PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 16/09

A Pattern Language for Self-Organizing Systems

Maíra Athanázio de Cerqueira Gatti

Carlos José Pereira de Lucena

Alessandro Fabricio Garcia

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 16/09 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena June, 2009

A Pattern Language for Self-Organizing Systems

Maíra Athanázio de Cerqueira Gatti, Carlos José Pereira de Lucena and Alessandro Fabricio Garcia

Laboratório de Engenharia de Software – LES
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brasil

{mgatti, lucena, afgarcia}@inf.puc-rio.br

Abstract. Developers and maintainers of self-organizing software systems need design patterns to facilitate design reuse. To tackle this problem, we propose a catalogue of agent-oriented design patterns for structuring the pivotal mechanisms of a self-organizing system. The presented pattern language has a twofold purpose. First, it defines the common self-organizing behavior and the underlying environment structure. Second, all the patterns describe how the information flow should be designed in a complex self-organizing system. The pattern language also shows how the basic patterns can be composed to design more sophisticated self-organizing mechanisms. The automated guided vehicles application was chosen as a unified example of the pattern language usage, while a number of well-known pattern uses are also described.

Keywords: Multi-Agent Systems, Self-organization, Design Patterns, Pattern Language.

Resumo. Desenvolvedores de sistemas de software auto-organizáveis precisam de padrões de projeto para facilitar o reuso do projeto. Neste sentido, propomos um catálogo de padrões de projeto orientados a agentes que estrutura os principais mecanismos de um sistema auto-organizável. A linguagem de padrões descrita possui dois propósitos principais. Primeiramente o de definir um comportamento em comum na estrutura do ambiente. E em segundo, os padrões descrevem como fluxos de informação devem ser projetados em um sistema auto-organizável. A linguagem de padrões também mostra como os padrões básicos podem ser compostos para projetar mecanismos mais sofisticados de auto-organização. A aplicação de veículos guiados automáticos foi escolhida como um exemplo unificado do uso da linguagem de padrões, enquanto que o uso de padrões conhecidos também é descrito.

Palavras-chave: Sistemas Multiagentes, Auto-Organização, Padrões de Projeto, Linguagem de Padrões.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br

Table of Contents

1. Introduction 1

2. An Agent-Oriented Pattern Language for Self-Organizing Systems 1

  2.1. Generic Self-Organizing Architecture 2

3. The Automated Guided Vehicles (AGVs) 5

4. Self-Organizing Patterns 7

  4.1. Diffusion 7

  4.2. Evaporation 8

  4.3. Aggregation 8

  4.4. Gradient Fields 9

  4.5. Pheromone Path 10

5. An Implementation Example of GSOA 10

6. Concluding Remarks 12

A Coordinated Statecharts Concepts 12

References 13

1. Introduction

Software architects are increasingly relying on self-organizing mechanisms to design distributed autonomous systems. Each component in self-organizing systems acquires and maintains information about its environment and neighbors without external control. The emergent behavior may evolve or change over time [1][2]. When engineering a self-organizing emergent solution, the problem-solving power mainly resides in the interactions and coordination between agents instead of in intelligent reasoning of individual agents. Documented design practices are essential to provide developers and maintainers of complex self-organizing systems with proper design guidance and reuse. They also facilitate quality assurance processes, such as software verification and testing.

To maximize reuse, patterns are often described at several levels of abstraction; for instance, the patterns in the Gang of Four book [3] are described via class models and implementation examples. However, to the best of our knowledge, there are only conceptual and architectural design patterns described in the literature [4][5][6] for self-organizing systems. The problem is that architectural patterns are high-level strategies that concern the global design properties of a system, whereas design patterns complement by describing commonly recurring structures of communicating components. Furthermore, a pattern language defines a collection of patterns and the rules to combine them according to an underlying architectural style [20]. Pattern languages can be used to describe software frameworks or families of related systems.

This paper proposes: (i) a catalogue of design patterns in the form of a pattern language described at the detailed design and implementation levels; and (ii) a generic agent-oriented self-organizing architecture that guides the structure of each pattern in the language. They are behavioral patterns [3] that help designers define the communication between agents and their behaviors. Furthermore, the patterns help to design the information flow in a complex self-organizing system. The definition of the pattern language was also based on an identification of well-known uses of the documented pattern solutions.

This work is organized as follows. Section 2 proposes the pattern language and a pattern describing a generic software architecture. Section 3 describes an application example using the pattern language: the automated guided vehicles problem. This application is largely used and referenced in the literature of self-organizing systems. Section 4 presents the patterns used in the pattern language. Section 5 presents an implementation example of the pattern language. Finally, we present the conclusions.

2. An Agent-Oriented Pattern Language for Self-Organizing Systems

A pattern language is a set of patterns that are used together to solve a problem. A pattern language guides a designer by providing workable solutions to several of the problems known to arise in the course of design. This section proposes an agent-oriented 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 [4][5].

The conception of our pattern language has three purposes: (i) the definition of a Generic Self-Organizing Architecture (GSOA for short); (ii) the description of five self-organizing mechanisms as patterns, and (iii) the organization of these patterns as a comprehensive pattern language for self-organizing software systems. The basic patterns are [5][10]: 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 or Pheromone Path. GSOA focuses on the forces acting over the instantiation of our generic structure and behavior for the basic patterns.

Figure 1 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. GSOA generates the micro-architecture for the three basic patterns. All other patterns are combinations of these. Thus, all five self-organizing patterns instantiate GSOA. 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?

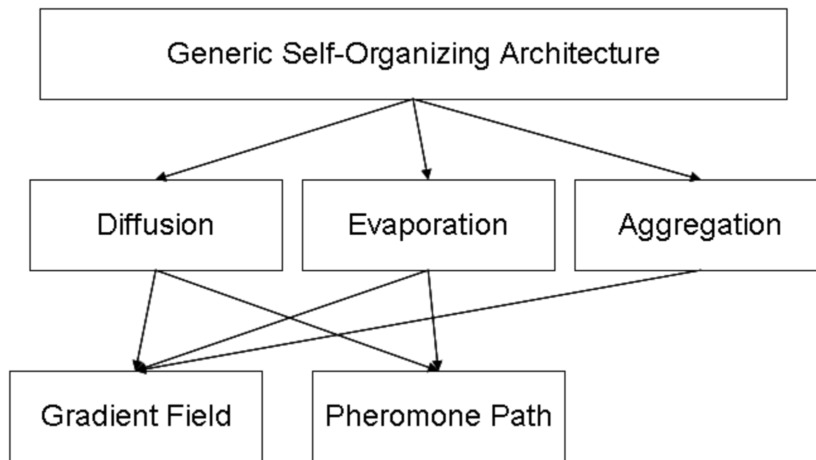

Figure 1. Self-Organizing Design Patterns and Their Relationships

2.1. Generic Self-Organizing Architecture

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 2).

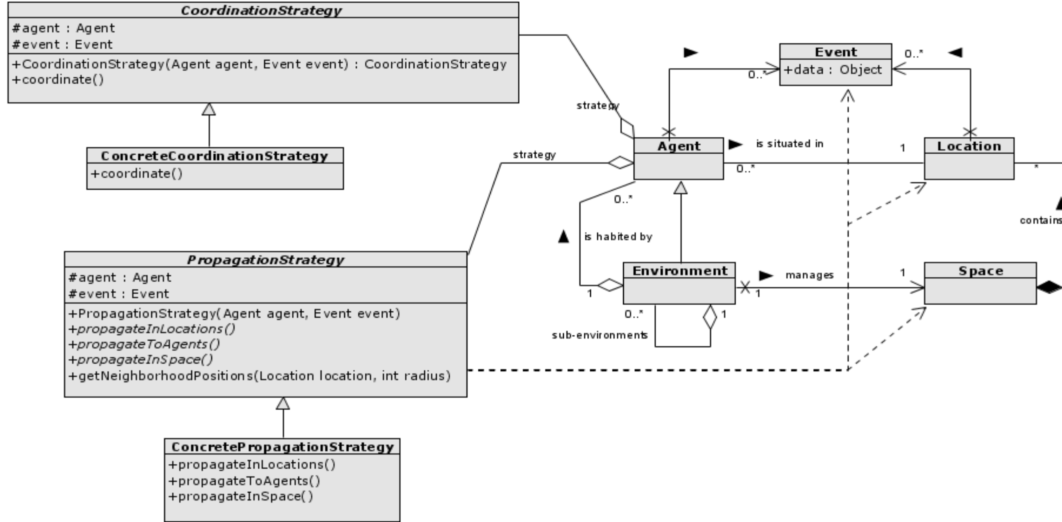

Figure 2: GSOA Structure

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 [3]: 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 illustrates the GSOA dynamics using Coordinated Statecharts that extend the orthogonal behavior to support self-organizing mechanisms design [1]. 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. 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.

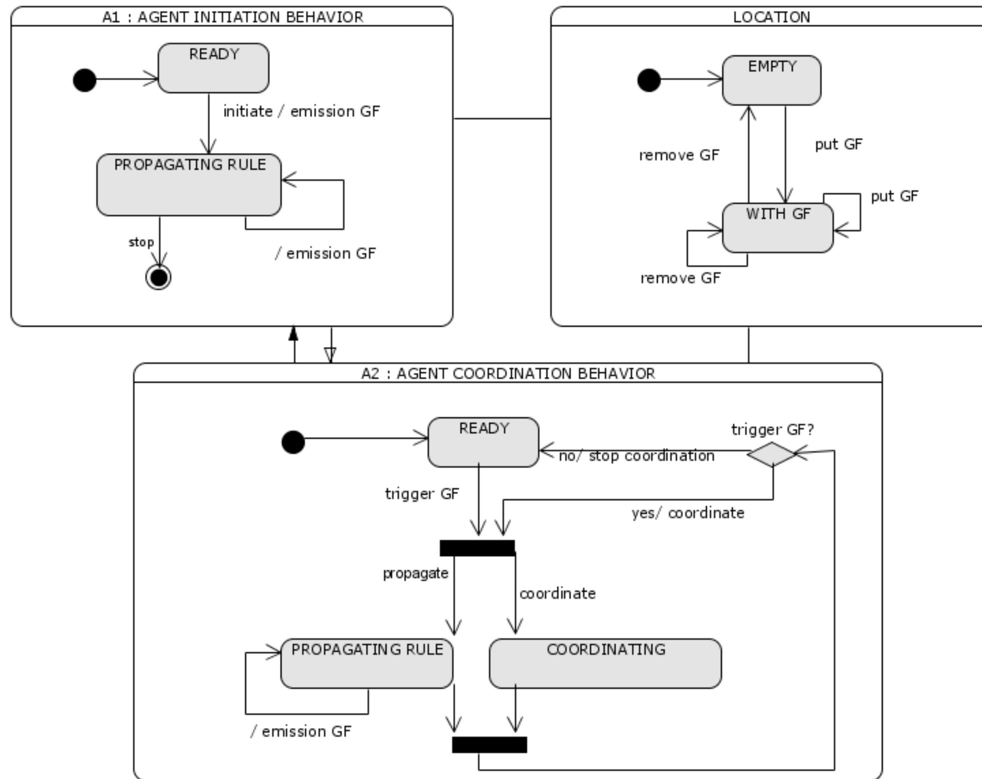

Figure 3. GSOA Dynamics

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 GSOA.
- Ad hoc implementations of self-organizing mechanisms could perform better than GSOA 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.

Implementation Factors

This pattern can be easily developed with object-oriented programming languages. Middlewares with space virtualization can be used to realize the GSOA relationships, structure and dynamics. For instance, Tuple Spaces based middlewares [14] and MESOF framework [11] provide space virtualization.

In 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 [3][11][15].

Another approach uses the concept of object composition and delegation [16] and extends the State Design Pattern [3]. 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. This application is designed in section 3. Another example is routing service applications in overlay networks [17], which are logical structures built on top of physical network. And also mobile ad-hoc networks (MANETs) [17] which are a set of wireless mobile devices that self-organize into a network without relying on a fixed structure or central control.

Known Uses

All the self-organizing patterns described in section 4, and widely used in systems as motion coordination [4], data clustering [4][5], autonomic application servers, biological computational simulation [2][11], TOTA[14], are instantiations of GSOA.

3. The Automated Guided Vehicles (AGVs)

In order to illustrate the applicability of this pattern language, we will use its self-organizing patterns over a large example, the design of an automated guided vehicle warehouse transportation system [10]. We will consider only the self-organizing-based aspects of such applications.

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, 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.

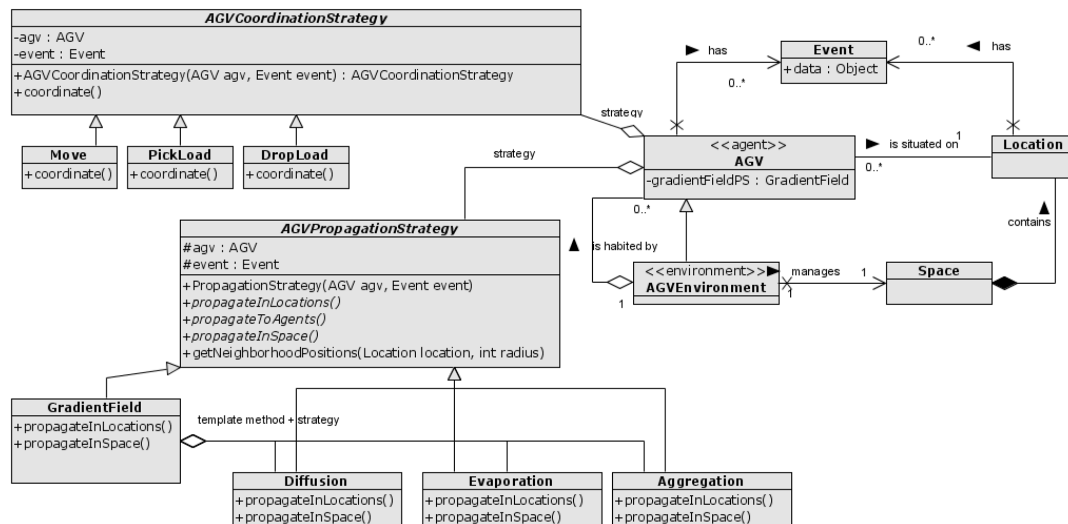

Figure 4. AGV Structure

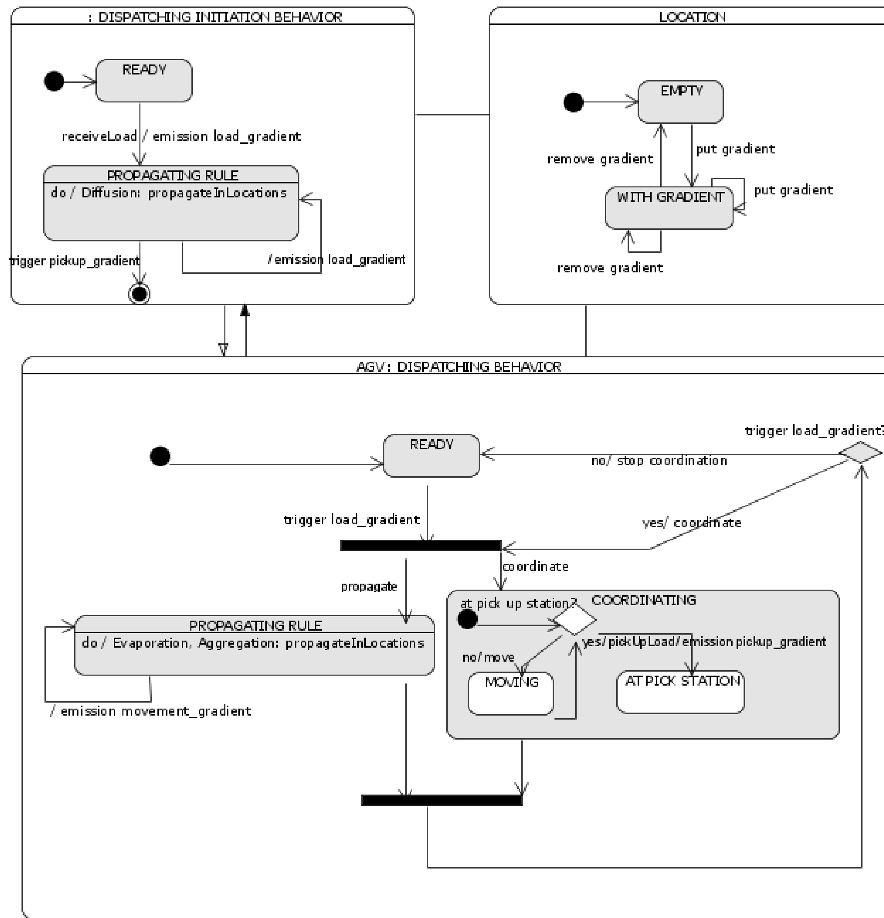

Figure 5. AGV Dynamics – Dispatching Property

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 towards 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 [4]. 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 4 illustrates the AGV structure and Figure 5 illustrates the AGV dynamics, both as an instantiation of the GSOA 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). The GradientField strategy is composed of the three basic strategies: Diffusion, Evaporation and Aggregation.

4. Self-Organizing Patterns

4.1. Diffusion

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.

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.

Known Uses

A common use of this pattern is in the problem of calculating global functions [17],[18], and load balancing [17].

4.2. Evaporation

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 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 [10] and pheromone path-based applications [19].

4.3. Aggregation

Context

In a feedback loop it might be useful to reinforce information in order to an emergent property - a path – 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 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 [10] and pheromone path-based applications [19]. It is also used in adaptive routing algorithms for wired and mobile networks [17].

4.4. Gradient Fields

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 [3]. 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 towards 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 [4], etc.

4.5. Pheromone Path

Due to space constraints the Pheromone Path pattern will not be fully described. However, the solution rationale is the same for the Gradient Fields pattern and known uses can be found in [19], for instance.

5. An Implementation Example of GSOA

A simple implementation of the GSOA and its instantiation are described in this Section using Java code and were developed in the multi-environment self-organization framework (MESOF, for short) **Error! Reference source not found.**. The interfaces `IAgent` and `IEnvironment` are not described in GSOA but used in the MESOF.

Three classes are partially described: `Agent`, `Environment` and the `Diffusion` class, the concrete propagation strategy for the Diffusion pattern. The `Agent` class (Table 1) implements the `IAgent` interface of MESOF and has a reference for the current `Location` and the `Environment` where it runs. It has a list of events, which are updated whenever there is an event in the `Location` where the agent is. Two references to the `CoordinationStrategy` and `PropagationStrategy` classes are needed. The subclasses of the `Agent` class must override the `start()` and `step()` methods. Depending on the state of the agent, it will set the strategies at runtime and call its actions `coordinate()` and `propagate()` that delegates the behavior to the strategies classes initialized.

The `Environment` class (Table 2) extends the `Agent` class, representing an active environment **Error! Reference source not found.** and implements the `IEnvironment` interface. It has a reference to the agents running on it, and eventually sub-environments. The `Environment` manages the `Space` that contains all the `Location` where agents can be situated. If the `Environment` to be instantiated is the main environment, then the `Environment(Id)` constructor must be called, otherwise the `Environment (Id, Environment)` constructer must be called passing the parent `Environment`. The methods `addAgent(IAgent)` and `addEnvironment (IEnvironment)` add agents and sub-environments to the `Environment`. When instantiating the `Environment` class the `start()` method must be called by the `start()` method implemented by the subclass. It starts the existent entities of the environment. The same happens to the `step()` method. The `Agent` class inherits all other behaviors of perceiving and acting.

Table 1. Agent Class

```
class Agent implements IAgent{
   private Id id;
   private Location location;
   private ArrayList<Event> events;
   private Environment env;
   private CoordinationStrategy coStrategy;
   private PropagationStrategy propStrategy;
   public Agent(Id id, Environment env)   {
      this.id = id;
      this.env = env;
      this.events = new ArrayList<Event>();}
   public Id getId(){
      return id;}
   public void receiveEvent(Event ev){
      this.events.add(ev); }
   public Event consumeEvent(){
      if (this.events.size()>0)
         return this.events.remove(0);
      else  return null; }
   public void start(){}
   public void step() {}
   private setCoordinationStrategy (CoordinationStrategy coordStrategy){
      this.coStrategy = coordStrategy; }
   private set PropagationStrategy (PropagationStrategy propStrategy)  {
      this.propStrategy = propStrategy;   }
   private coordinate()   {
      this.coStrategy.coordinate();  }
   private propagate()   {
      this.propStrategy.
          propagateInLocations(); }}
```

Table 2. Environment Class

```
class Environment extends Agent
                   implements IEnvironment {
   private ArrayList<IAgent> agents;
   private ArrayList<IEnvironment> environments;
   private Space space;
  public Environment(Id id)   {
      super(id, null);
      this.agents = new ArrayList<IAgent>();
      this.environments = new ArrayList<IEnvironment>();    }
   public Environment (Id id, Environment env) {
      super(id, env);
      this.agents = new ArrayList<IAgent>();
      this.environments = new ArrayList<IEnvironment>();}

    public void addAgent(IAgent agent) {
       this.agents.add(agent); }
    public void addEnvironment (IEnvironment environment) {
       this.environments.add(environment); }
    public void start() {
       super.start();
       for (IEnvironment subEnv : environments){
              subEnv.start(); }
       for (IAgent agent : this.agents){
              agent.start(); } }
       public void step(){
         for (IAgent agent : this.agents){
              agent.step(); }}
       public Space getSpace (){
              return this.space; }}
```

The `Diffusion` class (Table 3) illustrates the instantiation of GSOA and, hence the MESOF framework. It extends the `PropagationStrategy` class which implements common behavior to all strategies as `getNeighborhoodPositions (Location)`. It is necessary to pass the Agent reference, which is the context, and the `Event` to be propagated.

When the **`propagateInLocations()`** is called, for instance, the neighborhoods are returned and for each neighbor location in the **`Space`**, the event is propagated.

Table 3. Diffusion Class

```
class Diffusion extends PropagationStrategy{
   public Diffusion (Agent agent, Event event){
     super(agent, event);}
   public void propagateInLocations(){
     List<Location> neighbor = getNeighborhoodPositions (agent.getLocation());
      for (Location location: neighbor) {
        //before insertion, decrease weight
        //of the gradient encapsulated by
        //the event object.
        agent.getEnv().getSpace().insertObject(location, event);}}}
```

6. Concluding Remarks

There is increasing use of self-organizing mechanisms in the development of contemporary software applications. As a consequence, a reusable software architecture and design patterns are needed to facilitate the design, implementation and reuse of flexible and self-organizing systems.

In order to achieve this goal, this work presented an agent-oriented pattern language for self-organizing systems. We consider our pattern language to be complete and closed to the self-organizing domain for two reasons: (i) the five patterns, except GSOA, are widely used in many applications, and (ii) the GSOA is a result of the knowledge extracted from the engineering of a self-organizing framework and evaluated in real-world applications [11].

ACKNOWLEDGMENT. This work was supported by MCT/CNPq through the "*Grandes Desafios da Computação no Brasil: 2006-2016*" (Main Computational Challenges in Brazil: 2006-2016) Project (Proc. CNPq 550865/2007-1).

A Coordinated Statecharts Concepts

The foundation of self-organizing representation model [1][2] considered uses a UML-based model and it enables the design of event or data-oriented indirect communication through Coordinated Statecharts.

In a 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.

Coordinated statecharts combines statecharts [12] with action and communication of behaviors to allow the design of feedback loops [7][8]. More specifically, coordinated statechart reuses and adapts the UML 2 behavioral state machine [9]. Each agent and environment behavior is designed using behavioral state machine diagrams (Figure 6). Each behavioral state machine diagram can communicate with all the other diagrams through a communication channel and the desired feedback loop appears as a result of that communications/coordination.

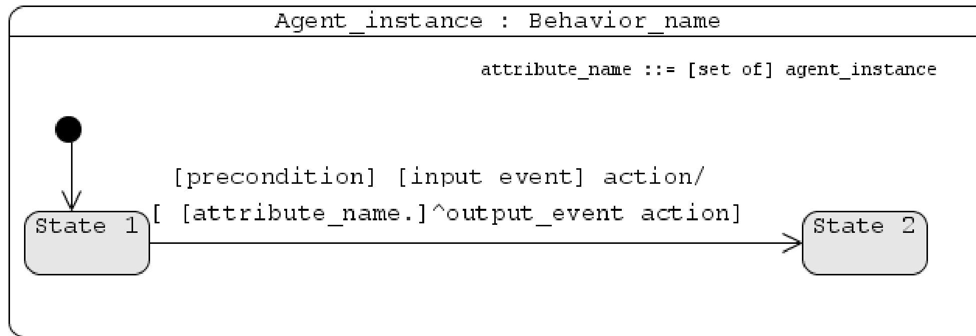

Figure 6. The abstract coordinated statechart for an agent's behavior

Moreover, behaviors are composed of actions. Actions are executed through input events and pre-conditions and raise output events, as described before. The input and output events vary according to the following stereotypes: (i) emission: signal an asynchronous interaction among agents and their environments. 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).

Coordinated statecharts compose 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.

Coordinated statecharts extend the orthogonal behavior to support self-organizing mechanisms. Each agent behavior can be considered as an orthogonal behavior with broadcasting capabilities [13]. But in broadcasting, 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 co-existing 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? Coordinated statecharts address these issues.

References

[1] Gatti, M.A. de C., Lucena, C.J.P.; "A Bio-inspired Representation Model for Engineering Self-Organizing Emergent Systems," XXII SBES, SP, Brazil, 2008.

[2] Gatti, M.A. de C., Lucena, C.J.P.: Engineering Self-Organizing Multiagent Systems based on a Bio-inspired Representation Model and Coordinated Statecharts. Submitted to a Special Issue Track in the ISJ, 25 pgs., 2009.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995 Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc.

[4] De Wolf, T. and Holvoet, T.; Design Patterns for Decentralised Coordination in Self-organising Emergent Systems, in Eng. Self-Organising Systems: Fourth Int. Work-

shop, ESOA 2006, Future University-Hakodate, Japan, 2006, Lecture Notes in Computer Science, Vol. 4335, 2007, pp. 28–49, Springer Verlag.

[5] Gardelli, L., Viroli, M., Omicini; A.; Design Patterns for Self-Organizing Multi-agent Systems. 2nd Int. Workshop on EEDAS'2007. At the 4th IEEE Int. Conf. on Autonomic Computing. June 11th, 2007, Jacksonville, Florida, USA.

[6] De Wolf, T. and Holvoet, T.; Designing Self-Organising Emergent Systems based on Information Flows and Feedback-loops. Proc. of the First IEEE Int. Conf. on (SASO), Editors: Di Marzo Serugendo et al., MIT, Boston, USA, pp 295-298, ISBN 0-7695-2906-2, July 9-11, 2007.

[7] Wiener, N.. Cybernetics or Control and Communication in the Animal and the Machine, Paris, Hermann et Cie - MIT Press, Cambridge, MA, 1948.

[8] Camazine,S., Deneubourg,, J.-L. Franks, N. R., Sneyd, J., Theraula, G., Bonabeau, E.; Self-Organization in Biological Systems. Princeton University Press, 2003.

[9] UML 2.x OMG Specification. http://www.omg.org/

[10] De Wolf, T.; Analysing and engineering self-organising emergent applications, Ph.D. Thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May, 2007, 183.

[11] Gatti, M.A. de C., Lucena, C.J.P.; A Multi-Environment Multi-Agent Simulation Framework for Self-Organizing Systems. In The 10th MABS at AAMAS'09, Budapest, May 2009.

[12] Harel, D.; On visual formalisms. Communications of the ACM, V31 I5 pp514-530, 1988.

[13] Yacoub, S.M. and Ammar, H.H.; A pattern language of statecharts, Proc. Fifth Annual Conf. on the PatternLanguages of Program (PLoP'98), Monticello, IL, USA, 1998, TR #WUCS-98-29.

[14] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In Proc. of the 2nd Int. Conference on Pervasive Computing and Communications. IEEE Computer Society, Washington, DC, USA, 2004.

[15] Gatti, M.A.C., Sangiorgi, U.B., Lucena, C.J.P.de; Towards a Model Driven Approach for Engineering Self-Organizing Multi-Agent Systems. In Monografias em Ciência da Computação, 11/09, Departamento de Informática, PUC-Rio, Brazil, March 2009.

[16] Niaz, A. I., Tanaka, J.; Code Generation from UML Statecharts.in Proc. 7 the IASTED International Conf. on SEA, Marina Del Rey, 2003.

[17] Babaoglu, O., Canright, G., Deutsch, A., Caro, G. A., Ducatelle, F., Gambardella, L. M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., and Urnes, T. 2006. Design patterns from biology for distributed computing. ACM Trans. Auton. Adapt. Syst. 1, 1 (Sep. 2006), 26-66.

[18] Weyns, D., Boucké, N., and Holvoet, T.; Gradient field-based task assignment in an AGV transportation system. In Proc. of the Fifth Int. Joint Conf. on AAMAS (Japan, May, 2006). ACM, New York, NY, 842-849.

[19] Parunak, H. V. D., Brueckner, S. A. and Sauter, J.; Digital pheromones for coordination of unmanned vehicles. In Environments for Multi-Agent Systems, volume 3374 of LNAI, pg. 246–263. Springer, February 2005.

[20] Alexander, C., Ishikawa, S., Silverstin, M. A Pattern Language. Oxford University Press, New York, 1997.