-based architecture for self-organizing systems more specifically must:

- Provide simulation features;
- Provide core components underlying the environment, including different structures and allow creativity in the design of self-organizing multi-agent systems;
- Provide coordination mechanisms to support reuse of self-organizing mechanisms;
- Provide mechanisms for action selection to an integral model that includes support for reusable validation mechanisms.
- Provide validation mechanisms to support the micro-macro relationship understanding.

Therefore, there are four macro requirements to be considered to produce a proper architecture: simulation, coordination, multi-environment and validation support. In this section we describe each of them separately and their motivation as a requirement.

3.1 Simulation Support

At its heart, a simulation-based self-organizing architecture should provide interfaces that allows discrete-event simulation. In discrete-event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time (which can be called a *step*) and marks a change of state in the system. The step exists only as a hook on which the execution of events can be hung, ordering the execution of the events relative to each other. The agents and environment are considered events at the simulation core. It is worth noting the difference of a steppable event and an event that can be fired in the environment. The former is a concept related to discrete event-based simulations, while the latter is the information that allows the coordination and the decentralized control of the system. On the other hand an event fired by an agent can also be steppable. It usually happens, for instance, when this event was propagated in the context of the Evaporation pattern. In this pattern, in some point of time after the event was fired, it has to disappear from the environment. This can only be achieved if the event is treated as a steppable event.

For any discrete-event simulation there are a number of requirements, and a self-organizing architecture based on simulation also inherits those requirements, such us: ability to compress or expand time, ability to control sources of variation, avoids errors in measurement, ability to stop and review, ability to restore system state, and others [12].

The main loop of a discrete-event simulation w.r.t a multi-agent-based simulation is:

Start
.Initialize *Ending Condition* to FALSE.
.Initialize system state variables: the environment and the agents.
.Initialize *clock* (usually starts at simulation time zero).
.Schedule initial events (add to the *Events list*): this means to schedule the environment and its agents.
"Do loop"
.While (*Ending Condition* is FALSE) then do the following:
- Set *clock* to next event time.
- Do next event (step the agent or environment) and remove from the *Events List*.
- Update statistics.

End
.Generate statistical report.

In a self-organizing system, the statistics are the desired guarantees related to the macroscopic properties. Furthermore, the scheduling mechanism should allow for more sophisticated dynamic schedules such that the execution of an event can itself schedule other events for execution in the future. This is particular used by the environment that on each addition or removal of an agent (or any steppable entity) will add the agent to/ remove the agent from the *Events list*, respectively.

3.2 Coordination Support

Among all the possible interaction mechanisms, a simulation-based self-organizing architecture has to support uncoupled and anonymous ones [10]. Uncoupled and anonymous interaction can be defined by the fact that the two interaction partners need neither to know each other in advance, nor to be connected at the same time in the network. Uncoupled and anonymous interaction has many advantages. Summarizing, uncoupled and anonymous interaction is suited in those dynamic scenarios where an unspecified number of possibly varying agents need to coordinate and self-organize their respective activities.

Therefore, the taxonomy created of the events in a simulation-based self-organizing architecture has to relies on what and how information is being communicated: explicit or implicit interaction, directly to the receiver, propagation though neighbors, and so on. Moreover, the agent may react in a different way according to the information type.

Figure 1 illustrates a coordination example of positive and negative feedbacks through the activation of agent or environment actions. Action A of an Agent X produces a growing behavior while can directly or indirectly activate Action B of an Agent Y (it could be also the Agent X itself) and the Action B in turn directly or indirectly activate Action A. While Action C produces a slowing behavior and is activated by Action B and also directly or indirectly activate Action B.

3.3 Multi-Environment Support

Depending on each agent type being developed, the environment types vary. The environment defines its own concepts and their logics and the agents must understand this in order to perceive them and to operate. The environment might be accessible, sensors give access to complete state of the environment or inaccessible; deterministic, the next state can be determined based on the current state and the action, or nondeterministic, and so on.

Each application domain has its own view of what is an environment and what are the functionalities implemented by an environment. In current approaches, each time a different aspect of the application domain is identified this aspect is then appended to the environment in an ad hoc manner. As a result, the environment centralizes all the different aspects of the targeted application. In particular, for a situated environment, an additional element characterizes this agent-environment relationship: the localization function is specifically provided by situated environment. In a situated environment, one can define the location of an agent in terms of coordinates within the environment [8], [13].

A self-organizing system has a structurally distributed environment; in other words, at any point in time, no centralized entity has complete knowledge of the state of the environment as a whole. Furthermore, a designer may decide to model environments using various underlying structures. For example, an environment can be modeled as a graph, a discrete grid, a continuous space or a combination of these (figure 3). In addition, to achieve performance in a cluster or computational grid, or even because of the domain application, the environment can be distributed from a processing perspective if it is designed to be executed in a distributed network. So, the more choices for environment structures, the broader its application in the field of multi-agent simulation systems.

The process of building such a self-organizing system with a multi-environment framework that merges several aspects is made clearer at both the design and implementation levels. So, the agents can exist in several and independent environments. Each environment is concerned only with a specific aspect and can be developed independently from other environments. Therefore, existing environments do not need to be redefined or modified. The environment has a dual role as a first-order abstraction: (i) it provides the surrounding conditions for agents to exist [14], which implies that the environment is an essential part of every self-organizing multi-agent system, and (ii) the environment provides an exploitable design abstraction to build multi-agent system applications.

3.4 Validation Support

The requirements for validation support in a simulation-based self-organizing architecture are related to the macroscopic properties (which are the desired guarantees of the self-organizing system to be achieved), monitoring and evaluation at the testing phase. Therefore interfaces have to be provided at the architecture level so the engineer is able to model the relation between the macro-micro levels and evaluate the macro level w.r.t the simulation at the micro level.

The architecture has to provide abstractions to enable an entity to perceive all the input and output events or actions from the agents and environments at the micro level. The impact of those actions on the environment at the macroscopic level is then evaluated. At this point, the execution flow has to be divided in two flows:

i) if an action had a positive impact in the simulation, i.e., contributed to the desired guarantees, nothing is done and the iteration is back to the beginning;
ii) if an action led the simulation to an undesired state or is deviating the system from the goal state, the Plan module has to be activated. It is responsible for effectively planning the system state backward steps so the Execution module can roll back and the system could converge to a desired or optimum state, if reachable.

Therefore, the architecture has to be able to provide interfaces for the definition of *symmetric actions*[1]so roll back procedures can be performed when needed. It is necessary to provide an interface that will be realized by a domain-based algorithm that operates through the flow of control according to the actions, declare the subset of states that characterize a goal or emergent property (for each), and provide the state evaluation strategy that is based on trends or allowed average behavior.

4 SSOA: A Simulation-based Self-Organizing Architecture

The SSOA (Simulation-based Self-Organizing Architecture) is the multi-agent simulation-based architecture for self-organizing systems proposed in this paper. The two main components of the architecture can be seen inside the dashed box: MESOF and MANAGER (figure 4). While the MESOF provides mechanisms and interfaces to build a

[1] Symmetric actions are reversible actions. This concepts is also used in Online Planning. In this context, the goal of planning is to synthesize a set of actions that, when executed, will achieve the user's goals or required properties. Currently deployed planners for real-world applications are frequently run in an online setting in which plan synthesis and execution run concurrently.

self-organizing system to be simulated. The MANAGER provides mechanisms and interfaces integrated to MESOF that helps on the validation of that system.

During the SSOA instantiation, the software engineer needs to address the microscopic issues: identify agents, environment, objects and structural relationships and the dynamics between them. As it is a simulation-based life-cycle, we also need to instantiate the simulation features of this application. This includes defining in which order the entities are created and started, if they are stepped in a sequence or in parallel.

After this, all the actions an agent or environment can perform have to be identified. There are two types of actions: the *internal actions* are the actions of the agents and the environment, and the *external actions* are the input for the system.

The SSOA is a simulation-based life-cycle, therefore we need to know *a priori* how the simulation of these models will be validated. Therefore we need to finish the SSOA instantiation w.r.t the validation method. This can be accomplished by first modeling a corresponding reversing action or set of actions for each action. SSOA provides a validation method through the MANAGER component that uses those reverse actions so the system can roll back if it enters in a undesired state. An example of reversing action is to move back to a position.

At this point we have modeled all the entities and their dynamics including self-organizing patterns. Now we can start modeling the desired guarantees which we are aiming for, through which we can assess the results of the simulations. We can do this by defining the state variables of the agents and environment. They will be composed to monitor the desired guarantees. We finish the SSOA instantiation w.r.t the validation method by modeling which subset of states of the system need to be matched. Then the MANAGER component will be able to run the state evaluation. This means that it will analyze all the global states already reached and will verify which subset of the states matches the set of guarantees for the macro properties that represents the desired behavior.

The Testing phase consists of running the simulation itself. However before that, it is necessary to provide scenarios. A scenario is a suitable set of parameters for the model. They depend on the kind of the simulation and, as we are dealing with self-organization and distributed control, some parameters are expressed in terms of occurrence rates [6]. The use of scenarios and simulation also enables the engineer to gather meaningful statistics about the macroscopic properties. Since it might be difficult for the engineer to put together a single, all-encompassing scenario, a simulation can be developed using accumulated results from other scenarios to obtain average-case metrics. Therefore, random events are generated according to predefined probabilities. Thus, events that occur very rarely can be assigned very low probabilities while others are assigned higher probabilities, and with the random selection of events this becomes realistic.

The required macroscopic properties are engineered in an iterative process. This is an **iterative method**, because it is possible the desired guarantees defined for the model built in the Design phase can never be satisfied. In this case, the MANAGER component retains the knowledge of all accepted and reverted actions. The MANAGER component has a key function in this process, since it speeds the process by enabling to identify which local behavior should be remodeled, and start the process again. Hence, the design discipline uses the testing results to get feedback and adapt the design to steer toward the required solution.

Moreover the MANAGER component also provides hooks for doing self-configuration in the simulation. Therefore, as we can customize this behavior depending on the model

being analyzed and design the MANAGER component to change the input parameters for the external actions if a set of roll backs was performed. It can speed the simulation process even more in the case of testing several scenarios at once.

4.1 MESOF Component Description

The MESOF component encapsulates a Multi-Environment Self-Organizing Framework. It provides the hierarchy concept of environments in self-organizing multi-agent systems, i.e., it allows the modeling of multiple environments with different structures in a single simulation. MESOF also provides a set of coordination components that assist in the engineering of self-organizing mechanisms.

4.1.1 MESOF Meta-Model

The MESOF meta-model is described in this section. Figure 5 illustrates its structural and hierarchies and is explained through the features that correspond to the architecture requirements as follows:

Simulation Features

The abstract *Simulation* class represents the simulation itself and has control of the simulation. It encapsulates the main environment (represented by the *Environment* class), being able to access its state. Another duty of the *Simulation* class is to give a unique identifier about the current simulation state. It will allow the Manager Component to monitor and rollback the simulation states when needed.

The *Environment* manages the *Schedule* of its entities when it is started by the *Simulator*. And, for each time step, it manages the entrance of *Entity* and schedules each new added entity. The entities being scheduled could be executed in different modes such as an ordered sequence, random sequence or parallel sequence.

Coordination Features

The *Entity* class is an abstract class that represents any entity that exists in the *Environment*. In a situated environment, the *Agent* holds a *Location* in the *Environment*. This class cannot be directly instantiated, rather, to do so through the *Agent* and *Environment* specialized classes.

The abstract *Agent* class represents the agent that can be either an active or reactive entity. It can observe and act in the *Environment* (sensing and producing events), always with a proposal of achieving its goals or reacting to events. An agent is able to: communicate with other agents and the environment, and to move between environments. The abstract *Environment* class is an active entity and, therefore, it is a specialization of the *Agent* class. If the *Environment* is a situated environment, it manages the *Space* in which the agents have a specific *Location*. Each *Location* can be given to an agent, a sub-environment, or events to be sensed by other entities.

An event (*Event* class) is any **information** fired by an *Agent* or *Environment*. It can be sent to the environment directly, to a specific location so all the agents in this location can perceive the event, or directly to an agent. The listeners (*EntityListener*, *EventListener*, *AgentListener* and *EnvironmentListener*) are interfaces that allow any element interested in these entities or events to be notified.

The *AbstractAction* class represents an action that own a source entity (the action performer) and a target entity (the action receptor) that can be either an *Agent* or

an *Environment*. Each subclass *Action* must have one or a set of *ReverseAction*(s) because they will be called by the Manager Component to perform rollbacks in the simulation. An *Action* does not require reverse behavior only if it is not interesting for the Manager Component to analyze it (when it has no impact in the system).

Regarding a situated environment, the coordination is achieved using directly communication (through treating events as messages) and indirectly communication through propagation of events in the neighborhood in 2D/ 3D and discrete/ continuous grid. Moreover, there is a specific type of event, called Positional Event, which can be propagated instead of a regular event. The Positional Event has a time to live in the environment. Therefore, if an agent takes too many time steps to reach the source location of the event, it might have disappeared. This is useful for the Diffusion pattern, for instance, and for its combination with other patterns.

Another important principle of the MESOF Component that allows the coordination to be flexible and fast is that the space of the situated environments are considered sparse fields. Therefore, many objects can be located in the same location and different search strategies can exist for each entity type. The MESOF also provides a set of neighborhood lookups for each environment type such as: get agents at a node/ position, get agents within distance, get events at location.

Furthermore, the *Agent* and *Environment* use the Template Method design pattern in order to implement the invariant parts of the common behavior:

- **Agent: *step* template method**
  This template method first, post all events. Then, for each perceived event, the agent tries to handle this event. If the event was handled, notify all the event and agent listeners.Then, do agent behavior.

```
STEP(s)
input: Simulation s
1  count := events.size
2  for (i := 0; i < count; i++)
3     e := events[i]
4     if (doHandleEvent(e, s))
5         e.notifyHandled(this)
6         for (EntityListener listener in listeners)
7           if (listener instanceof AgentListener)
8               listener = (AgentListener)listener
10              listener.eventHandled(this, e)
11        events[i] := NULL
12        i := i - 1
13        count := count - 1
14  doBehavior(s)
```

- **Agent: *doHandleEvent* "hook" operation**
  Handles an event. Return true if and only if the event was handled. Otherwise, return false. If the event was not handled, it will be kept in the queue so it can be handled at a later step.
- **Agent: *doBehavior* "hook" operation**
  Performs any activities other than handling events. Note that this method is only called when the agent has no event left to be handled.
- **Environment: *doBehavior* override**
  All the subclasses of *Environment* have to call the doBehavior implementation of

the *Environment* class. The *Agent* and *Environment* subclasses might need to use the reference for the simulation, therefore, this override has the simulation as an input. The dead structure is a hash that contains the received and handled events. If the event was received by an entity but not handled, it will not be added to the dead hash. Otherwise, it will be added and at the end will be removed from the location. This algorithm propagates all the event of a specific location to all agents at the same location.

```
doBehavior(s)
input: Simulation s
1  dead := new HashSet<EventV>()
2  for (Location l in locations)
3    content := locEvents.get(l)
4    dead.clear()
5    for (Event e in content)
6      entities := getOtherEntitiesAt(l, e.getSource())
7      for (Entity entity in entities)
8        if ( ((Agent)entity).receiveEvent(e) )
9      if (not e.update())
10       dead.add(e);
11   for (Event event in dead)
12     content.remove(event)
```

Multi-Environment Features

Regarding the multi-environment features, at the meta-model we have the simulator engine that schedules the main environment. All the agents and sub-environments on the main environment are scheduled by the main environment and added to the simulator engine depending on their states. The environment state is dynamic and if one agent leaves the environment or moves itself, the environment state changes.

We have seen that the environment is locally observable to agents and if multiple environments exist, any agent can only exist as at most one instance in each and every environment. In self-organizing systems, the environment acts autonomously with adaptive behavior just like agents and interacts by means of reaction or through the propagation of events.

The meta-model provides the *AgentNetwork* and *EnvironmentNetowrk* abstract classes for situated environment using a graph network, which is represented by the class *Network*. This class handles the addition, removal and search of agents and events in a graph network with a double point location.

The meta-model also provides the *Agent2D* and *Environment2D* abstract classes for situated environment using a discrete 2D double point grid, which is represented by the class *Grid2D*. This class handles the addition, removal and search of agents and events in a double point location.

Regarding the 3D environment, the meta-model provides a 3D continuous space through the *ContinuousGrid* class, and the entities are represented by a triple (x, y, z) of floating-point numbers. All the agent-environment relationships and simulation schedule described for a non-situated environment is reused in these components.

4.2 Manager Component Description

The continuous style box shows that an application can be totally decoupled from the Manager Component (see figure 4), if desired. I.e., the application, which is a self-organizing multi-agent system to be simulated, can instantiate MESOF and run without instantiating and turning on the Manager Component. However, it is a constraint to design the agents and environment to realize the *Action* interface provided by MESOF in order to be able to activate the Manager in the future. The Manager instantiation consists of realizing the *Goal* interface.

4.2.1 Manager Meta-Model

Validation Features

The *Goal* interface defines a desired guarantee (figure 6). In the planning context, a goal is satisfied when one or more state variables have optimum values. A set of goals (*GoalSet* class) can be used when it is necessary to define more than one desired guarantee. The goals are the method pillar. The system can only be considered valid if all goals are satisfied.

When an action is performed, the *Manager* has to evaluate if this action contributes to the goals defined or if it leas the simulation to an undesired state. The *Manager* class is the central class of the Manager Component. This class unifies all the auxiliary resources to monitor and validate the simulation.

The mechanism starts with the *Manager* being notified about an action execution or an environment step (see figure 7). This is possible because the *Manager* realizes the *ActionListener* and *EnvironmentListener* interfaces. After this, the process is divided by two execution flows:

1. If the verification was started because of an action execution, the *Manager* checks if the goal (or set of goals) is satisfied. And the three execution flows can be executed:
   a. If the goal is satisfied, the cycle returns with *success* and the action is *accepted*.
   b. If the goals was not satisfied, the *Manager* tries to revert the current action. If the action is *reverted*, the cycle returns with *success*.
   c. Otherwise, if the action could *not* be reverted, then the cycle returns with *error*.
2. If the verification was started because the environment step, including all its entities steps, has finished, the three execution flows can be executed:
   a. If the goal is satisfied, the cycle returns with *success* and the step is *accepted*.
   b. If the goals was not satisfied, the *Manager* tries to revert the current step, including all entities steps. If the step(s) is(are) *reverted*, the cycle returns with *success*.
   c. Otherwise, if the step(s) could *not* be reverted, then the cycle returns with *error*.

4.3 A Framework that Implements SSOA

To demonstrate the feasibility of the SSOA, an object-oriented framework in Java that implements the SSOA components was developed: in [13], [7] one can find the implementation of the architecture's core. The framework shows a concrete design of the architecture and supports the development of simulation-based self-organizing multi-agent systems that help with the engineering of self-organizing systems design.

The MESOF component was built on top of MASON [15] that offers many interesting resources for simulating multi-agent systems in a discrete and event-based manner as two- and three-dimensional visualizations, charts and reports construction, video recording and much more. The entities being scheduled can be both executed in all modes provided by MASON library, i.e., sequential types and parallel sequence.

If we take a look in the figure 5, the *Simulation, Schedule, Steppable* and *Stoppable* classes were replaced by the corresponding MASON classes and we specialized the *Simulation* class (which is called *SimState* in MASON) with more functionalities specified in SSOA description. Also *Network* and *Grid2D* classes were implemented through the existent classes in MASON. On the other hand the *ContinousGrid* class had to be created. All other entities did not exist previously in MASON.

Developing the framework was a valuable experience. It is worth saying that several versions of the case studies were developed in order to reach the final architecture here proposed and that enable the framework development [16]. Furthermore, the framework development has improved our general understanding of important aspects of self-organizing systems such as the state of the environment, multi-environment hierarchy, the coordination and information flows design and the application of a middle-out approach that relate the micro and macro level to validate and sometimes to speed the design solution development. We also learned that deriving a concrete design from the architecture is not self-evident, in particular because there are different environment structures, and it requires a lot of effort and expertise of the designer.

4.4 Implementation Blueprints

In this section we present some implementation blueprints to be considered when implementing and instantiating SSOA.

4.4.1 How to initialize the simulation

First the *Simulator* controller must instantiate the *Environment* during initialization. This will represent the main environment. Then other entities can be initialized then after this when needed. Still during the simulation initialization, if the Manager Component will be activated, the following code should be added to initialize this component and activate it:

```
Simulation class
void INIT()
...
Manager manager := Manager.getInstance()
manager.addListener(this)
manager.init(goals)

void START()
...
Manager.getInstance().start(this)
```

This ensures that: there are only one instance of the *Manager* in the simulation (Singleton pattern), that the simulation is a listener of the Manager (in order to gather

statistics about number of actions or steps reverted or accepted, and the goals were passed to the *Manager* initialization. Also all the external actions have to be created during initialization.

When instantiating the *start()* method of the *Environment* class , if overridden, must be called by the *start()* method implemented by the subclass. It starts the existent entities of the environment.

4.4.2 How the Manager knows the action to revert

If one action needs to be reverted, the Manager must have a reference to this class. Therefore, each time that an *Action* is instantiated and executed, it has to be added to the *Agent* actions list.

```
1  MyAction action := new MyAction(source, target)
2  actions.push(action)
3  action.execute(simulation)
```

As an observation, when creating an *Action* be sure that all the previous attributes state are recorded and passed to the *ReverseAction* class during its instantiation, so it is able to rollback the state of all changed attributes.

5 The Case Study

To evaluate the feasibility of the Simulation-based Self-Organizing Architecture, we have developed some prototype systems that instantiated the framework that implements the SSOA. In this paper we present the results achieved to the AGV problem. The objective with this evaluation is to show the use of the middle-out approach with the validation method going from the specification to the experimental results.

The case study is structured as follows: (a) the design and architecture instantiation, including internal and external actions description; (b) validation instantiation, including state variables, goals and state evaluation; and (c) experimental results. The experimental results are described with the same methodology as follow: first we describe the scenarios, then we describe the scenarios that were not validated by the SSOA Manager and the scenarios that were validated, and a comparison of some of them w.r.t the desired guarantees. Finally, we finish the evaluation presenting the results regarding the Manager overhead during the simulation execution.

5.1 The Automated Guided Vehicles

There are three main entities in the process of the AGV transportation system:

1. Pickup station: represents the beginning of the transportation task. Pickup stations receive new loads to be dispatched and transport requests are sent to the vehicles. For simplification, all the loads have the same priority. This means that any available vehicle will transport the load at the nearest pickup station.
2. Drop station: represents the end of the transportation task. Drop stations are always waiting for new loads routed by the vehicles. For simplification, all the drop stations can receive any load.

3. Intermediate station: represents the factory internal stations were vehicles move throughout.
4. Vehicle: an automated vehicle must to be able to receive a request for routing, to move in the environment, enter/ leave the pickup and drop station, pickup a load, drop off a load, and route a load.

5.1.1 The Design and Architecture Instantiation

In the AGV transportation system the main environment is represented by the *Warehouse* class, which is a situated 2D environment and manages the factory layout where the stations are (see figure 8). The stations are non-situated sub-environments of the *Warehouse*. Hence, they have a location on it according to the specified layout. The vehicles are agents represented by the *AGV* class and they move from one location to another in the *Warehouse*. The gradients are events that the agents (vehicles) perceive while they move, and are represented by the *Gradient* class, which extends the *Event* class from MESOF. It was necessary to define some fields in this class as magnitude and angle, so the vehicles could reason about them. Moreover, a *FailableState* class was defined, because any vehicle or station could be in a failure state. Both the *PickupStation* and *DropStation* have a device for exchanging loads. Because of this the *LoadStation* class was created.

In [14] and [9], the authors proposed to use gradient fields to help on the decentralized control. In physics, vector fields are often used to model the strength and direction of some force, such as the magnetic or gravitational force, as it changes from point to point. The association of these vectors in each space position allows the identification of the source coordinate direction. Therefore, the emission of gradient fields can be used to enable the dispatching and routing with decentralized control. Each gradient is emitted by a different entity and with different proposals as follows:

1. Pickup Gradient (load_gradient) is fired by the Pickup Station in order to notify that there are loads to be dispatched. This gradient is propagated by the Intermediate Stations and, as far from the Pickup Station, the weaker the gradient is. Therefore, each vehicle have to calculate the resultant gradient when handling the gradient.
2. Drop Gradient (drop_gradient) has the same essence of the pickup gradient, however, they are fired by the Drop Stations in each time step. The function of this gradient is to route loaded vehicles to the correspondent Drop Station in order to drop off the load.
3. Vehicle Gradient (agv_gradient) is a gradient that avoids collision between vehicles. Their magnitude is negative hence repeal other vehicles while they are moving throughout the warehouse.

Internal and external actions

All the vehicles and stations share two external actions represented by the *Fail* class and *Recover* class (reverse action of the *Fail* action). A probability rate was defined as an input parameter to start the *Fail* action. Another probability rate was defined as an input parameter to start the *Recover* action.

The *PickupStation* class has as external action the creation of a load (*CreateLoad*) executed by a human that add new loads to be transported by the vehicles. And this load is removed from this station by a vehicle through the execution of the *ReceiveLoad*. Besides this if a *PickupStation* is active, which means that is not in a failure state and

has loads to be dispatched, it is always firing *load_gradient* event. The *DropStation* is the one on which the enqueue of loads is an internal action (*DeliverLoad*) executed by a vehicle. Besides this if a *DropStation* is active, which means that is not in a failure state, it is always firing *drop_gradient* event. Beyond the *ReceiveLoad* and *DeliverLoad* actions, the vehicle (AGV class) has the *Move* action and its reverse action. All the actions have their corresponding reversing actions.

5.1.2 Validation Instantiation

State Variables

All the vehicles and stations share the failure state, that can be either healthy or with failure. The stations have another state variable: the number of loads. The vehicles have two more state variables: the location and loading state. The latter reports if the vehicle is carrying a load or not. There are three macroscopic state variables to be monitored: (i) the total number of loads waiting to be transported at the pickup station; (ii) the total number of loads transported by the vehicles; and (iii) the time span for transporting each of them.

Goals and State Evaluation

To achieve the two macroscopic properties defined: routing and dispatching, we need more than guarantees about delivering all packets any time, we want it to be as fast as possible. Therefore, we need statements that assure the desired evolution of the average, and those statements represent the macro properties. While figure 9 illustrates the SSOA instantiation, the statements are presented as a description of the instantion.

1. *The system is guaranteed to have a monotonically increasing loads dispatching throughput.*
   The dispatching throughput measures how long a load waits to be dispatched at a pickup station and it can be calculated as follows. Let the QS_i be the queue size of a pickup station i, k the number of pickup stations, the dispatching throughput DT at a given time step x is given by:

$$DT(x) = \sum_{i=1}^{k} (QS_i(x) - QS_i(x-1)) \tag{1}$$

   If $DT(x) > 0$ it means there are more loads coming than being dispatched. If $DT(x) < 0$, there are more loads being dispatching than coming. Because we want the later, we need to minimize the normalized measure of DT regarding a trial of N simulations, this gives:

$$\min DTN(x) = \sum_{i=1}^{N} \frac{DT_i(x)}{N} \tag{2}$$

   This goal and state evaluation instantiation are represented by the *MaxDispatchingThroughput* class and have the following algorithm :

```
MaxDispatchingThroughput class
boolean isSatisfied(s)
input Simulation s
```

```
1  warehouse := s.getEnvironment()
2  prevDT := ( s.prevDT == null ? 0 : s.prevDT)
3  for each pickupStation in warehouse
4    DT := DT + pickupStation.getLoadCount() -
               pickupStation.getPrevLoadCount()
5  if DT-prevDT > 0
6    nbrOfPositiveTimes := nbrOfPositiveTimes + 1
7    if nbrOfPositiveTimes > threshold
8        return false
9  s.prevDT := DT
10 return true
```

Where the *threshold* is a parameter that says how many times it is allowed to have an increasing behavior. If never is desired, it should be initialized with 0.

2. *The system is guaranteed to have a monotonically increasing loads routing throughput.*
The routing throughput measures the routing rate of a load from the moment a vehicle gets a load and delivers it at the drop station. It is calculated based on the aggregation of the number of loads at the drop stations divided by their corresponding routing time. Let RT be the routing throughput at time step x, the QS_i be the queue size (number of loads) at the drop station i, and Δt_j the time a vehicle took to carry the load j:

$$QS = \sum_{i=1}^{w} QS_i(x) \tag{3}$$

Where QS is the total number of loads of all drop stations,

$$\Delta t_A = \frac{\sum_{j=1}^{w} \Delta t_j}{QS} \tag{4}$$

Where Δt_A is the average time span for routing a load, and we want to maximize this:

$$RT(x) = \frac{QS}{\Delta t_A} \tag{5}$$

We need to maximize the normalized measure of RT regarding a trial of N simulations, this gives:

$$\max RTN(x) = \sum_{i=1}^{N} \frac{RT_i(x)}{N} \tag{6}$$

This goal and state evaluation instantiation are represented by the *MaxRoutingThroughput* class and have the following algorithm :

```
MaxRoutingThroughput class
boolean isSatisfied(s)
input Simulation s
1  warehouse := s.getEnvironment()
2  prevRT := ( s.prevDT == null ? 0 : s.prevDT)
3  for each dropStation in warehouse
4     QS := QS + dropStation.loadQueue.size
```

```
5     for each load in dropStation.loadQueue
6       T := T + (load.deliverTime() - load.dispatchTime())
7   if QS>0
8    TA := T / QS
9    RT = QS / TA
10  if RT-prevRT < 0
11    nbrOfNegativeTimes := nbrOfNegativeTimes + 1
12    if nbrOfNegativeTimes > threshold
13      return false
14 s.prevRT := RT
15 return true
```

Where the *threshold*, like the previous goal, is a parameter that says how many times it is allowed to have a decreasing behavior. If never, it should be initialized with 0.

5.1.3 Experimental Results

For this problem it is possible to set up 250 (two hundred and fifty) different scenarios: one for each combination of 1 to 5 pickup stations, 1 to 5 drop stations, and 1 to 10 vehicles. We ran 30 simulation executions of 1,000 steps. And we collected and normalized the state variables and desired guarantees for the state evaluation. To describe the results, we chose 3 scenarios we found interest and sufficient to analyze: 5 pickup and drop stations, with 5 vehicles; 5 pickup and drop stations, with 8 vehicles; and 7 pickup and drop stations, with 3 vehicles.

In figure 10 we show the routing throughput that measures the routing rate of a load from the moment a vehicle gets a load and delivers it at the drop station. The best scenario was with 8 vehicles and 5 stations of each type. The results show how the scenario is set is crucial to the model and how the variables behave w.r.t the gradient-based self-organizing mechanism. Regarding the dispatching throughput (Figure 11), which measures how long a load waits to be dispatched at a pickup station, the results showed us that the best scenario was also with 8 vehicles and 5 stations (negative values).It is important to notice that we only plotted the behavior of the other two scenarios (5 vehicles and 5 stations, and 7 vehicles and 3 stations) because we wanted to compare the results. To do this, we had to turn off the Manager, because during the normal process, the Manager reverted several steps for those scenarios, until finding the scenario where the goals were satisfied. Those two scenarios hadn't the goals satisfied.

5.2 Manager Overhead

As a final evaluation, we present in this section the manager overhead analyzes in the simulation process. Figure 12 shows the comparison of the simulation performance considering the Manager on and the Manager off. The rate was defined as how much time a simulation step lasts in seconds. Thus the lower the better since it means we have a fast simulation. We computed the metric for the AGV case study. If the Manager is off, the step lasts in average 20 seconds. While when the Manager is on, the step lasts in average 120 seconds. As it is shown, the Manager's performance must be improved, since the difference was on average six times higher. Of course that once we have the

distributed parallel architecture, this wouldn't be a problem. But, for the simulation to run in a single computer, one may care about these results.

6 Related Work

While there is plenty of literature about the analysis of self-organizing and emergent mechanisms, the interest in engineering aspects grew only recently.

In this section we only summarize those works about self-organizing systems that are strictly related to the design of self-organizing systems, architectures, and validation or verification methods applied to those systems.

Parunak [17] describes several optimization algorithms for which an environment is needed but does not provide an architecture or middleware.

Omicini proposed TuCson [18], a middleware that allows the coordination of parallel processes (agents) through tuplespaces which can be seen as early (and ongoing) work to provide an environment wherein agents can interact. Mamei et al. [10] provide a middleware environment, called TOTA, which allows agents to coordinate their movements in a mobile network. Both TOTA and TuCson do not provide architectural simulation, multi-environment and validation features. And, finally, Weyns [8] proposed a reference architecture for situated multi-agent systems that includes a set of self-organizing features and considers the environment as a first-class abstraction. The reference architecture provides a set of mechanisms for architectural design, including: environment infrastructure for perception, action, and communication; laws that constrain the activity of agents; dynamics in the environment [19], [20], [21]; virtual environment [22]; selective perception [23]; advanced action selection mechanisms with roles and situated commitments [24], [25]; and protocol-based communication [26]. However, the focus of this architecture is to support the design from the agent-environment models to the **deployment level**. It was not designed to support discrete simulations, and it does not exploit different environment structures, or validation support.

Luca Gardelli [6] proposed a meta-model and a methodological approach for engineering self-organizing multi-agent systems. The meta-model is based on stigmergy, i.e, indirect communication where individual parts communicate with one another only by modifying their local environment. Artifacts are first-class entities representing the environment which mediates agent interaction and enables emergent coordination: as such, they encapsulate and enact the stigmergic mechanisms (diffusion, aggregation, selection, etc.) and the shared knowledge upon which emergent coordination processes are based.

Gardelli also proposed a simulation approach. The models are analyzed using stochastic simulations (stochastic Pi-calculus [27], [28] and the Stochastic Pi-Machine (SPiM) [29], with the goal to describe the desired agent behavior and a set of working parameters. These are calibrated through a tuning process. Gardelli partially formally modeled three self-organizing applications and analyzed the system-wide behavior that address qualitative simulation. He integrated a formal model approach with configuration or parameter tuning to accomplish the verification of the macroscopic behavior of self-organizing multi-agent systems and thus proving their correctness.

De Wolf [9] has proposed agent-based simulations combined with numerical analysis algorithms for dynamical systems verification at macro-level. By using a quantitative validation method, De Wolf applied the equation-free approach, first proposed by Kevrekidis [31], as the verification technique. In scientific computing research, there

exists a whole store of numerical analysis algorithms that support the analysis of the system dynamics and which have a mathematical foundation. Typically, these are applied to formal equation-based models. The "Equation-Free Macroscopic Analysis" approach supports the empirical application of these analysis algorithms without needing a formal equation-based model. In fact, the evolutionary equations are replaced by small simulations of the system evolution, considering some input parameters. This technique results in more valuable and advanced verification results and are supported by dynamical systems theory.

De Wolf did not provide an architecture for this approach neither support for the simulation, coordination and multi-environment features. However, De Wolf's validation method is complementary to the validation method proposed in this work. The scientific numerical analysis algorithms could be encapsulated in the Manager in a way that they could be easily executed and re-initialized to different application domains.

7 Conclusions and Future Work

Self-organization and emergence are important aspects of decentralized distributed systems. Through the applicability of self-organizing mechanisms, distributed systems can have decentralized control and increase in robustness. Although self-organization is an old biology concept, it is not a mature computer science concept, and the community that investigates the particularities of its usage in computer systems is still quite small, and mostly associated to the Distributed Systems area.

We believe the widespread of building self-organizing emergent systems depends on software engineering techniques and this was the focus of the research presented in this paper. This research represents a step toward the advances on architectural design of self-organizing emergent systems. The contributions of this work are twofold:

1. We applied an agent-oriented technology to build self-organizing emergent systems. This technology provides a Simulation-based Self-Organizing Architecture (SSOA) and best practices as reuse and modularity. This architecture provides an asset base engineers can draw from when developing self-organizing applications. The application of the engineering guidelines and simulation-based self-organizing architecture in a well known self-organizing problem is a contribution. Finally, none of the architectures or middleware presented in the related work section were designed to support discrete simulations and they do not exploit different environment structures. Therefore, we can clearly observe the need and contribution of an integrated architecture which encompasses both simulations, environment structures and dynamics, coordination components, self-organizing and validation mechanisms support as it was proposed in this work.
2. In order to use an agent-oriented technology, validation architectural features for validating emergent behavior of multi-agent systems was proposed. Regardless self-organizing mechanisms, agents in multi-agent systems need to be coordinated in some way, and those customizations help on this task. Also, emergent behavior is an inherent multi-agent system characteristic, and the validation method helps on the monitoring and controlling of agents misbehavior.

7.1 Future Directions

This work has uncovered some problems to be solved, which are listed below. Some of them are current ongoing work.

The main directions in which the SSOA architecture can be evolved is twofold: regarding the self-organizing patterns, and the Manager component w.r.t the validation method. A catalog of self-organizing patterns can be encapsulated in SSOA architecture allowing the software engineer to easily instantiate them in specific problems.

Regarding the Manager component, a set of debug improvements can be encapsulated in a way that the Manager could provide causality relations between local actions that could not be accepted with regard to the specific macroscopic properties.

Also, the entropy concept can be introduced. From [30], [32], [33], (spatial) entropy is suitable to reflect the spatial distribution of entities between different states and it is defined as:

$$E = \frac{-\sum_{i=1}^{N}(p_i \times \log p_i)}{\log N} \tag{7}$$

Where p_i is the probability that state i occurs and $\sum_{i=1}^{N} p_i = 1$. Dividing by $\log N$ normalizes E to be between 0 and 1. Entropy is high (close to 1) when the considered states have an equal probability to occur, and low (close to 0) when only a few of the states have a high probability to occur. De Wolf applied this measure to the distribution of AGVs: the different states for the entropy measure are defined as the desired situations for the AGVs. And considering the AGVs are already distributed between the desired situations. The probability for such an AGV to be in one specific desired situation at one moment in time is used as the probability in the entropy equation.

Therefore, if the SSOA provides specific interfaces for instantiating this operator, the Manager could be able to evaluate the system w.r.t the entropy and re-initialize the simulation adaptively. In addition, the scientific numerical analysis algorithms used by De Wolf (and other related ones) could also be encapsulated in the Manager in a way that they could be easily executed to different application domains.

Another future direction is to provide a transparent distributed parallel simulation middleware. In 2D or 3D situated environments, regardless of the visualization process, one requirement is the space management. In a sequential simulation this does not incur in a problem because each request is treated in the order of arrival. However, in a distributed parallel environment requests may arrive at any time, so it is necessary to provide a solution for concurrency.

To achieve this goal, we have being investigating on how to provide the capabilities for the proposed architecture to become a distributed and parallel solution, in the sense that it could work on a cluster environment in order to achieve better processing times or larger problem sizes. The goal for this new architecture is to provide the user with: the solution in less time, or; a solution to a larger instance of the problem in the same time frame.

References

1. A. Visser, G. Pavlin, S.P. van Gosliga, M. Maris. "Self-organization of multi-agent systems", Proc. of the International workshop Military Applications of Agent Technology in ICT and Robotics, The Hague, the Netherlands, 23–24 November 2004.

2. G. Di Marzo Serugendo, M.–P. Gleizes, A. Karageorgos; Self-organization in multi-agent systems. The Knowledge Engineering Review, Vol. 20:2, 165:189, 2005, Cambridge University Press.
3. M. Wooldridge, N.R. Jennings. Intelligent Agents: Theory and Practice. Knowledge Engineering Review 10(2), 1995, pp. 115-152.
4. N. R. Jennings. On Agent-Based Software Engineering. Artificial Intelligence Journal, 117 (2) 277-296, 2000.
5. T. De Wolf, and T. Holvoet, Design Patterns for Decentralised Coordination in Self-Organising Emergent Systems, Editors: Sven Brueckner, Salima Hassas, Mrk Jelasity and Daniel Yamins, Engineering Self-Organising Systems: Fourth Int. Workshop 2006, LNCS, vol 4335, 2007, Springer Verlag.
6. Gardelli, L., Viroli, M., Casadei, M., and Omicini, A. 2008. Designing self-organising environments with agents and artefacts: a simulation-driven approach. Int. J. Agent-Oriented Software Engineering. 2, 2 (Feb. 2008), 171–195.
7. Gatti, M. A. C. ; Lucena, C.J.P. de; A Multi-environment Multi-agent Simulation Framework for Self-organizing Systems; G. Di Tosto and H. Van Dyke Parunak (Eds.): MABS 2009, LNAI 5683, pp. 61-72, 2010. Springer-Verlag Berlin Heidelberg 2010.
8. D. Weyns. An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems, PhD Thesis, 2006.
9. DeWolf, T.; Analysing and engineering self-organising emergent applications, Ph.D. Thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May, 2007, 183 pags.
10. Mamei, M., Zambonelli, F.: Self-maintained distributed tuples for field-based coordination in dynamic networks. In: The 19th Symposium on Applied Computing (SAC 04), 2004.
11. Pressman, R.S. Software Engeneering: A practitioner's approach. Mc-GRAW-HILL, 2001.
12. G. Fishman; Discrete-Event Simulation: Modeling, Programming, and Analysis, 2001, pp. 26-27.
13. Gatti, M. A. C. ; Lucena, C.J.P. de . A Multi-Environment Multi-Agent Simulation Framework for Self-Organizing Systems, 2009, Budapest. The Eighth International Conference on Autonomous Agents and Multiagent Systems, 2009.
14. Weyns, D., Omicini, A. Odell, J.; Environment, First-Order Abstraction in Multiagent Systems. Autonomous Agents and Multi-Agent Systems, v.14 n.1, p.5–30, February 2007.
15. S Luke, C Cioffi-Revilla, L Panait, and K Sullivan, "MASON A New Multi-Agent Simulation Toolkit", Department of Computer Science and Center for Social Complexity, In Proceedings of SwarmFest, Michigan, USA, 2004.
16. Gatti, M. A. de C.; Lucena, C. J. P. de. Engineering Self-Organizing Emergent Multi-Agent Systems: A Design Method and Architecture. Rio de Janeiro, 2009. 153p. DSc Thesis - Departamento de Informatica, Pontificia Universidade Catlica do Rio de Janeiro.
17. Parunak, V.: "Go to the Ant": Engineering principles from natural multi-agent systems. Annals of Operations Research 75 (1997) 69-101.
18. Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R., eds.: Coordination of Internet Agents: Models, Technologies, and Applications. Springer (2001).
19. D. Weyns, G. Vizzari, and T. Holvoet. Environments for situated multiagent systems: Beyond Infrastructure. In Proceedings of the Second International Workshop on Environments for Multi-Agent Systems, Utrecht, 2005, Lecture Notes in Computer Science, Vol. 3380. Springer Verlag.
20. D. Weyns and T. Holvoet. On Environments in Multiagent Systems. AgentLink Newsletter, 16:1819, 2005.
21. D. Weyns and T. Holvoet. On the Role of the Environment in Multiagent Systems. Informatica, 29(4):408421, 2005.
22. D.Weyns, K. Schelfthout, and T. Holvoet. Exploiting a Virtual Environment in a Real-World Application. In Proceedings of the Second International Workshop on Environments for Multi-Agent Systems, Utrecht, Lecture Notes in Computer Science, Vol. 3830. Springer Verlag, 2005.
23. D. Weyns, E. Steegmans, and T. Holvoet. A Model for Active Perception in Situated Multiagent Systems. In 1st European Workshop on Multi-Agent Systems, Oxford, UK, 2003.
24. D. Weyns, E. Steegmans, and T. Holvoet. Integrating Free-Flow Architectures with Role Models Based on Statecharts. In Software Engineering for Multi-Agent Systems III, SELMAS, Lecture Notes in Computer Science, Vol. 3390. Springer, 2004.

25. E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers. A Design Process for Adaptive Behavior of Situated Agents. In Agent-Oriented Software Engineer- ing V, 5th International Workshop, AOSE, New York, NY, USA, Lecture Notes in Computer Science, Vol. 3382. Springer, 2004.
26. D. Weyns, E. Steegmans, and T. Holvoet. Protocol Based Communication for Situated Multi-Agent Systems. In 3th Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, USA, 2004. IEEE Computer Society.
27. R. Milner, J. Parrow, and David Walker. A calculus of mobile processes I. Information and Computation, 100(1):1–40, September 1992.
28. C. Priami. Stochastic pi-calculus. The Computer Journal, 38(7):578,589, 1995.
29. A. Phillips. The stochastic pi-machine (SPiM). Version 0.044 available online at http://research.microsoft.com/ aphillip/spim/, November 2007.
30. S. Guerin and D. Kunkle. Emergence of Constraint in Self-Organizing Systems. NDPLS: Nonlinear Dynamics, Psychology, and Life Sciences, vol. 8, no. 2, pages 131:146, 2004.
31. I. G. Kevrekidis, C. W. Gear and G. Hummer. Equation-free: The computer-aided analysis of complex multiscale systems. AIChE J., vol. 50, no. 7, pages 1346–1355, 2004.
32. H. V. D. Parunak and S. Brueckner. Entropy and self-organization in multi-agent systems. In AGENTS'01: Proceedings of the fifth international conference on Autonomous agents, pages 124:130, New York, NY, USA, 2001. ACM Press.
33. T. Balch. Hierarchic Social Entropy: An Information Theoretic Measure of Robot Group Diversity. Autonomous Robots, vol. 8, pages 209:237, 2000.

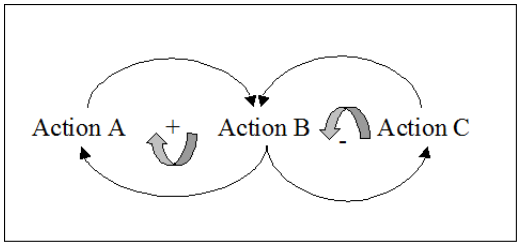

Fig. 1 Coordination Support for Feedback Loops

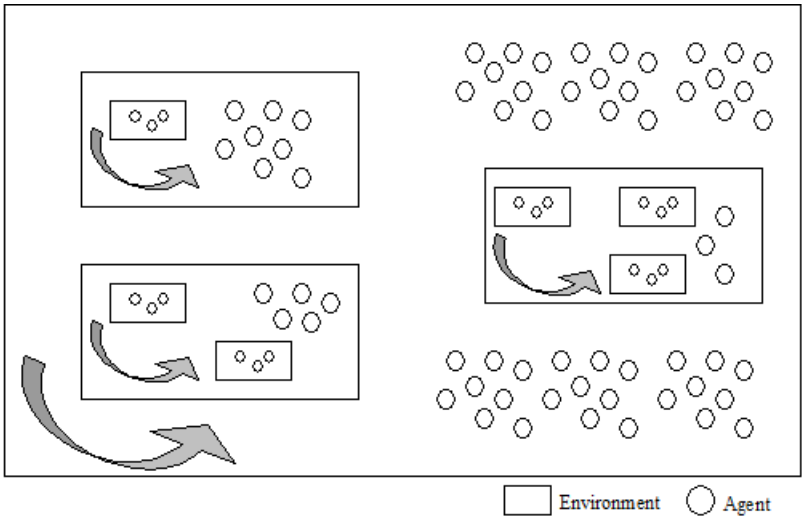

Fig. 2 The Multi-Environment Perspective

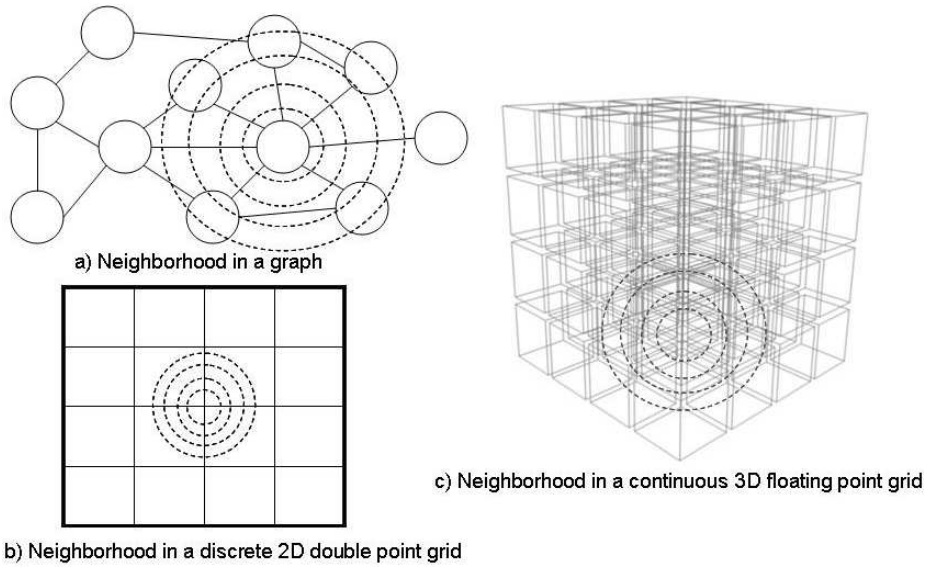

Fig. 3 The Multi-Environment Perspective: a)Graph: each agent or sub-environment can be located in a node and perceives its neighbors; b) 2D double point grid: each agent or sub-environment can be located in a discrete 2D double point position in the grid; c)3D continuous grid: each agent or sub-environment can be located in a 3D floating point grid.

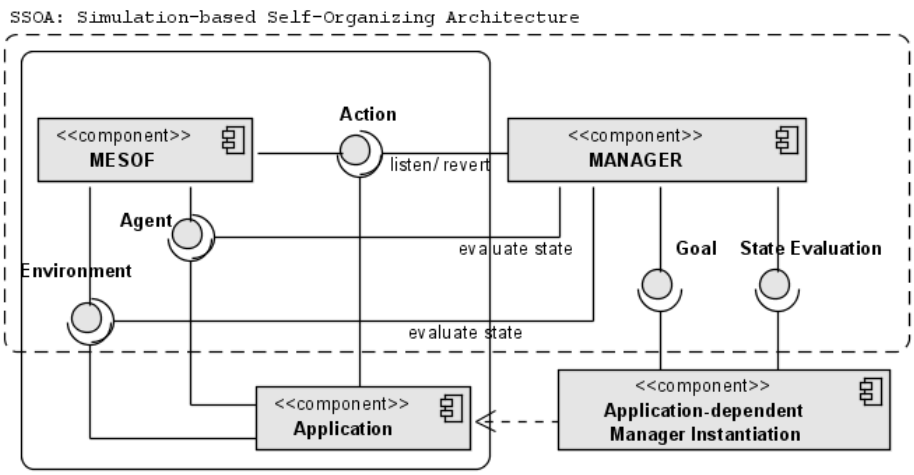

Fig. 4 The SSOA Components

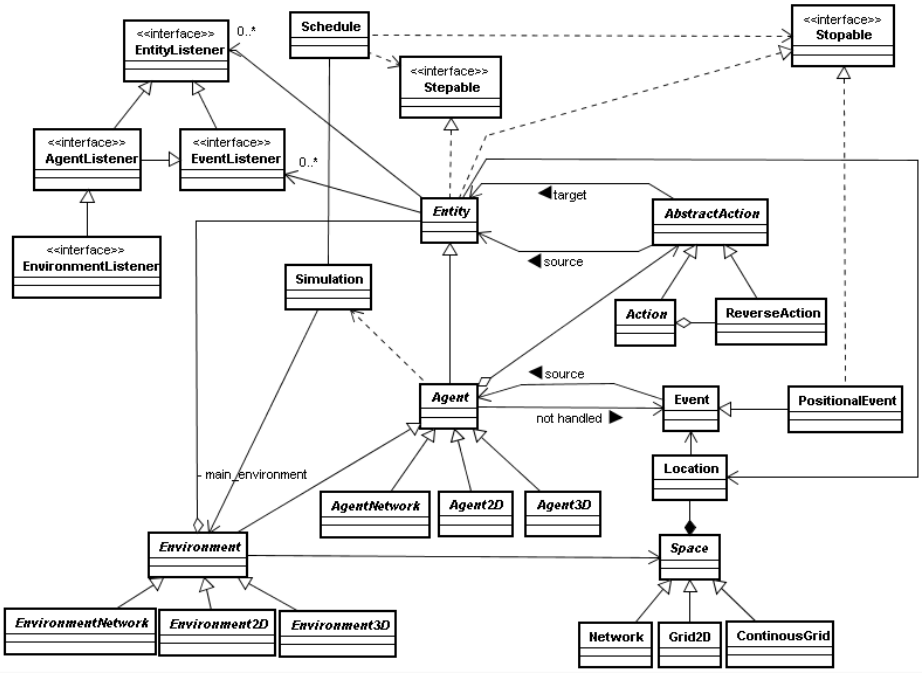

Fig. 5 The MESOF Meta-Model

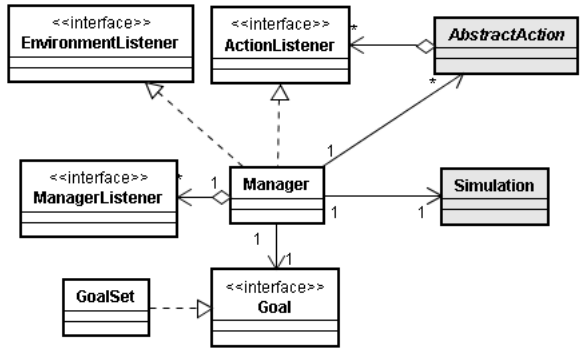

Fig. 6 The Manager Meta-Model

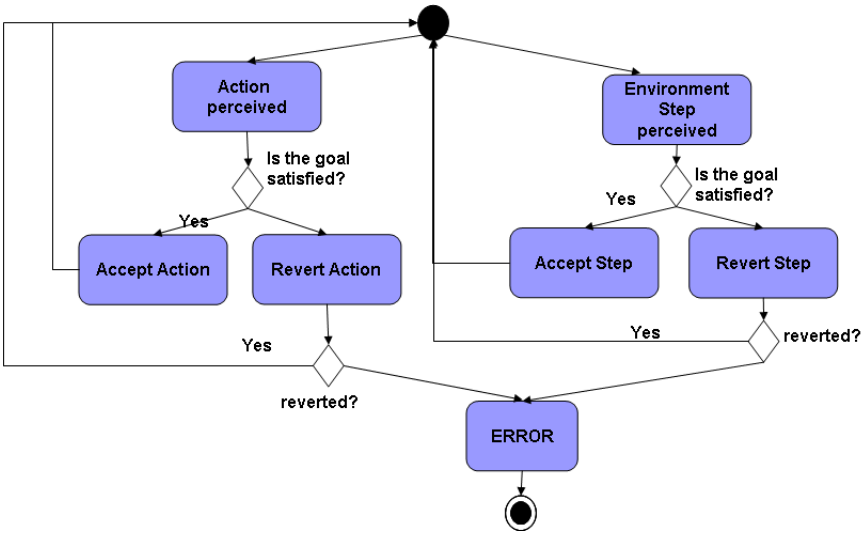

Fig. 7 The Manager Life Cycle

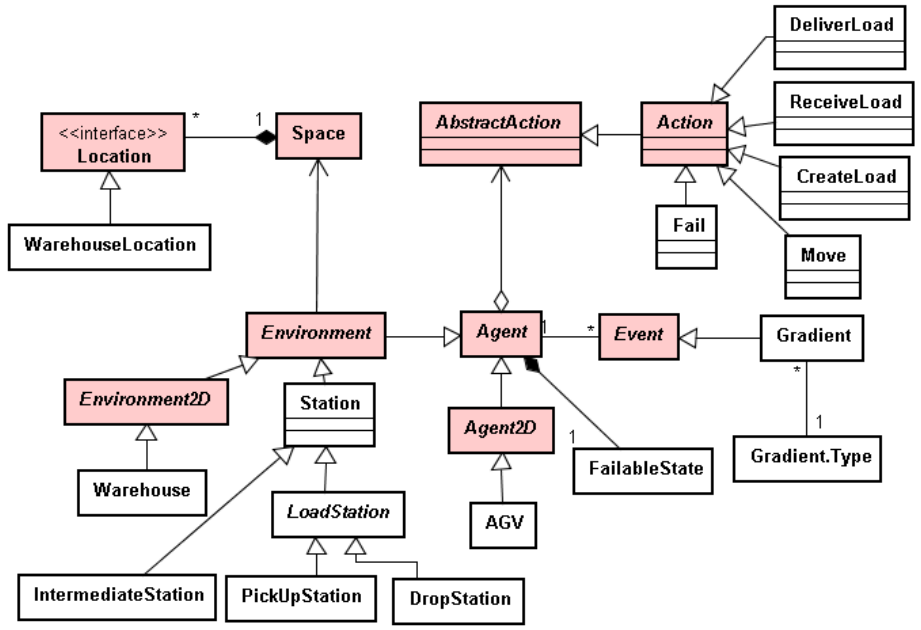

Fig. 8 The AGV Class Diagram (partial view)

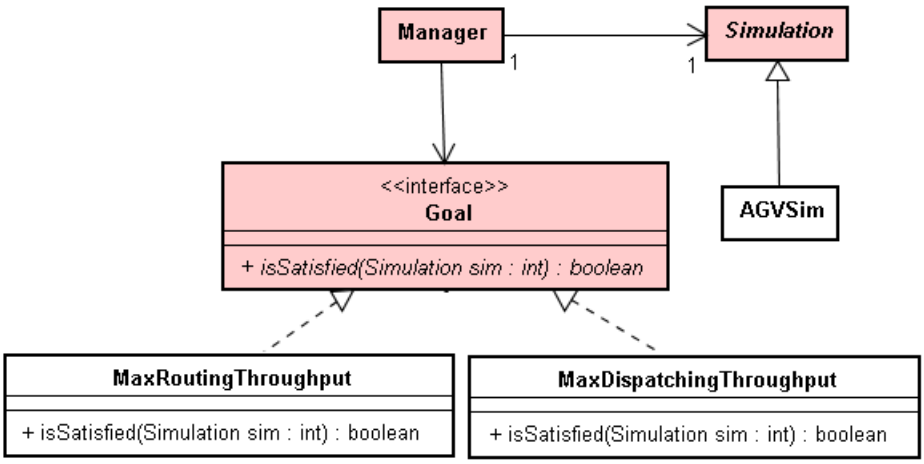

Fig. 9 AGV Goals and SSOA Manager Component instantiation

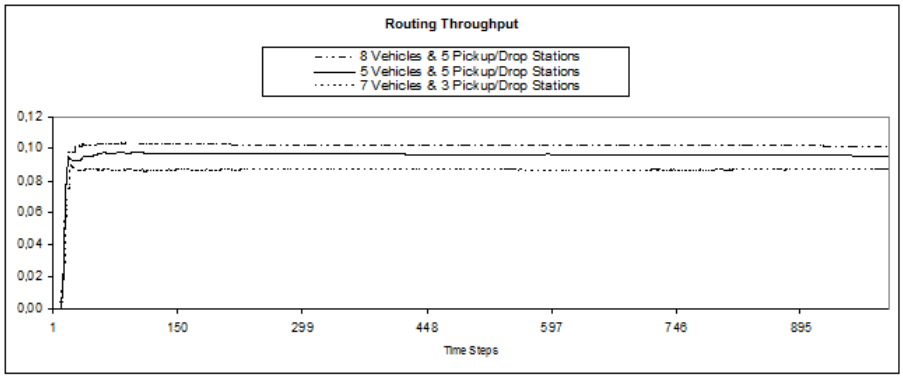

Fig. 10 Routing throughput analysis

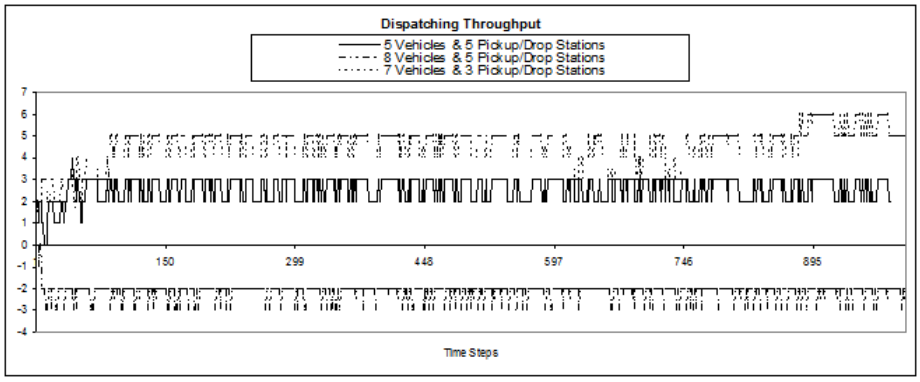

Fig. 11 Dispatching throughput analysis

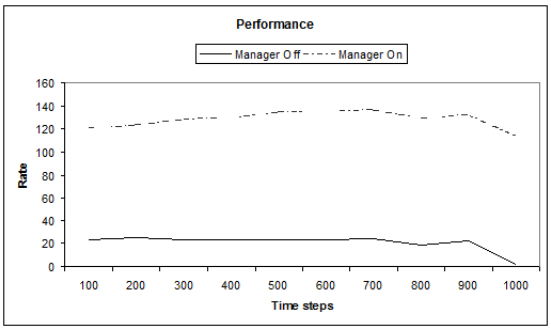

Fig. 12 Manager overhead in the simulation