3
Notational Support

As it is the case with any new software engineering paradigm, the successful and widespread deployment of self-organizing systems require: notations that explore the use of self-organizing related abstractions and promote the traceability from the design models to code. Although information flows are essentials to support self-organization, an important matter is to design considering modularity and reuse. Due to lesser customization, and less learning time, reuse reduces cost, time, effort, and risk; and increase productivity, quality, performance, and interoperability. In addition, modularity should be easy to work with because modules can be easily understood in isolation, and changes or extensions to functionality would be easily localized. Hence we need modeling approach that combines information flows, state dynamics and software engineering principles as reuse and modularity. That said, this chapter presents UML 2 [OMG 2005] customizations that address these issues.

PUC-Rio - Certificação Digital Nº 0621318/CA

3.1 Information Flow Design

In a self-organizing multi-agent system an agent can execute several actions regarding its goals or perceptions. As well, the environment has the same features. The action behavior feature is executed during agent or environment execution without explicitly being called by other objects or agents. Agents interact with one another and the environment, sending and receiving messages or sending and receiving events through propagations in the environment.

In this context, in order to design the essential self-organizing feedback loops, the interplay between information flows AND actions AND states on control flows needs to be considered. De Wolf proposed to define Feedback Loop = Information Flows + Actions. Here we propose to extend this concept to Feedback Loop = Information Flows + Actions + *Behavioral States*. A behavioral state is a particular instance of the agent or environment in a scenario that represents a typical path through the state space within a single state machine, i.e., an ordered sequence of state transitions triggered

by events and accompanied by actions. To enable the design that addresses this new feedback loop concept, we propose to use Coordinated Statecharts, an extension of UML State Machine.

Next sections describe what is needed to exist in the UML 2 meta-model in order to support the customizations and more details about Coordinated Statecharts and its relation to existent statecharts concepts.

3.2 UML Customization with Coordinated Statecharts

The foundation of the design abstraction that enables the design of information flow with modularity and reuse principles relies on the customization of the UML 2 meta-model and behavioral statecharts, called Coordinated Statecharts [Gatti 2008a]. Our objective is not to compete with any of the existent efforts on modeling languages and methodologies for agent-oriented systems, but rather to extend and apply a widely accepted modeling and representational formalism (UML) in a way that makes it useful in communicating across the self-organizing research groups.

(a) Motivation

Before presenting the extensions it is important to understand why traditional statecharts are not enough and why/what we had to customize. First, a state machine diagram is a graph that represents a state machine. States and various other types of vertices (pseudostates) in the state machine graph are rendered by appropriate state and pseudostate symbols, while transitions are generally rendered by directed arcs that connect them or by control icons representing the actions of the behavior on the transition [OMG 2005].

Odell [Odell 2000] has already proposed to use statecharts to specify the internal processing of a single agent based on Singh's notion of agent skeletons [Singh 1998]. On the other hand it was not enough to design the feedback loop as we proposed in the last section.

We propose Coordinated Statecharts as a combination of statecharts [Harel 1988] with action and communication of agent and environment behaviors to allow the design of feedback loops. We consider the environment as an explicit part of multi-agent systems, considering both the environment and the agents as first-order abstractions. The rationale for making the environment a first-order abstraction in multi-agent systems is presented in [Weyns 2005b, Weyns et al. 2005, Weyns et al. 2007]. In self-organizing sys-

PUC-Rio - Certificação Digital Nº 0621318/CA

tems the environment is an essential part of every multi-agent system, and the environment provides an exploitable design abstraction to build multi-agent system applications [Weyns et al. 2007]. 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.

A State Machine Diagram describes discrete behavior modeled through finite state-transition systems. The UML 2 behavioral state machines [OMG 2005] is an object-based variant of Harel state charts. The most noticeable concepts of statecharts are hierarchy, orthogonality and submachine [Harel 1988]. The hierarchy concept allows hierarchical state decomposition, i.e., nesting of states within states. The outer enclosing state is called a superstate, and the inner states are called substates. The problem is that the parent state is always in a single child state.

Another important concept, called orthogonality, allows a state on one statechart to be decomposed into two or more concurrent and independent orthogonal regions. Each of the orthogonal regions is named and operates independently of the other regions, and the state of the entire machine or enclosing superstate is represented by a combination of active states of the orthogonal regions. For instance, if a statechart consists of two, X and Y, orthogonal regions. When an event occurs, it is transferred to both orthogonal regions X and Y simultaneously, resulting in the two final states for each region. It provides little concurrency support when dealing with concurrent real-time tasks, in this case, agent behaviors.

Regarding the submachine concept, it is often convenient to reuse a fragment of a state machine in other state machines. A state machine can be given a name and referenced from a state of one or more other machines. The target state machine is a submachine, and the state referencing it is called a submachine state. The problem is that it allows only a single behavior within itself at a time, thus executing concurrent behaviors (for each agent/environment) only in a sequential manner.

Coordinated statecharts extend the orthogonal behavior to support self-organizing mechanisms. Each agent behavior can be considered as an orthogonal behavior with broadcasting capabilities. But in broadcasting [Yacoub 1998], for instance, when an event occurs, it is transferred to all orthogonal regions simultaneously, resulting in the several (the number of regions) final states. Therefore, how could you have orthogonal behavior coexisting although not being activated at the same time? Furthermore, how could you detach this behavior one from another, so you can reuse it in other models?

PUC-Rio - Certificação Digital Nº 0621318/CA

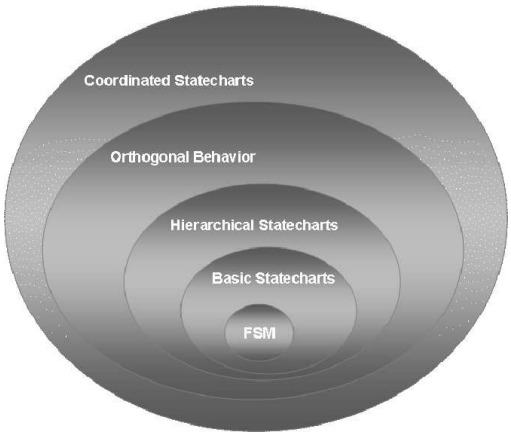

Figure 3.1: Cordinated Statecharts Concept

(b) Customized Meta-model

Coordinated statecharts address these issues by composing behaviors in parallel. With coordinated statecharts, a behavior is a particular instance of the agent or environment in a scenario that represents a typical path through the state space within a single state machine, i.e., an ordered sequence of state transitions triggered by events and accompanied by actions. I.e., each agent and environment behavior is designed using behavioral state machine diagrams. Each behavioral state machine diagram is composed of actions and can communicate with all the other diagrams through a communication channel and the desired feedback loop appears as a result of that communications/coordination.

We are not addressing the agent local interactions through protocols and messages in this work since there has been a huge effort from the agent research community about inter-agent communication. For instance, MAS-ML or AUML protocols diagrams can be combined with the work proposed here for designing interaction protocols.

In order to better understand the extensions, Fig. 3.2 shows the new meta-classes that have been proposed. First, the *CoordinatedStateMachine* meta-class extends the *StateMachine* meta-class, which in turn extends the *Behavior* meta-class. Any agent or environment behavior can be coordinated using *CoordinatedStateMachine*. It adds the composition relationship which means that a *CoordinatedStateMachine* only exists if connected to another *CoordinatedStateMachine*. Another variation that is not represented in the diagram is that a *CoordinatedStateMachine* may not have a final state. Which means: there is always an active state to that agent or environment behavior waiting for receiving an event or to complete an action.

PUC-Rio - Certificação Digital Nº 0621318/CA

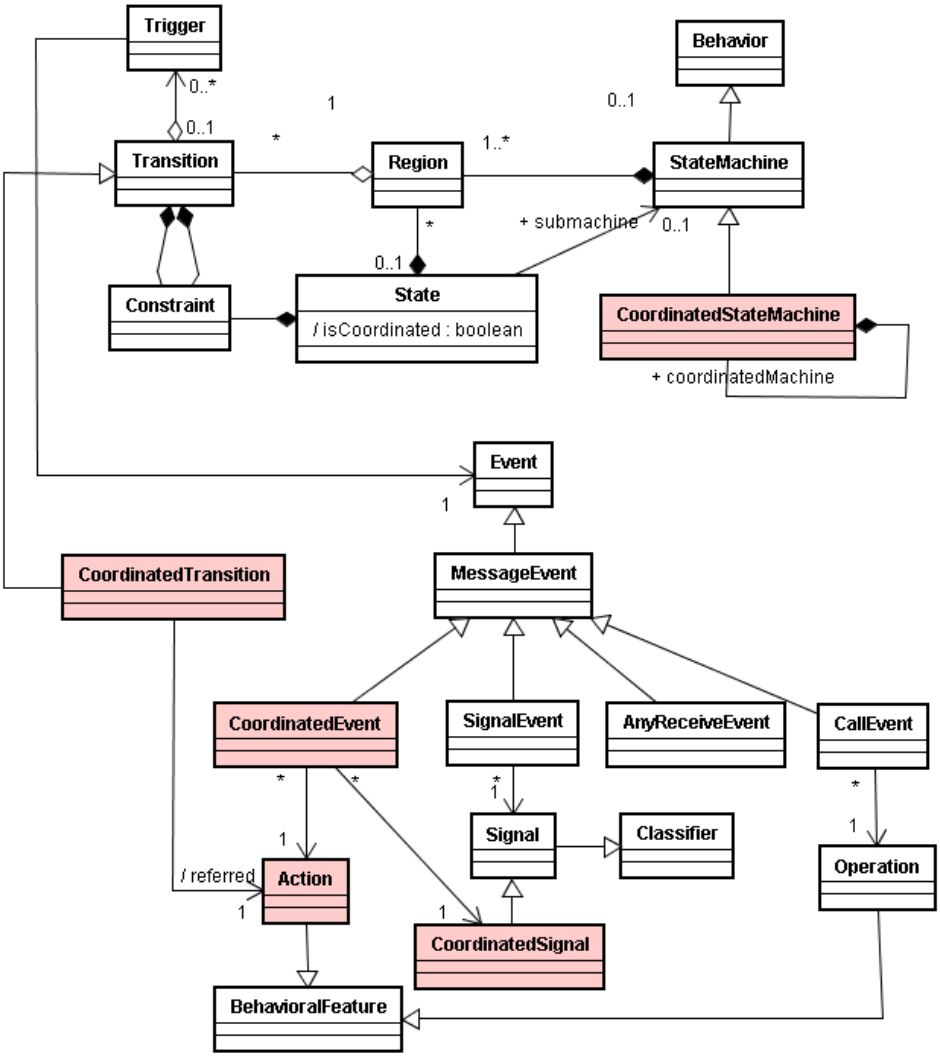

Figure 3.2: The customized UML 2.0 meta-model

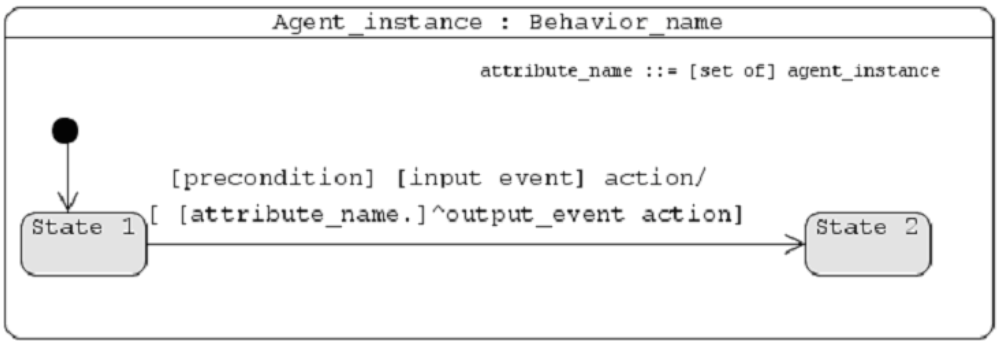

Figure 3.3: The coordinated state machine notation for an agent's behavior

PUC-Rio - Certificação Digital Nº 0621318/CA

All transitions of a *CoordinatedStateMachine* must be coordinated transitions. The *CoordinatedTransition* meta-class extends the *Transition* meta-class with regard to relationships with other meta-classes, such as Vertex, but it redefines the semantics.

In UML, a trigger that causes a transition to occur is called as an event or action. An event can appear synchronously or asynchronously. Call and exception are synchronous events whereas a signal event is asynchronous. A signal is an event sent by another object.

Apart from the operation call event, events are generally used for expressing a dynamic behavior interpretation of coordinated state machines. An event that is not a call event can be specified on coordinated transitions.

The problem with the original signal event concept is that the receiving object handles the signal instance as specified by its receptions, like a call event, at the time that the receiving object receives the signal. On the other hand, agents are autonomous. They may not execute an action right on time if they so decide. Therefore, the signal cannot be specified as a call event at the target object. Due to this constraint, we extended the Signal element with the *CoordinatedSignal* meta-class. A new event meta-class was defined, called *CoordinatedEvent*, which represents the receipt of an asynchronous coordinated signal instance. A coordinated event may, for example, cause a state machine to trigger a transition when the specified action is executed due to the coordinated event and agent's decision. Nevertheless, an *Action* meta-class is a *BehavioralFeature* that represents any agent or environment action according to its decision-making process. An action is executed without explicitly being called by other objects or agents. Agents interact with one another and the environment, sending and receiving messages or sending and receiving coordinated signals.

Fig. 3.3 illustrates the coordinated state machine notation for an agent's behavior. From the diagram top you have to define the instance of the agent for that behavior state. If it was an environment behavior, then the agent instance name has to be replaced by the environment instance name. Each behavior state diagram must start with a transition. The behavior state of the agent (or environment) instance may change according to a transition firing (action execution); the transition will only be fired if the agent executing the specified behavior is in State 1 and according to the input event (a coordinated signal send by other agents) received and the evaluated pre-condition, if specified.

In order to identify which entity would perceive the output event (a coordinated signal sent to other agents) in a complex composition behavior, attributes can be defined and specified in the transition before the "caret"

PUC-Rio - Certificação Digital Nº 0621318/CA

symbol (^) and the output event to be perceived by the entity (ies).

We also classify the coordinated signals as:

(i) emission: signal an asynchronous interaction among agents and their environment. Broadcasting can be performed through emissions;

(ii) trigger: signal a change of agent state as a consequence of a perceived event. For instance, an agent can raise a trigger event when perceiving an emission event which changed its state;

(iii) movement: signal an agent movement across the environment;

(iv) reaction: signal a synchronous interaction among agents, however without an explicit receiver. It can be a neighbor of the agent or the environment; and

(v) communication: signal a message exchange between agents with explicit receivers (one or more).

Note that when the coordinated signal is an emission, the agent does not wait for a response, while in a communication it may wait. Each of those coordinated signals may be raised by actions performed by agents or by the environment and updates their states. Furthermore, the messages exchanged through communication events are interaction protocols and should be FIPA [FIPA 2009] compliant so they can be understood by the receiver agents.

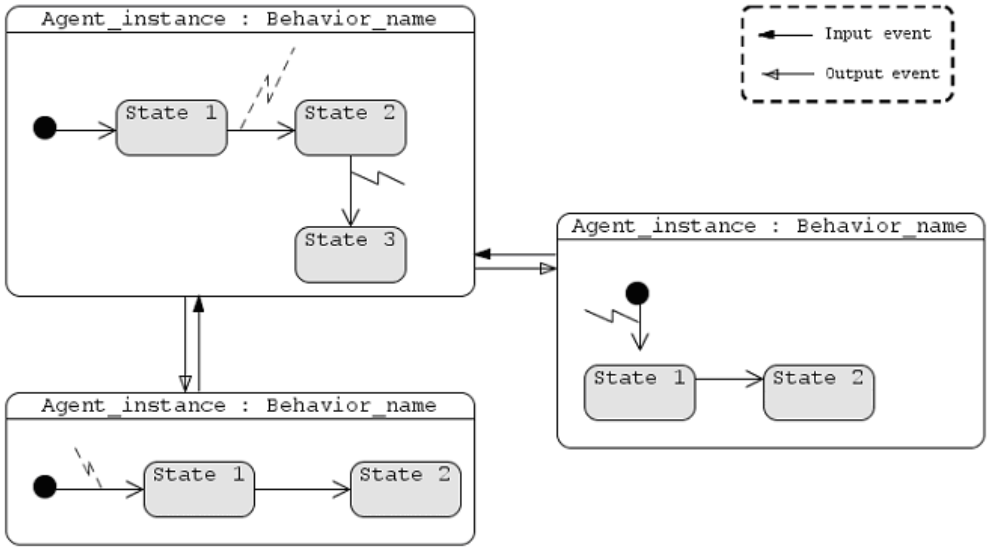

Figure 3.4: Coordinated state machine notation - input versus output coordinated signal perceptions

Fig. 3.4 shows how behaviors can communicate. The figure shows the coordinated state machine communication channels and input versus output

PUC-Rio - Certificação Digital Nº 0621318/CA

coordinated signal perceptions. For instance, an output coordinated signal from the upper left model from State 1 to State 2 (dashed line[1]) may be perceived as an input coordinated signal that starts at the bottom left model and takes it to State 1. The arrows between behaviors' state models show how they communicate among themselves and whether they perceive an input or an output coordinated signal.

3.3 Exemplar Application: Toward a Self-Organizing Pattern Language

This section illustrates the Coordinated Statecharts use through the proposal of a self-organizing pattern language. A pattern language is a set of patterns that are used together to solve a problem. It guides a designer by providing workable solutions to several of the problems known to arise in the course of design. This section proposes a pattern language for self-organizing systems. The language is compliant with the well-known foundations for self-organized systems, which are previously defined in conceptual design patterns [De Wolf 2007a, Gardelli 2008].

The conception of our pattern language has two purposes:(i) the description of five self-organizing mechanisms as patterns, and (ii) the organization of these patterns as a comprehensive pattern language for self-organizing software systems. The basic patterns are [Gardelli 2008]: Diffusion, Evaporation, and Aggregation. They were isolated from the other patterns in the language so that they can be used individually. The combination of basic patterns is required to produce more complex patterns of self-organizing systems as Gradient Fields.

This section presents several contributions. First, to the best of our knowledge there is no evidence in the literature of a pattern language in self-organizing systems areas, only isolated design patterns or combination of them to achieve a domain specific macroscopic behavior. Second, the existent patterns have not been presented at the modeling level such as statecharts, but rather only at the conceptual architectural level. And third, during the pattern language description the already known patterns are described with regard to the presented architectural models and dynamics. This gives a deep knowledge about how those patterns can be realized using a reusable architecture. Hence, the low level allows them to be implemented in a framework and used as templates for many applications.

[1]The dashed lines are illustrative and have no semantics or representations at the meta-model customizations

PUC-Rio - Certificação Digital Nº 0621318/CA

The self-organizing mechanisms are described in a format known in mainstream software engineering which promotes their usage. We combine the pattern scheme presented in the Gang of Four (GoF) [Gamma et al. 1995] and in the catalog of patterns illustrated with UML (Patterns in Java). Therefore, we have based the pattern language with the following structure:

Context A specific scenario motivating the pattern need and what should be a design solution.

Problem What is solved by this pattern? Engineers compare this section with their problem in order to select coordination mechanisms.

Applicability What are the situations in which the pattern can be applied? What are examples of poor designs that the pattern can address? How to recognized these situations?

Forces What are the considerations that lead to the general solution presented in the Solution section?

Solution[1] How does the pattern solve the problem that the pattern addresses?

Consequences How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspects of the solution does it let the designer vary independently?

Implementation Factors What pitfalls, hits, or techniques should you be aware of when implementing the self-organizing pattern? Are there language specific issues?

Example[1] A scenario that illustrates a design problem and how the classes and or objects, agents and environment structures in the pattern solve the problem. The scenario will help one understand the more abstract description of the pattern that follows.

Known Uses What are examples of the pattern found in real systems?

Moreover, the pattern language has a micro-architecture that focuses on the forces acting over the instantiation of our generic structure and behavior for the basic patterns. Figure 3.5 is a directed acyclic graph of dependence among patterns. An edge from pattern A to pattern B means pattern B is generated from pattern A. All other patterns are combinations of the three

[1]Those sections contain novel reusable design details at the low level if compared to the existent design patterns in the literature

PUC-Rio - Certificação Digital Nº 0621318/CA

basic patterns. And all five self-organizing patterns instantiate the micro-architecture. A walk on the graph is directed by two questions: What self-organizing mechanisms should be used to address application needs? And how should the self-organizing mechanisms be structured as reusable and flexible components?

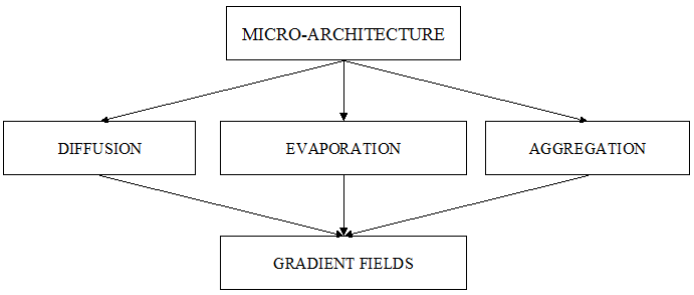

Figure 3.5: Self-Organizing Design Patterns and Their Relationships

PUC-Rio - Certificação Digital Nº 0621318/CA

In order to illustrate the applicability of this pattern language,the self-organizing basic patterns is used over the automated guided vehicle warehouse transportation system case study described in chapter 2, subsection II.3.a. Only the self-organizing-based aspects of such applications are considered.

(a) MICRO-ARCHITECTURE PATTERN

Context

Two or more entities are coordinated through self-organizing mechanisms in an environment. The design of self-organizing components should be modular so that they can be easily combined to achieve the target coordination.

Problem

How to design a flexible agent-oriented micro-architecture for a self-organizing design in order to facilitate component reuse?

Applicability

When defining the best combination of self-organizing mechanisms to achieve optimal coordination. When a generic micro-architecture to several kinds of self-organizing mechanisms is necessary.

Forces

The dependencies between coordination features and application code should be minimized in order to facilitate reuse. The readability of programs with self-organizing code should be increased.

Solution

The Environment is inhabited by Agents and may contain Sub-Environments. The Environment manages the Space that contains Locations. Each Agent is situated in one Location. A Location may have several Agents and Events. An Agent perceives the Events in each Location and may react or not to the events (Figure 3.6).

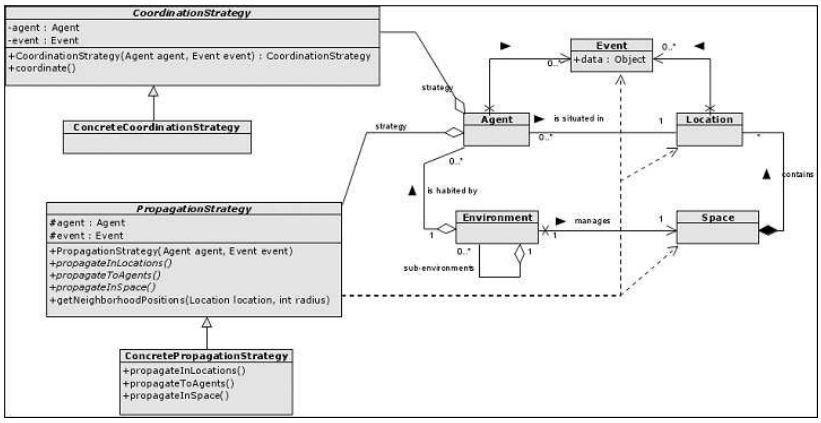

Figure 3.6: Micro-Architecture Structure

For each coordination action, i.e. the agent reaction to a gradient, or propagation rule action there is an abstract class representing the Strategy Design Pattern [Gamma et al. 1995]: the *CoordinationStrategy* and *PropagationStrategy* classes. The Strategy pattern is useful for dynamically swapping the algorithms used in an application. The Agent is the Context of the strategies. Hence the Agent must define them to later execute their behaviors.

The *coordinate()* action will call one of the concrete coordination strategies. And the *propagate()* action will first choose from one of the propagation types, in locations, to agents or in space implemented by the respective operations *propagateInLocations()*, *propagateToAgents()*, *propagateInSpace()*.

Figure 3.7 illustrates the micro-architecture dynamics using Coordinated Statecharts. The A1 Initiator Agent starts the coordination mechanism through the emission of an event, for instance the *GF* event. An A2 agent will trigger the *GF* event and will propagate it in the neighborhood locations or spaces or agents according to the propagation rules. At the location, the propagation might represent the addition of the *GF* event in the location or its removal. At the space, the propagation happens in all the locations it contains.

PUC-Rio - Certificação Digital Nº 0621318/CA

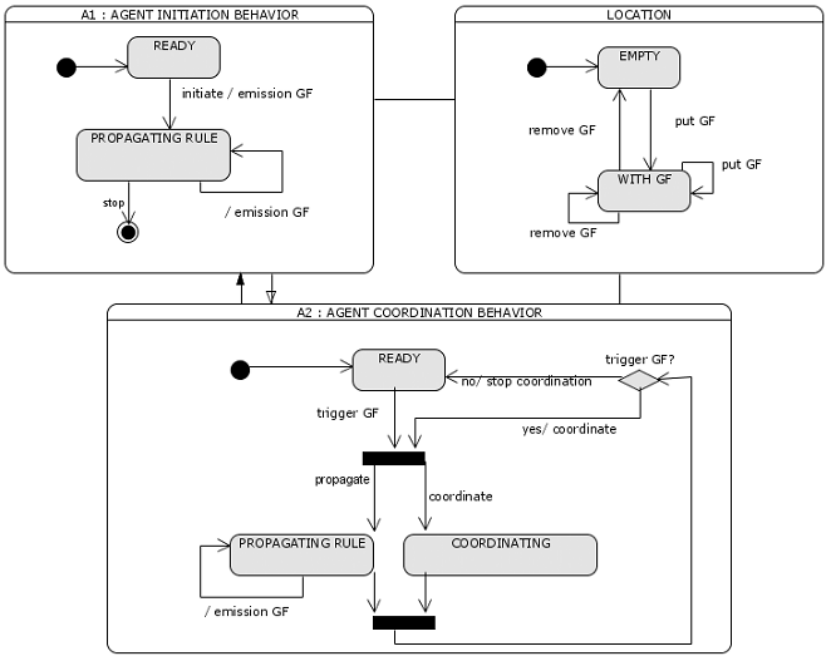

Figure 3.7: Micro-Architecture Dynamics

Other agents will perceive the event at the time of the propagation in the case the event is propagated to their locations or when they move their locations. Once an agent triggers the *GF* event, it decides for starting the coordination. It can also spread more *GF* events or stop the coordination. This decision depends on the coordination rules. Once the coordination process is stopped, the feedback loop is closed.

Consequences

- All the self-organizing mechanisms will present a common behavior and environment structure with concepts of Space and Location.
- Feedback loops can be represented and reused when instantiating the micro-architecture.
- Ad hoc implementations of self-organizing mechanisms could perform better than micro-architecture instantiation if does not use an object-oriented approach.
- Flexible and adaptable systems with self-organizing mechanisms can be more easily obtained when coordination and propagation algorithms are decoupled from their implementations, and these two are, in turn, decoupled from the self-organizing mechanisms.
- Behaviors are defined as separate interfaces or abstract classes and their corresponding concrete specific classes.

PUC-Rio - Certificação Digital Nº 0621318/CA

Implementation Factors

This pattern can be easily developed with object-oriented programming languages. Middlewares with space virtualization can be used to realize the micro-architecture relationships, structure and dynamics. For instance, Tuple Spaces based middlewares [Mamei 2004] and MESOF framework [Gatti 2009], which is an implementation of the architecture presented in chapter 4, provide space virtualization.

In the literature one can find that there are different ways to implement statecharts. The most common technique to implement statechart is the doubly nested switch statements with a "scalar variable". The latter is used as the discriminator in the first level of the switch and event-type in the second [Douglass 1998, Ali 1999, Rhapsody].

Another approach uses the concept of object composition and delegation [Niaz 2004] and extends the State Design Pattern [Gamma et al. 1995]. In this case, each state in the statechart diagram becomes a class. Each transition from that state becomes a method in the corresponding class and each action becomes a method in the context class that, in our case, will be the agent behavior. The context class delegates all events for processing to the current state object. It makes it possible to easily compose behaviors at run-time and to change the way they are composed. Although they have shown that this approach reduces source code in comparison to the first one, with a few more agents there would be an explosion of small classes since an Agent might have several behaviors. Moreover, the event cannot be implemented as a method. The event has to be added to a list that the agent manages and process the event whenever desired.

Example

An Automated Guided Vehicle (AGV) warehouse transportation system that uses multiple computer guided vehicles which move loads in a warehouse. Another example is routing service applications in overlay networks [Babaoglu 2006], which are logical structures built on top of physical network. And also mobile ad-hoc networks (MANETs) [Babaoglu 2006] which are a set of wireless mobile devices that self-organize into a network without relying on a fixed structure or central control.

Recall that in the AGV warehouse transportation system the AGVs move loads (e.g. packets, materials) in a warehouse. Each AGV can only conduct a limited set of local actions, such as move, pick load,

PUC-Rio - Certificação Digital Nº 0621318/CA

and drop load. The goal is to efficiently transport incoming loads to their destination. The AGV problem is dynamic: many lay-outs, loads arrive at any moment, AGVs move constantly and fail, obstacles and congestion might appear, etc. AGV movement should result in feedback to each other and the environment.

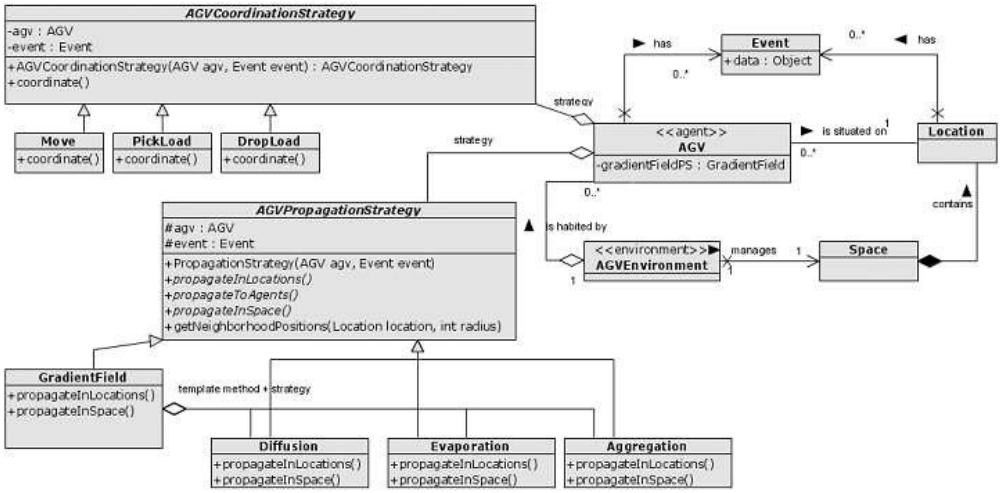

Figure 3.8: AGV Structure

Load dispatching means "assigning" incoming loads to suitable AGVs. A load is only permanently assigned to an AGV when it has picked up the load. Until that moment, other AGVs that become better suited should be able to take over. In the case of routing, for moving toward a pick-up station and after a load is picked up, the AGV is routed through the factory.

The dispatching and routing activities require a mechanism that enables aggregation and calculation of extra information while flowing through intermediate stations.

Gradient fields allow this [De Wolf 2007a]. The pick up stations generate gradients while they have loads to be delivered, and propagate them in the neighborhood. The AGVs also propagate gradients of movement in the environment. Such gradients can be used for information about obstacles and congestions.

Figure 3.8 illustrates the AGV structure and Figure 3.9 illustrates the AGV dynamics, both as an instantiation of this pattern language. Each station is a Space with one or more Locations. Each AGV is an agent. The global execution contains a global environment (*AGVEnvironment*) where the stations and the AGVs are situated. Each action that the AGV might take is realized as a coordination strategy (e.g. *Move*).

PUC-Rio - Certificação Digital Nº 0621318/CA

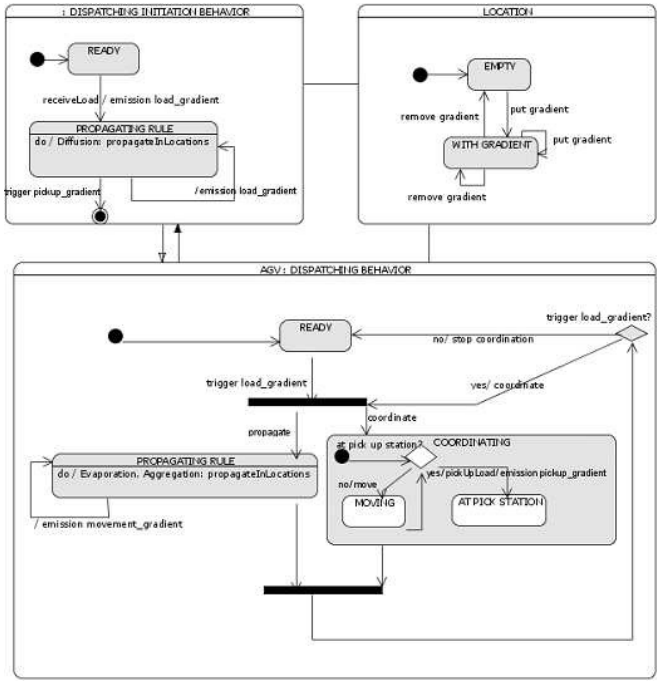

Figure 3.9: AGV Dynamics: Dispatching Property

The *GradientField* strategy is composed of the three basic strategies: Diffusion, Evaporation and Aggregation.

Known Uses

All the self-organizing patterns described in next section are widely used in systems as motion coordination [De Wolf 2007a], data clustering [De Wolf 2007a, De Wolf 2007], autonomic application servers, biological computational simulation [Gatti 2009], TOTA [Mamei 2004], and can be instantiations of the micro-architecture pattern.

(b) DIFFUSION PATTERN

Context

A distributed entity wants to send information (represented by gradients) to a distant entity that is unaware of neither the entity nor its location.

Problem

How can the information be propagated in order that the distant entities react to them?

Applicability

When distributed entities need to be coordinated without a central control and without the knowledge about complete neighboring space.

PUC-Rio - Certificação Digital Nº 0621318/CA

Forces

Without a central control or knowledge about the environment, distant entities cannot be coordinated, unless a diffusion mechanism propagates the information in the environment and guides the entities' actions.

Solution

The *coordinate()* method of the concrete coordination strategy will be executed. The *propagateInLocations()* method of the propagation strategy implemented by the Diffusion class will be called by the *coordinate()* method. It will fire an event stamped with a weight to the location in the neighborhood of radius one. Each entity on the target locations will trigger the event and will propagate in the same way (except to the locations already with the event). When propagating, the weight will be decreased locally and correspondingly increased in the neighborhood.

Consequences

Gradients are propagated in all directions without taking into account other gradients already present in different spaces or locations. There is the risk of some spaces having many gradients and there being too little in other spaces. For instance, in the AGV problem this pattern will work properly, because the AGVs will avoid these locations and consequently avoid congestion. However, in situations where the gradient represents loads and the goal is to achieve an equal distribution of loads the result is an inefficient load balancing mechanism.

Implementation Factors

Different radius sizes can be used for this pattern. It mostly depends on the kind of application being developed and on the access to the available neighborhood.

Example

An AGV wants to send information about its position when it is moving. Hence, the other AGVs can avoid congestion. The AGV will call the *coordinate()* method of the coordination strategy implemented by the Move class, which in turn will call the *propagateInLocations()*. Each AGV on the target locations will trigger the event and will propagate in the same way (except to the locations already with the event) but decreasing the weight locally and correspondingly increasing the weights in the neighborhood. The AGV might move or stay at the same location depending on the event weight.

PUC-Rio - Certificação Digital Nº 0621318/CA

Known Uses

A common use of this pattern is in the problem of calculating global functions and load balancing [Babaoglu 2006].

(c) EVAPORATION PATTERN

Context

Gradients were propagated in locations in the environment in order to coordinate (for instance, attract or repel) distributed entities. Once the coordination is achieved or the goal is satisfied, the gradients must disappear.

Problem

How can the gradients disappear from their locations?

Applicability

When the application is overwhelmed by information or gradients released.

Forces

The memory must be released to achieve higher performance and the information is no longer useful.

Solution

From time to time, the Environment will actively or reactively call the *propagateInLocations()* or *propagateInSpace()* methods of the Evaporation class. Thus, it will apply the evaporation rate in obsolete gradients. Obsolete gradients can be gradients not being perceived by Agents in a period of time. The evaporation rate, for instance, can be decreasing the gradient's weight until it reaches zero.

Consequences

Gradients cannot be recovered once evaporated.

Implementation Factors

The choice of the Environment actively evaporates gradients, or reactively (in response to a specific event) depending on performance requirements.

Example

The load_gradient event fired by the Dispatching Initiation Behavior will be diffused in the Environment. However, once the pickup_gradient event is fired by the AGV when it is at the Pickup Station and picks the load

PUC-Rio - Certificação Digital Nº 0621318/CA

up, the *AGVEnvironment* triggers this event and calls the *propagateInLocations()* or *propagateInSpace()* methods of the Evaporation class in order to apply this pattern and evaporate all load_gradient events propagated in locations. Hence, other AGVs will not look for this load.

Known Uses

The most common uses of this pattern is in stigmergy-based systems and pheromone path-based applications [De Wolf 2007, Parunak 2005].

(d) AGGREGATION PATTERN

Context

In a feedback loop it might be useful to reinforce information in order to an emergent property (an information trace) appear as a response of the reinforcement.

Problem

How to reinforce a positive or negative feedback loop in a self-organizing system?

Applicability

When Agents are guided by the gradient with higher intensities in order to produce learning paths.

Forces

If the Agents do not follow the gradient with higher intensities, they might take too long to reach the coordination goal. The shortest paths save time, resources and increase performance.

Solution

Each time the same information is deposited in a Location, its intensity is increased locally. The Evaporation class implements this behavior through the *propagateInLocations()* and *propagateInSpace()* methods that can be called by Agents or Environment.

Consequences

Shortest paths are produced from the distributed reinforcement learning process, although not necessarily the shortest path of all; i.e., for space circumstances a path emerges but might not be the shortest.

Implementation Factors

There are two main factors that impact on the result of this pattern at the implementation level: the rule for increasing the gradient intensity

PUC-Rio - Certificação Digital Nº 0621318/CA

and the neighborhood radius. Also, how the intensity is modeled may influence the result. It could be a simple or more complex structure.

Example

For each AGV there would be a learning path so that they avoid other AGVs' paths (to avoid congestion). Thus, on each call to coordinate() method of the Move class, the *propagateInLocations()* method of the Evaporation class will be executed and will propagate the correspondent gradient exactly and only to the new Location.

Known Uses

This pattern is commonly used in stigmergy-based systems and pheromone path-based applications [De Wolf 2007, Parunak 2005]. It is also used in adaptive routing algorithms for wired and mobile networks [Babaoglu 2006].

PUC-Rio - Certificação Digital Nº 0621318/CA

(e) GRADIENT FIELDS PATTERN

Context

A system composed of distributed autonomous entities must be self-managed, self-configured to achieve a global coordination function.

Problem

How to adaptively orchestrate distributed autonomous entities achieving a pattern formation?

Applicability

When Agents must be coordinated to achieve macro properties without any external or internal central control.

Forces

A centralized solution is often a bottleneck and single point of failure in a very dynamic situation. The solution must be flexible to achieve robustness.

Solution

This pattern is the composition of the Diffusion, Evaporation and Aggregation patterns. There are two basic ways to achieve the composition: they can be randomly composed (i) using the three patterns at the same time in the Environment; or (ii) composing them while propagating the information. As a result, agents follow the shape of the coordination combined field. If one wants to compose in a controlled manner

the composition can be achieved using the Template Method pattern [Gamma et al. 1995]. It will prevent others from replacing all your composition implementation and offering them a specific extension point.

Consequences

Usually, following the gradient field is the shortest path toward the initiator of the field. Although this pattern can be considered greedy because of the strictly local perspective of the agents.

Implementation Factors

Create a *GradientField* subclass of the *PropagationStrategy* which contains the gradient to be propagated. And delegate the order of the basic propagation strategies to the *GradientField* class.

Example

Instead of perceiving the individually events in the AGV example (for instance, the load_gradient event), an AGV will perceive an event that contains the combined gradient and will react in response to it.

Known Uses

This pattern is commonly used in intelligent agents exploring the web, spatial shape formation, urban traffic management [De Wolf 2007a], etc.

3.4 Chapter Remarks

This chapter presented a UML-based notational support for the engineering of self-organizing systems. We have presented the Coordinated Statecharts concept which motivated the UML meta-model customization in order to provide both structural and behavioral features that help with the addition of semantics to the proposed modeling abstraction. In order to illustrate the use of the modeling abstraction, we have presented a pattern language built on top of self-organizing patterns that already exist in the literature. This pattern language contributes to the state of the art in many ways: (i) providing a guide to use the patterns, (ii) providing reusable structures and dynamics, and finally (iii) providing a way of combining basic patterns into a complex one – the Gradient Fields. With the results reported in this chapter, we believe that the self-organizing system designer has now available a design abstraction complemented with a notational support to help with her design task.

PUC-Rio - Certificação Digital Nº 0621318/CA