4
SSOA: A Simulation-based Self-Organizing Architecture

An architectural design helps on the development of a modular program structure and on the representation of the control relationships between modules [Pressman 2001]. It provides a software engineer with a picture of the program structure and behavior. SSOA is a multi-environment-centric simulation-based architecture that encourages the software engineer to concentrate on architectural design before worrying about optimizations or code. The main goal of SSOA is to provide an architecture that helps on the design, simulation, and validation of self-organizing systems. At this point on this thesis, the reader should be aware of the basic requirements that a self-organizing architecture should fulfil. But in order to better understand the requirements and their motivation, we first describe the engineering method to be applied when using the architecture. The engineering method follows a simulation-based methodological approach adapted to include the architecture instantiation activity. Therefore, this is a very reuse-based engineering method. After presenting the engineering method and the architecture requirements, we describe the architecture, its meta-model, and dynamics, including how it addresses the requirements presented. A framework that implements SSOA was developed during the case studies development presented in the next chapter. Therefore, we briefly present this framework that implements SSOA. At the end we finally present some blueprints on how to go from the designed models using coordinated statecharts (presented in chapter 3) to the architecture instantiation giving the reader an integrated design overview.

4.1 SSOA Iterative Life-Cycle

The earliest two main self-organizing life-cycles proposed in the literature are due to De Wolf [De Wolf 2007] and Gardelli [Gardelli 2008]. They proposed a simulation-based life-cycle for engineering self-organizing systems, in particular with the goal of verifying (Gardelli) and validating (De Wolf) these systems. A simulation-based life-cycle is mainly based on three phases: modeling, simu-

lation, and tuning. It enables the software engineer to build an implementation model, execute it, and thereby gain an understanding on how the system as specified would behave if implemented. It is particularly important in an environment with high costs to enable operation, as the AGV problem or the Contracts that regulate cross-organizational business applications. Of course this approach can also be applied to biological systems simulation. We saw in chapter 2 that biological systems have the same properties that self-organizing systems, in fact decentralized systems are modeled with self-organizing mechanisms which are bio-inspired. Therefore, the simulation-based process can also be applied on the simulation of biological systems. To fully understand how it can be applied, please refer to the solution of the third case study – the stem cell behavior computational modeling – in next chapter.

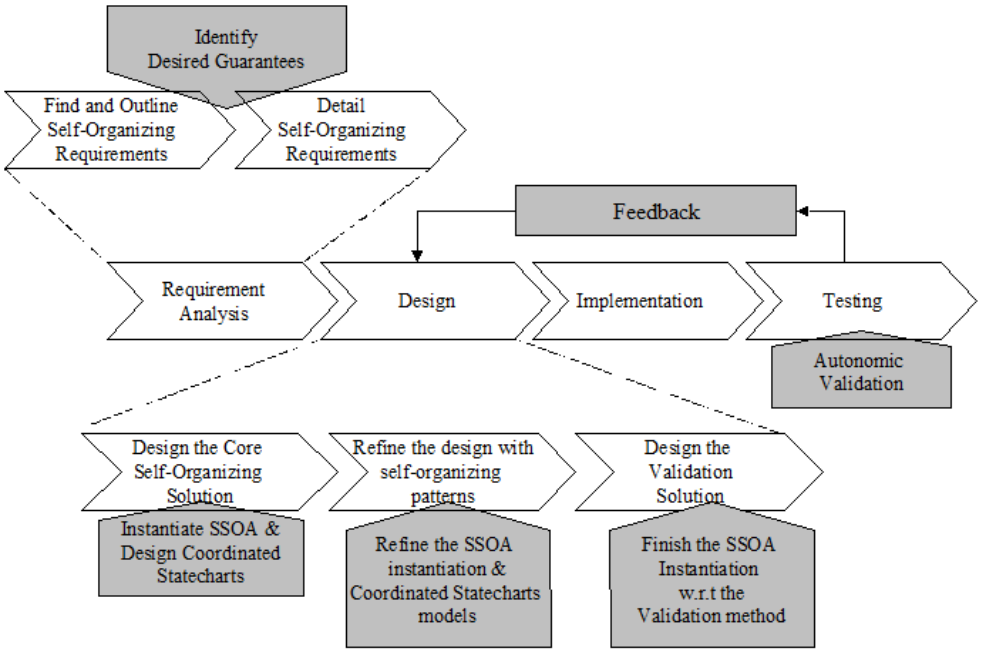

Figure 4.1: The Design Iterative Life-Cycle w.r.t the Unified Process

In this work, we have extended the simulation-based life-cycle with the addition of both the UML-based notational support – by using Coordination Statecharts, presented in chapter 3 – and the instantiation of the architecture here proposed. Because De Wolf founded his approach on the Unified Process, defining what he called The Customized Unified Process, we shall also *illustrate* the design approach proposed in this thesis based on the Unified Process for two reasons. First, it allows us to compare the difference between both approaches. And second, because it helps a software engineer to understand how the activities are related to a well known and accepted software process and its phases. This section has no intention to describe a full complete methodology.

Instead, it is a starting point from which a full life-cycle methodology can be based.

The Unified Process is an iterative and incremental development process. In particular, when considering a simulation-based approach, a solution is successively refined, with cyclic feedback from testing to design and adapting the solution accordingly to converge to a suitable solution.

De Wolf argues that one should focus on how to address the desired macroscopic properties in each discipline. We provide additional architectural details that helps with this respect.

Requirement Analysis phase

Based on the Unified Process, this phase includes three types of activities: eliciting, analyzing and recording requirements. They can be accomplished by communicating with the customers and users to determine what their requirements are. Also determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then resolving these issues, and documenting the requirements in various forms, such as natural-language documents, use cases, user stories, or process specifications, respectively. The problem is partitioned into functional and non-functional requirement.

De Wolf customized this discipline with the identification of macroscopic requirements (at the global level). To this end the software engineers have to know how to identify a macroscopic requirement in the context of a self-organizing system. De Wolf gave a hint on how to achieve this task by explaining that a self-organizing emergent MAS typically "maintains" certain macroscopic properties. Therefore, explicitly identifying those requirements that are "ongoing" and have to be adaptively maintained, is important. A specific question related to the macroscopic properties that have to be addressed and explicitly engineered is how to measure the system performance.

Although the customization proposed by De Wolf is of paramount importance, we further exploit it and split this discipline into:

- **Find and Outline Self-Organizing Requirements**
  Once the functional and non-functional requirement are gathered, an analysis has to be done to identify between those requirements, which are related to self-organization. More specifically, which of them are goals and desired guarantees that could not be reducible to local parts but drive the decentralized control and emergent

behavior? The purpose is to understand the problem considering the desired guarantees in the system and document this.

- **Detail Self-Organizing Requirements**
  The purpose of this activity is to describe one or more desired guarantees derived from the self-organizing requirements in sufficient detail to be able to validate the desired guarantees, to ensure compliance with stakeholder expectations and to permit software design to begin.

Design phase

The design phase is concerned with the problem-solving and planning for a software solution. The functional specification created describes what has to be implemented. The design describes how the functions will be realized using a chosen software environment. This phase is split into :

- **Design the Core Self-Organizing Solution**
  In this phase we specify and design the model of the self-organizing solution. To achieve this, two design methods are provided: Instantiate SSOA and Design Coordinated Statecharts.

  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. In order to identify and model the actions, we have proposed to apply the notational support described in chapter 3: Coordinated Statecharts.
- **Refine the design with self-organizing patterns**
  Once the core self-organizing solution has been modeled, we need to refine it with self-organizing patterns. A catalog of patterns can be used to identify the most suitable one (or more). After each pattern is chosen, we need to refine the models created using Coordinated Statecharts models.
- **Design the Validation Solution**
  The SSOA is a simulation-based life-cycle, therefore we need to

know *a priori* how the simulation of these models will be validated. This activity is related to the macroscopic properties that De Wolf introduced in the life-cycle and that we have been modeling as the desired guarantees. 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 an autonomic validation method that uses those reverse actions so the system can be run backward 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 SSOA w.r.t the validation method by modeling which subset of states of the system need to be matched. Then the autonomic validation module 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.

Implementation phase

The design then is expressed by using a specific language. De Wolf proposed the programmer to focus on the microscopic level of the system while implementing. In addition, we propose to also focus on the macroscopic level when the programmer implements the validation instantiation modeled in the last phase.

Testing phase

This 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 [Gardelli 2008]. 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 Autonomic Validation module retains the knowledge of all accepted and reverted actions. The Autonomic Validation module 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 Autonomic Validation module 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 Autonomic Validation module to change the input parameters for the external actions if a set of backward was performed. It can speed the simulation process even more in the case of testing several scenarios at once.

Therefore, our solution proposes a middle-out approach on the engineering of self-organizing multi-agent systems. While one can have a bottom up approach on the microscopic analysis of the self-organizing mechanisms, we also propose a top down approach achieved with the integration of the Autonomic Validation module.

4.2 The Architecture Requirements

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

(a) 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 simulation-based self-organizing architecture 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 [Fishman 2001].

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.

(b) Coordination Support

Among all the possible interaction mechanisms, a simulation-based self-organizing architecture has to support uncoupled and anonymous ones [Mamei 2004]. 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.

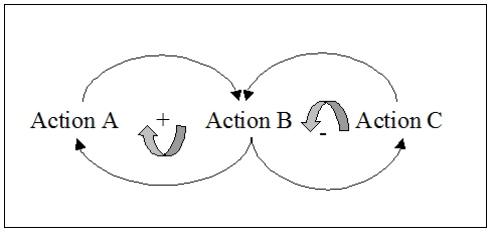

Figure 4.2: Coordination Support for Feedback Loops

Figure 4.2 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.

(c) 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.

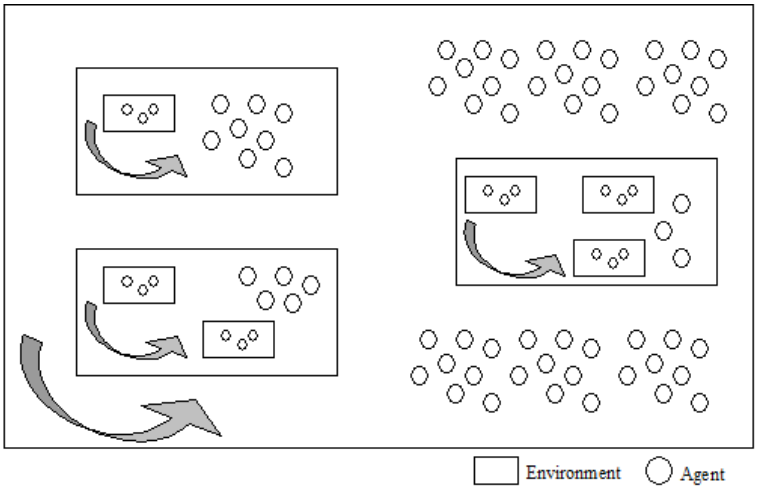

Figure 4.3: The Multi-Environment Perspective

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 [Weyns 2006, Gatti 2009].

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 4.4). 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

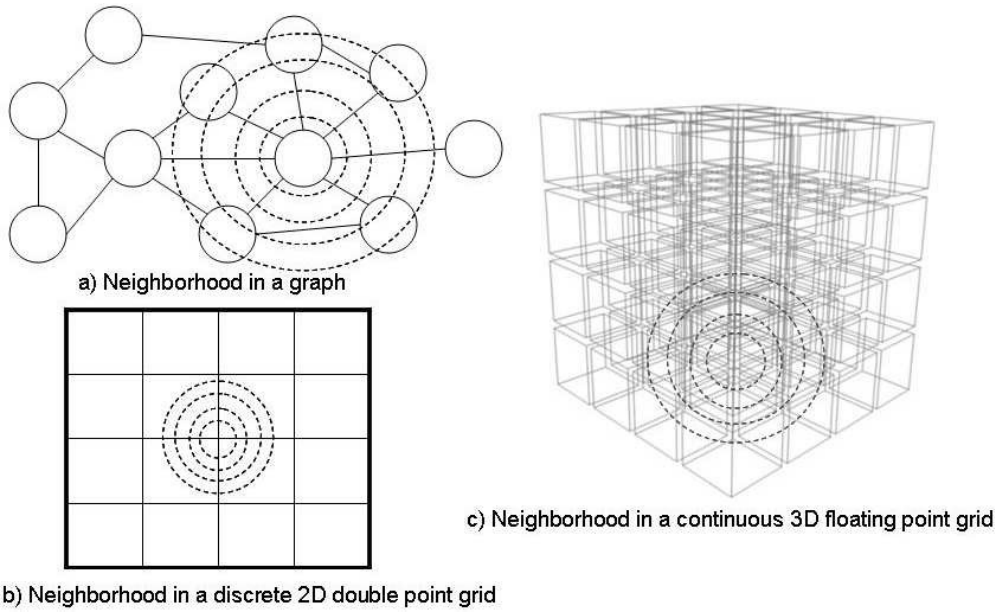

Figure 4.4: The Multi-Environment Perspective: a)Graph: each agent or sub-environment can be located in a node and perceives its neighbors; b) 2D double point grid: each agent or sub-environment can be located in a discrete 2D double point position in the grid; c)3D continuous grid: each agent or sub-environment can be located in a 3D floating point grid.

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 [Weyns et al. 2007], 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.

(d) Validation Support

The requirements for validation support in a simulation-based self-organizing architecture are related to the macroscopic properties design support and 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 have 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 execute backward 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 so backward 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.

4.3 The SSOA Description and Dynamics

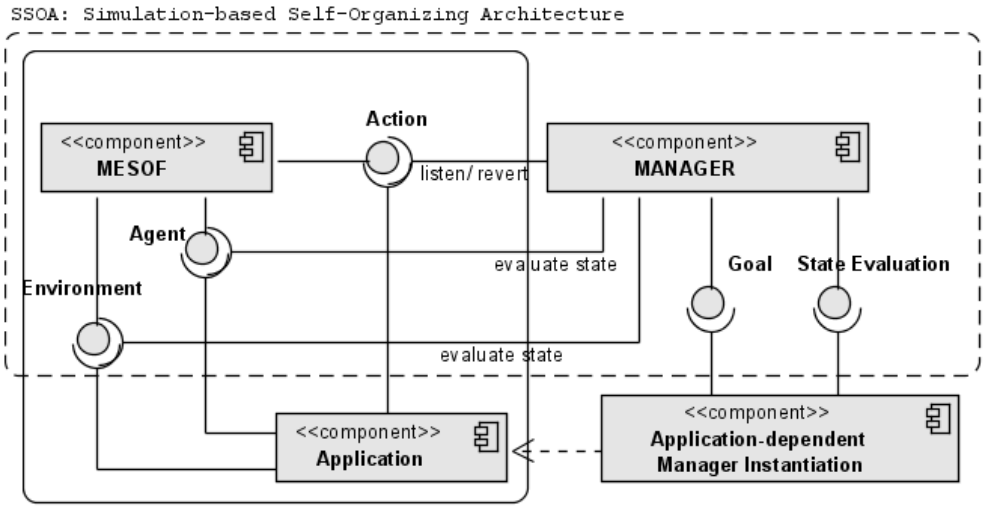

Figure 4.5: The SSOA Components

The two main components of the architecture can be seen inside the dashed box: MESOF and MANAGER.

(a) MESOF Component

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.

MESOF Meta-Model

The MESOF meta-model is described in this section. Figure 4.6 illustrates its structural and hierarchies and is explained through the features that correspond to the architecture requirements as follows:

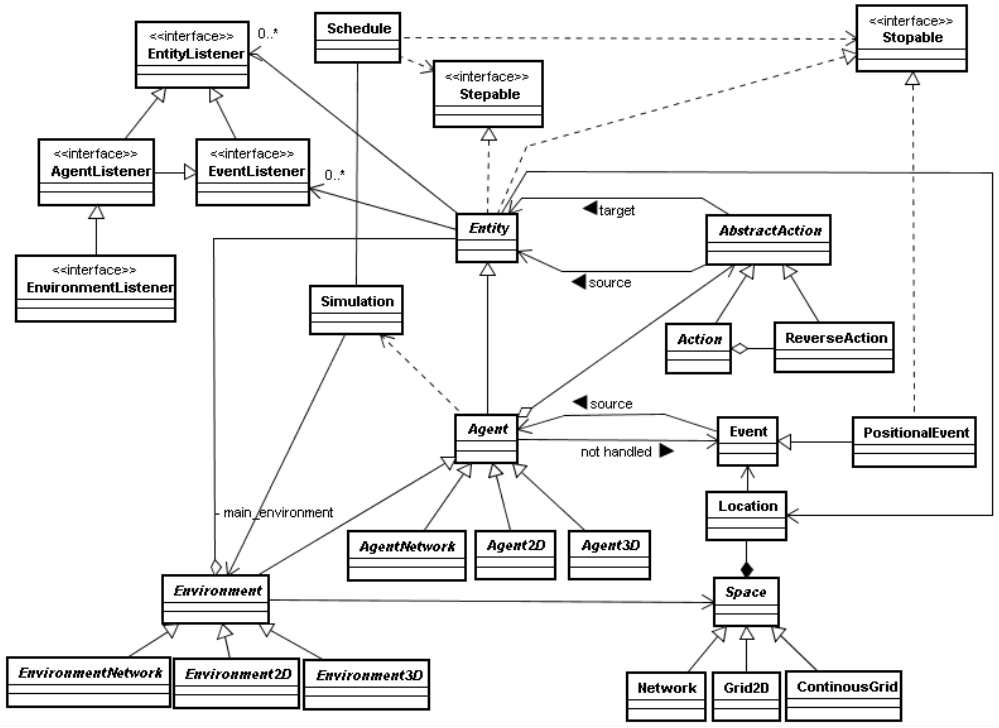

Figure 4.6: The MESOF Meta-Model

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.

To better explain these behavior from the coordination point of view, consider two agents A1 and A2 and their behavior considering the template method and hooks operations described:

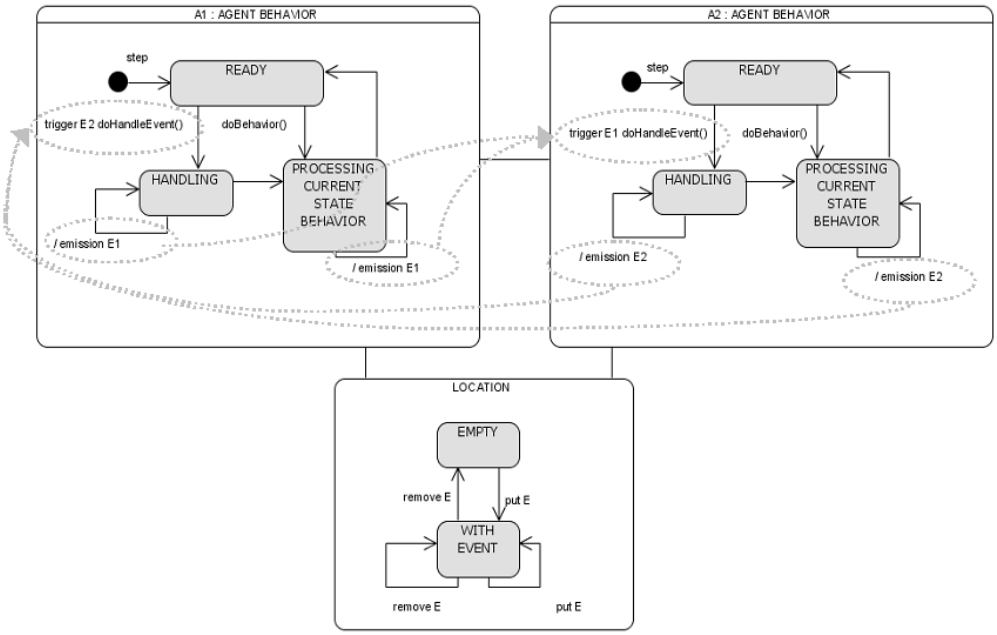

Figure 4.7: The coordination between two agents

Note that there are two properties at this coordinated statecharts model. First, the A1 Agent Behavior is exactly the same as the A2 Agent Behavior, despite the name of the event being fired. The name is different because they are different instances. This illustrates the reuse principle on each SSOA resides and how coordinated statecharts modularize behaviors to support reuse. Second the information flow between the two agents are represented by the gray dashed lines (which are illustrative, they are not part of the diagram). Note how the emission of an event (for instance, E1) reinforces the emission of other event (following the example, E2) which in turn emits more E1.

- **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 contain 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.

(b) Manager Component

The continuous style box shows that an application can be totally decoupled from the Manager Component, 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.

Manager Meta-Model

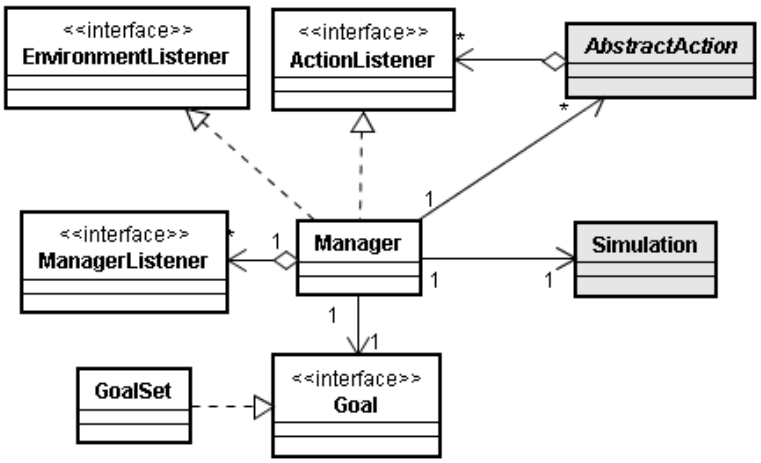

Figure 4.8: The Manager Meta-Model

Autonomic Validation Features

The *Goal* interface defines a desired guarantee. 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. 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*.

4.4 A Framework that Implements SSOA

To demonstrate the feasibility of the SSOA, we have developed an object-oriented framework in Java that implements the SSOA components [Gatti 2009]. 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 [Luke 2004] 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 4.6, 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 [Faustino et al. 2008] 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 presented in next chapter were developed in order to reach the final architecture here proposed and that enable the framework development. 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.

4.5 Implementation Blueprints

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

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.

How to implement coordinated statecharts into code

Transitions are fired per step according to pre-conditions and input events. The events are handled during the *doHandleEvent()* hook method. States can be defined as enumeration constants and Events can be typed using the Event.Type enum constant. Therefore, event-trigged actions are executed as a match of Event and State inside the *doHandleEvent()* method.

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.

4.6 Chapter Remarks

This chapter contains the main contribution of this thesis. Here we connected all the others contributions, as the notational support, with the description of the architecture proposed. We illustrated through the Unified Process how the SSOA life-cycle is realized. Then we presented the architecture requirements and the details of all entities, structures, components, and

dynamics of the SSOA. Some lessons learned were discussed during the development of the framework that implements SSOA and, finally, some implementation blueprints were presented for the software engineer who wants to build an executable simulation model that implements SSOA.