A Pattern Language for Multi-Agent Systems

Danny Weyns
DistriNet Labs, Katholieke Universiteit Leuven, Belgium
Email: danny.weyns@cs.kuleuven.be

Abstract

Developing architectural support for self-adaptive systems, i.e. systems that are able to autonomously adapt to changes in their operating conditions, is a key challenge for software engineers. Multi-agent systems are a class of decentralized systems that are known for realizing qualities such as adaptability and scalability. In this paper, we present a pattern language for multi-agent systems. The pattern language distills domain-specific architectural knowledge derived from extensive experiences with developing various multi-agent systems. The pattern language, consisting of the five interrelated patterns, supports architects with designing software architectures for a family of self-adaptive systems. We illustrate the patters for a case study in the domain of automated transportation systems.

1. Introduction

Self-adaptivity has been proposed as an effective approach to tackle the increasing complexity of constructing and managing modern-day software systems. Self-adaptability endows a system with the capability to adapt itself to changes in its environment and user requirements. Several researchers have argued that software architecture provides the right level of abstraction and generality to deal with the challenges of self-adaptability. One of the major challenges in self-adaptive systems is dealing with distribution and decentralization [1]. Decentralized control is crucial for quality requirements such as openness and scalability.

Over the past 8 years, we have been studying decentralized architectures for realizing self-adaptivity based on multi-agent systems (MAS). A MAS architecture structures the software in a number of interacting autonomous entities (agents) that cooperatively realize the system goals. Agents flexibly adapt their behavior and interactions to dynamics in the system or its environment. In the course of designing and building various MAS applications, we derived a number of architectural patterns that provide generic solution schemes for recurring design problems. Together, this set of interconnected patterns makes up a pattern language.

To further mature the domain of software architecture, Shaw and Clements argue for the creation of reference materials that give engineers access to the field's systematic knowledge [2]. This paper contributes with a pattern language for MAS. Our objective is document well-proven design expertise for a family of self-adaptive software systems. A discussion of a methodology to apply the patterns is out of scope of this paper. However, we illustrate the patters with excerpts from an industrial case study in which we have applied the pattern language.

The paper is structured as follows. In Sect. 2 we explain the target domain of the pattern language. Sect. 3 gives an overview of the pattern language and presents the pattern template we use to document the patterns. Sect. 4 introduces the case study that we use to illustrate the patterns. In Sect. 5 we present the various patterns of the pattern language, and we give pointers to related work from the MAS domain. Sect. 6 draws conclusions.

2. Target Domain

The pattern language embodies the architectural knowledge we gained from the design and development of various MAS applications. We extensively used the Packet–World, a simple robotic application, as a study case for investigation and experimentation. We derived expertise from the design and development of a distributed peer-to-peer file sharing system. Our focus in this application was on coordination mechanisms inspired by principles of social ants. We have applied MAS in several experimental robotic applications focussing on the roles agents play to set up collaborations. We have employed a decentralized MAS architecture in an industrial transportation system for controlling automatic guided vehicles. We elaborate on this application in section 4. Recently we have been using agents in an intelligent transportation system for monitoring traffic jams. The focus here is on agents that set up organizations that dynamically adapt based on the changing context in which the agents are situated. For technical descriptions of these applications, we refer the interested reader to [3].

The key characteristics and requirements shared by these systems define the target domain for the family of software systems that are supported by the pattern language for MAS:

1) The software systems are subject to highly dynamic and changing operating conditions (such as dynamically changing workloads and variations in availability of resources and services) and are expected to manage the dynamics and changes autonomously.
2) Activity in the systems is inherently localized, i.e. global control or access to resources is difficult to achieve or even infeasible.

3) Important stakeholder requirements are flexibility (adapt to variable operating conditions) and openness (cope with parts that come and go during execution).

Typical domains are robotics, mobile and ad-hoc networks, and automated transportation systems.

3. Overview of the Pattern Language

Fig. 1 shows an overview of the pattern language with the relationships between the patterns.

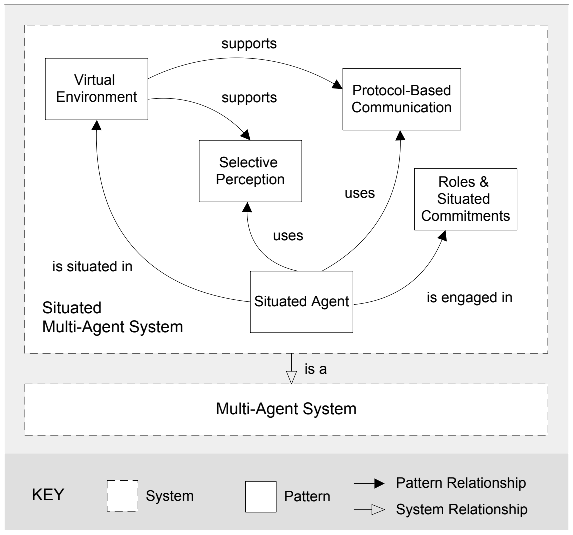

Figure 1. Overview of the pattern language

Situated MAS is one family of MAS. The focus of situated MAS is on modularization of agent behavior, efficient decision making, and indirect coordination. This contrast with deliberative approaches that emphasize knowledge representation, rationality, planning, and direct communication [4].

The basic patterns of the pattern language are *Situated Agent* and *Virtual Environment*. A situated agent is an autonomous problem solving entity that encapsulates its state and controls its behavior. The responsibility of an agent is to realize the application specific goals it is assigned. Situated agents are cooperative entities that are able to flexibly adapt their behavior and interactions with changing conditions. Agents are situated in a virtual environment that maintains a virtualization of the relevant parts of the world and serves as a coordination medium for the agents, i.e. it mediates both the interactions among agents and the access to resources.

Selective Perception enables a situated agent to sense the virtual environment and update its knowledge about the world. *Protocol-based Communication* enables situated agents to exchange messages according to prescribed communication protocols, i.e. well-defined sequences of messages. *Roles & Situated Commitments* are social attitudes of situated agents. A role represents a coherent part of functionality of a situated agent in the context of a collaboration. A situated commitment defines an engagement among agents in a collaboration. A situated commitment affects the behavior of the agents involved in the commitment in favor of the roles the agents play in the collaboration.

Some of the patterns in the pattern language are optional. For example, for the design of agents that do not communicate by exchanging messages, the Protocol-based Communication pattern can be omitted. We elaborate on a number of options in the pattern language in Sect. 5.

Pattern Template. To document the patterns we use the following template:

1) The name of the pattern.
2) A primary presentation that shows the elements and their relationships in the pattern. We use component and connector models to describe the pattern's units of execution.
3) A description of the architectural elements with their specific responsibilities.
4) An element specification that rigorously specifies the elements and how they are used with one another. We use π-ADL [5], a formally founded architectural description language. Due to space constraints, we only provide fragments of the specification of the Virtual Environment and Situated Agent patterns. For a complete specification we refer to [6].
5) A rationale that motivates the design of the pattern.
6) Pointers to related patterns.

This pattern template is inspired by the approach for documenting architectural styles presented in [7].

4. Case Study

The case study we use to illustrate the patterns is an automated transportation system that we have developed in collaboration with Egemin[1], a producer of automated logistic systems. An automated transportation system consists of a number of automatic guided vehicles (AGVs) that transport loads in an industrial environment. Transports are typically generated by an enterprise resource planning system. The main functionalities that an AGV transportation system has to fulfill are assigning transport tasks to appropriate AGVs, routing the AGVs efficiently while avoiding collisions and deadlocks, and maintaining the AGVs' batteries.

An AGV transportation system has to deal with dynamic and changing operating conditions. The stream of transports that enter the system is typically irregular and unpredictable, AGVs can leave the system for maintenance, production machines may have variable waiting times, certain areas in the warehouse may be closed for maintenance services, etc.

Although central control in AGV transportation systems is feasible, the activity in the system is typically localized. Vehicles have to avoid collisions on crossroads, AGVs typically execute tasks in their neighborhood, etc.

1. http://www.egemin.com/

Traditionally, the AGVs systems deployed by Egemin are directly controlled by a central server. The server plans the schedule for the system as a whole, dispatches commands to the AGVs and continually polls their status. This results in reliable and predicable solutions and enables easy diagnosis of errors. However, a shift in user requirements challenges the centralized architecture. Customers increasingly request self-adapting systems, i.e. systems that are able to adapt their behavior with changing circumstances *autonomously*. Self-adaptation with respect to system dynamics translates to two specific quality requirements: flexibility and openness. Flexibility refers to the system's ability to deal with dynamic operating conditions autonomously. Openness refers to the system's ability to deal with AGVs leaving and entering the system autonomously. To deal with these new quality requirements, we developed a radically new architecture based on situated MAS. Applying a situated MAS opens perspectives to improve flexibility and openness of the system: the AGVs can adapt themselves to the current situation in their vicinity, order assignment is dynamic, and the system can deal autonomously with AGVs leaving and entering the system. Fig. 2 shows a high-level model of the system.

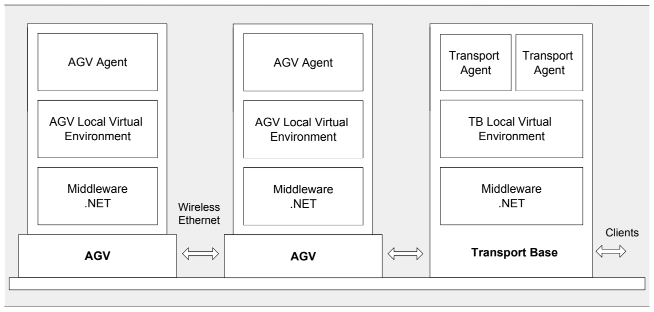

Figure 2. High-level model of a transportation system

Each vehicle is controlled by an AGV agent, and each transportation task in the system is represented by a transport agent that is deployed at the transport base. To realize the system requirements, AGV agents and transport agents have to coordinate, e.g. for transport assignment, for collision avoidance, etc. Therefore, the agents exploit a local virtual environment. The states of neighboring local virtual environments are synchronized using middleware services.

5. Pattern Language

Now we explain the patterns of the pattern language: virtual environment, situated agent, selective perception, roles & situated commitments, and protocol-based communication.

5.1. Virtual Environment

The primary presentation of the virtual environment pattern is shown in Fig. 3. In the case study, a virtual environment is deployed on each AGV and on the transport base.

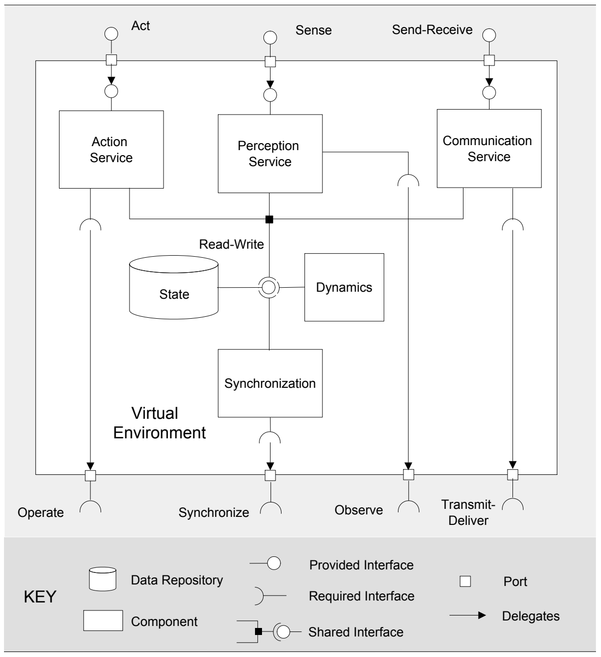

Figure 3. Primary presentation of virtual environment

5.1.1. Architectural Elements.

The **State** repository contains data that is shared between the components of the virtual environment. Data typically includes an abstract representation of external resources and additional state that is used for coordination purposes. An example of state related to external resources in the case study is the map of the warehouse. Examples of additional state are virtual marks situated on the map. Fig. 4 shows a fusion view[2] of the AGV local virtual environments of three AGVs. To avoid collisions, an AGV agent projects a *requested* operating space of its AGV in the AGV local virtual environment. This operating space represents the physical area the AGV intends to occupy. The AGV local virtual environment marks the operating space as *locked* when the conditions for the AGV are safe to move on.

Synchronization is responsible for synchronizing state of the virtual environment with state of particular external resources as well as state of the virtual environments on neighboring nodes. An example of synchronized state in the case study are the actual positions of AGVs. The synchronization component may pre-process the collected information before it updates the state of the virtual environment. An example is conflict resolution in case the requested operating spaces of two or more AGVs overlap. To resolve this conflict, the synchronization components of the involved nodes execute a distributed mutual exclusion protocol to decide which AGV gets priority. A typical way to collect

2. The view is generated on a remote machine that collects the state of the virtual environment of the AGVs via a wireless network and fuses the information into one image.

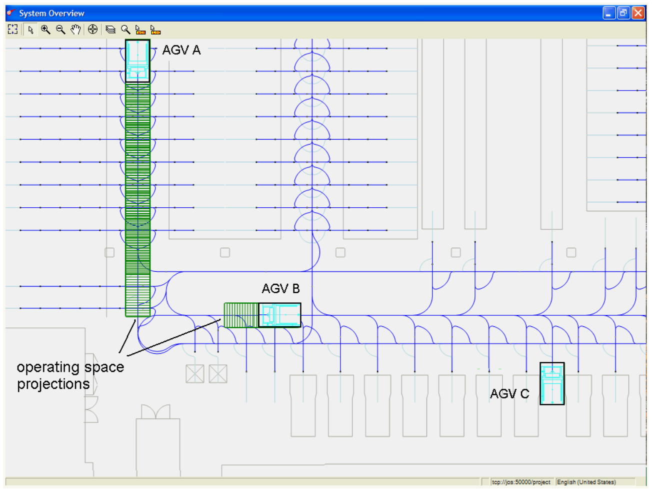

Figure 4. Operating space projections in the virtual environment

data in a distributed setting is by using suitable middleware. For the AGV transportation system, we used ObjectPlaces middleware that provides services to collect and maintain particular data of a set of nodes in a mobile environment (e.g. the position of all AGVs within a distance of 30 m of a node).

Dynamics contains processes in the virtual environment that happen independent of agents and external resources. The processes maintain the state of the virtual environment according to their application-specific definitions. In one of the approaches for transport assignment we developed for the case study, transport agents emit local computational fields in the virtual environment from the location of the transports. These computational fields are distributed data structures spread in the virtual environments which attract AGV agents. Each AGV agent of an idle AGV combines the sensed fields and follows the gradient of the combined field, guiding its AGV to a load [8]. In this case, the dynamics components of the local virtual environments are responsible for maintaining the fields.

The **Perception Service** provides the functionality to agents for sensing their neighborhood, resulting in a representation. A representation is a data structure that represents the sensed elements in a form that can be interpreted by the agent. The perception service supports *selective perception*. Selective perception enables an agent to direct its perception at the relevant aspects according to its current task. This facilitates better situation awareness and helps to keep processing of perceived data under control. To direct its perception an agent selects a set of *foci*. Each focus is characterized by a particular perceptibility, but may have other characteristics too, such as an operating range, a resolution, etc. Examples in the case study are a focus to observe the positions of nearby vehicles and a focus to observe the status of an operating space. When an agent invokes a sense request, the perception service collects the required information from the state repository of the virtual environment, or from external resources (via the Observe interface).

Action Service is responsible to deal with agents' actions. Actions can be divided in two classes: actions that modify state of the virtual environment and actions that modify the state of external resources. An example of the former is an agent that projects an operating space on the map in the AGV local virtual environment. An example of the latter is an AGV agent that commands the vehicle to pick a load. An action that modifies the state of the virtual environment may trigger the synchronization component to update the state of the virtual environment with the state of the virtual environments on other nodes. Action service can provide additional functions to translate actions related to external resources to low-level operations. E.g., a pick action is translated to low-level control commands that actuate the corresponding actuators.

The **Communication Service** is responsible for managing the communicative interactions among agents. It is responsible for collecting messages; it provides the necessary infrastructure to buffer messages, and it delivers messages to the appropriate agents. An agent communication message typically consists of a header with the message performative (inform, request, propose, etc.), followed by the subject of this performative, i.e. the content of the message that is described in a content language which is based on a shared ontology. The ontology defines a vocabulary of words that enables agents to refer unambiguously to concepts and relationships between concepts in the domain when exchanging messages. Such message descriptions enable a designer to express the communicative interactions independently of the underlying communication technology. To actually transmit the messages, the communication service makes use of a distributed communication system provided by underlying middleware. The communication service may provide specific services to enable the exchange of messages in a distributed setting, such as white and yellow page services.

5.1.2. Elements Specification.

We zoom in on the top-level specification of Virtual Environment and Communication Service.

```
value SituatedMultiagentSystem is abstraction ()
{
 type Message is view[
       ID : Integer,
       sender : String, receiver : String,
       performative : String, content : any];
 type StateItem is view[
       name : String, val : any];
 ...

 value VirtualEnvironment is abstraction ()
 {
  //external interfaces
  Send_Receive : Connection[Message];
  Transmit_Deliver : Connection[any];
  Sense : ...
  //connections among the components
  C_Read_Write : Connection[StateItem];

  //component composition
  compose
  {
    via CommunicationService send Void where {
    C_Read_Write renames Read_Write,
    Send_Receive renames Send_Receive,
    C_Transmit_Deliver renames Transmit_Deliver};
  and
    via State send Void where {
    C_Read_Write renames Read_Write};
  ...
  }
 }

 value CommunicationService is abstraction ()
 {
  Send_Receive : Connection[Message];
  Transmit_Deliver : Connection[any];
  Read_Write : Connection[StateItem];

  message_in, message_out : Message;
  state_item : StateItem;
  deliver_in, transmit_out : any;

  choose
  {
    //send message
    via Send_Receive receive message_out;
    unobservable;
    ...
    via Transmit_Deliver send transmit_out;
  or
    //deliver message
    via Transmit_Deliver receive deliver_in;
    ...
    unobservable;
    via Send_Receive send message_in;
  }
 }
 ...
}
```

The `Send_Receive` interface of the virtual environment provides, among others, operations for agents to exchange messages (type `Message`). The `Transmit_Deliver` interface makes use of underlying communication infrastructure. The concrete operations provided by this interface are application specific (type `any`). The component composition specifies how the various elements are inter-connected. `CommunicationService` provides functionality to send outgoing messages and deliver incoming messages.

5.1.3. Design Rationale.

The two primary principles that underly the design of the virtual environment pattern are the use of a shared data style and separation of concerns.

The shared data style results in low coupling among the components, improving modifiability and reuse. Low coupled elements do not require detailed knowledge about the internal structures and operations of the other elements.

By separating the various concerns (communication, perception, synchronization, etc.), the decomposition yields a flexible modularization that can be tailored to a broad family of application domains. E.g., for applications in which agents interact via marks in the virtual environment but do not communicate via message exchange, the communication service can be omitted. For applications in which there are no dynamic processes, the dynamics component can be omitted.

5.1.4. Related Patterns.

A variety of approaches have been developed related to the virtual environment pattern. Digital pheromones [9] and computational fields[10] are approaches which provide a virtualization of the underlying environment in which agents can manipulate marks to coordinate their behavior (similar to operation spaces in the case study).

Artifacts [11] is another related approach tailored to cognitive agents. An artifact is an abstract architectural building block that provides services to agents similar as the virtual environment. Among other approaches, we mention EASI (Environment as Active Support of Communication) [12], which proposes an architectural structure, similar to the virtual environment pattern. The focus of EASI is on communicative interactions between agents and the regulation of the scope of the interactions.

5.2. Situated Agent

The primary presentation of the situated agent pattern is shown in Fig. 5. In the case study, two types of situated agents are used: AGV agents which control vehicles and transport agents which manage tasks in the system.

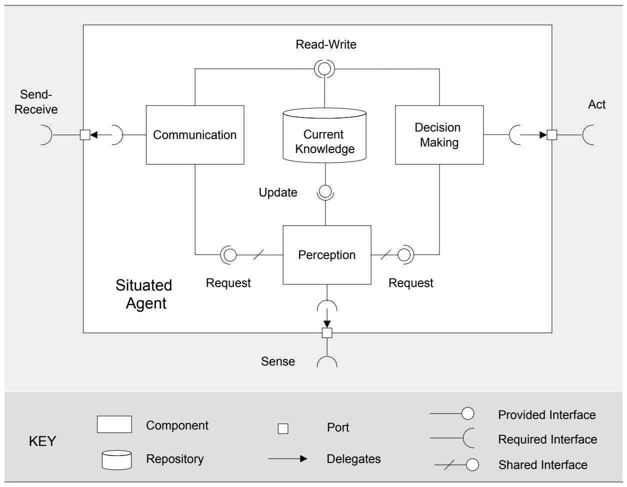

Figure 5. Primary presentation of situated agent

5.2.1. Architectural Elements.

The **Current Knowledge** repository contains state that is shared among the data accessors: Perception, Decision Making and Communication. State can be either static or dynamic. Static state refers to the agent's state of the system that does not change over time. An example in the case study is the identity of an agent. Dynamic state refers to the agent's current context. Examples are the position of the AGV and state about a commitment to transport a load.

Perception is responsible for collecting runtime information from the virtual environment. Perception requests are triggered by the communication component or the decision making component. A perception request includes a set of selected *foci* and a set of selected *filters*. The perception component uses the foci to sense the virtual environment for specific types of information. Perception processes the resulting representation with the selected filters before updating the agent's current knowledge. An example of a focus is the location of all transports in an area of 20 meters from the AGV position and an example of a filter is the location of the nearest transport. We elaborate on perception when we discuss the selective perception pattern.

Decision Making is responsible for realizing the agent's tasks by invoking actions in the virtual environment. Actions either intend to manipulate the state of the virtual environment or manipulate external resources. An example of the former is an AGV agent that draws an operating space in the local virtual environment, an example of the latter is an AGV agent that instructs the vehicle to move over a particular distance. We elaborate on decision making when we discuss the roles & situated commitments pattern.

Communication is responsible for communicative interactions with other agents. Message exchange enables agents to share information and set up collaborations. The communication component processes incoming messages, and produces outgoing messages according to well-defined communication protocols. A communication protocol specifies a set of possible sequences of messages. Communicative interactions are based on an communication language that defines the format of the messages and an ontology that defines a shared vocabulary of words agents use in messages. We elaborate on communication and give a concrete example of a protocol when we explain the protocol-based communication pattern.

5.2.2. Elements Specification. We limit the discussion to the specification of the structure of a situated agent and the way an agent senses the virtual environment.

```
value SituatedMultiagentSystem is abstraction ()
{
 type Focus is view[
    focus_name : String,
    focus_params : sequence[any]];
 type Filter is view[
    name : String,
    val_min : any, val_max : any];
 type Foci is sequence[Focus]];
 type Filters is sequence[Filter]];
 type PerceptionRequest is view[
    agent_id : String,
    foci: Foci, filters : Filters];
 type SenseRequest is view[
    agent_id : String, foci: Foci];
 type Representation is any;
 ...

 value SituatedAgent is abstraction ()
 {
   //external interfaces
   Sense_Request : Connection[SenseRequest];
   Sense_Result : Connection[Representation];
   Act : Connection[Action];
   Send_Receive : Connection[Message];

   //connections
   C_Request : Connection[PerceptionRequest];
   C_Read_Write : Connection[KnowledgeItem];
   C_Update : Connection[Knowledge];

   //component composition
   compose
   {
     via Perception send Void where {
     C_Request renames Request,
     C_Read_Write renames Read_Write,
     C_Update renames Update,
     Sense_Request renames Sense_Request,
     Sense_Result renames Sense_Result};
   and
     via CurrentKnowledge send Void where {
     C_Read_Write renames Read_Write,
     C_Update renames Update};
   ...
   }
 }

 value Perception is abstraction ()
 {
   Request : Connection[PerceptionRequest];
   Sense_Request : Connection[SenseRequest];
   Sense_Result : Connection[Representation];

   Read_Write : Connection[KnowledgeItem];
   Update : Connection[Knowledge];

   perception_request : PerceptionRequest;
   sense_request : SenseRequest;
   representation : Representation;
   knowledge_items : Knowledge;

   choose
   {
     //perception request
     via Request receive perception_request;
     unobservable;
     via Sense_Request send sense_request;
   or
     //knowledge update
     via Sense_Result receive representation;
     unobservable;
     via Update send knowledge_items;
   }
 }
 ...
}
```

The `Communication` and `DecisionMaking` components use a `PerceptionRequest` to request a perception. The request contains the agent's identity and a set of `Foci` (both are used to generate a `SenseRequest`) and a set of `Filters`. When the agent receives the `Representation` it is used to update the agent's current knowledge.

5.2.3. Design Rationale. In a situated MAS, control is divided among the agents. Situated agents manage the dynamic and changing operating conditions locally and autonomously. Both are important properties of the target applications of the pattern language. However, decentralized control implies a number of tradeoffs and limitations: (1) Decentralized control typically requires more communication. The performance of the system may be affected by the communication links between agents. (2) There is a trade-off between the performance of the system and its flexibility to handle disturbances. (3) Agents' decision making is based on local information only, which may lead to suboptimal system behavior. These tradeoffs and limitations should be kept in mind throughout the design and development of a situated MAS. Special attention should be payed to communication which could impose a major bottleneck.

The collaboration among the components of a situated agent contributes to the adaptability of the system. Decision making is responsible for selecting suitable actions. Communication is responsible for the communicative interactions with other agents. The separation of functionality allows both functions to act in parallel and at a different pace. In many applications, the time required for performing actions in the environment differs significantly from the time for communicating messages. E.g., when an AGV agent establishes a collaboration with a transport agent, the AGV agent drives the AGV to the load of that transport. However, if the transport agent meanwhile notices that another AGV becomes available that can handle the task more efficiently, it may abandon the initial collaboration and start a collaboration with the new AGV agent. Similarly, the AGV agent may switch its collaboration to a new transport agent that is more suitable. Reconsidering the coordination while agents perform actions, improves adaptability and efficiency.

5.2.4. Related Patterns. The initial work on architectures for situated agents mainly focused on agents' decision making, see for example the seminal work of Brooks [13] and Maes [14]. Ferber [15] extended the basic architecture with support for perception. The Situated Agent pattern builds upon this work and extends it with support for communication.

5.3. Selective Perception

The primary presentation of the selective perception pattern is shown in Fig. 6.

5.3.1. Architectural Elements.

Sensing takes a set of foci to produce a perception request that is passed to the virtual environment. As a result, the virtual environment produces a representation. We have explained the concepts of a focus and a representation in the discussion of the Virtual Environment pattern.

The **Descriptions** repository contains a set of *descriptions* to interpret representations. A description provides a template that specifies a particular pattern of a representation. E.g., consider a representation that represents a number of AGVs in a certain area. When the interpreting component interprets this representation it may use a description to interpret the group of AGVs as a cluster. This information can be useful to avoid traffic when selecting a path in the warehouse.

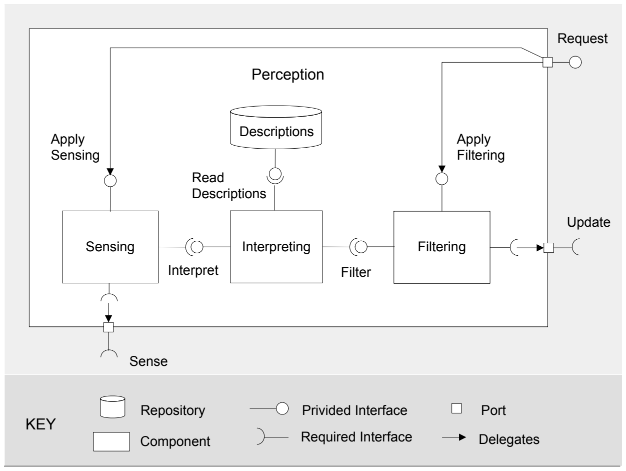

Figure 6. Primary presentation of selective perception

The **Interpreting** component uses the descriptions to extract a *percept* from the representation. A percept consists of data elements that describe elements sensed in the virtual environment or external resources in a form that can be used to update the current knowledge of the agent. Each match between the description template and the examined representation yields data of a percept.

The **Filtering** component filters a percept using set of selected *filters*. Filters allow the agent to select only those data elements of a percept that match specific selection criteria. Each filter imposes conditions on a percept that determine whether the data elements of the percept can pass the filter or not. We gave an example of a filter in the discussion of Situated Agent pattern. The Filtering component uses the filtered percept to update the agent's current knowledge.

5.3.2. Design Rationale. The components of perception collaborate in a pipe-and-filter like style. In this collaboration, each component provides a clear-cut functionality, while the coupling between the component is kept low. Foci, descriptions, and filters, are considered as first-class elements in the pattern. This helps to improve modifiability an reusability. The Interpreting component can be omitted in case the internal state of the agent and the observable state of the virtual environment are represented by the same data types.

Selective perception contributes to the adaptability of the system. By selecting an appropriate set of foci and filters, the agent directs its attention to the current aspects of its interest, and adapts it attention when the operating conditions change, contributing to the flexibility of the system.

5.3.3. Related Patterns. Explicit support for perception is often neglected in agent-based systems. One related approach which provides support for selective monitoring of the environment is CArtAgO [16]. CArtAgO introduces the abstraction of a virtual sensor. A virtual sensor allows an agent to sense the environment selectively based on particular properties similar to foci and filters. In EASI (see also section 5.1.4), the notion of a filter is used to regulate the scope of communicative interactions.

5.4. Roles & Situated Commitments

The primary presentation of the roles & situated commitments pattern is shown in Fig. 7.

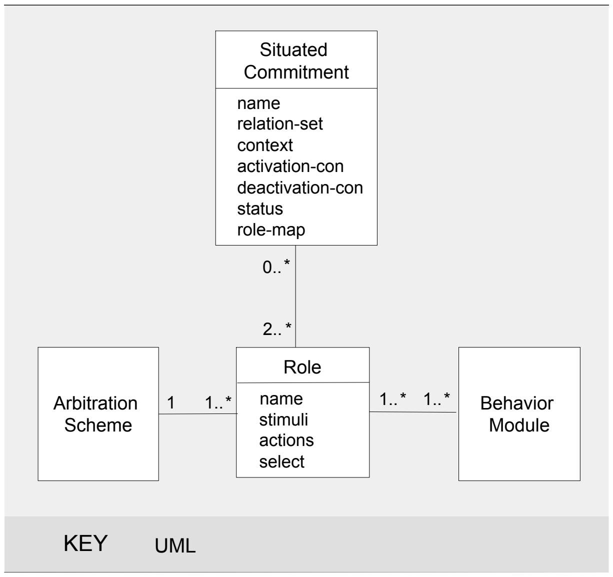

Figure 7. Primary presentation of the roles & situated commitment pattern

5.4.1. Architectural Elements. To select actions, a situated agent employs a behavior–based action selection mechanism. The main advantages of behavior–based action selection mechanisms are efficiency and flexibility to deal with dynamism in the environment. A behavior–based action selection mechanism consists of a set of behavior modules. Each behavior module is a relatively simple computation module that tightly couples sensing to action. An arbitration scheme controls which behavior-producing module has control and selects the next action of the agent. The roles and situated commitments pattern provides the means for situated agents to set up collaborations.

Role. Behavior modules that represent a coherent part of functionality in the context of an organization are denoted as a role. An organization consists of a group of agents that can play one or more roles and that work together. Roles are the building blocks for social organization of a MAS [17], provided that collaborations between situated agents are bounded to the context in which the agents are situated.

A role has a well-known *name* that is shared among agents in the system. *stimuli* are internal data or externally perceived data that affect the selection of actions of a role. Based on the actual stimuli, *select* determines the relative preferences for each of the possible *actions* that can be selected by the role. An arbitration schema uses the relative preferences for all actions of all the roles to determine which role has control and which action is selected for execution. Example roles of an AGV agent in the case study are `work`, `park`, and `charge`. Examples of stimuli of the `work` role are `transporting()` and `atLoad()`. Examples of actions are `drive(segment)` and `pick(load)`.

Situated Commitment. Collaborations are explicitly communicated cooperations reflected in mutual commitments. A situated commitment is defined in terms of the *roles* of the involved agents and the local *context* they are placed in. Agents agree on mutual commitments in a collaboration by exchanging messages (see the Protocol-Based Communication pattern). Once a collaboration is established, the mutual situated commitments will affect the selection of actions in favor of the agents' roles in the collaboration.

As for roles, situated commitments have a well-known *name*. Explicitly naming roles and commitments enables agents to set up collaborations, reflected in mutual commitments. The *relation-set* contains the identities of the related agents in the situated commitment. The *context* describes contextual properties of the situated commitment such as descriptions of objects in the local environment. *activation-con* and *deactivation-con* are the activation and deactivation conditions that determine the *status* of the situated commitment. When the activation condition becomes true, the situated commitment is activated, and the behavior of the agent will be biased according to the specification of the $rolemap$. The $rolemap$ specifies the relative weight of the preferences of the actions of different roles. In its simplest form, the $rolemap$ narrows the agent's action selection to actions in one particular role. An advanced example is a $rolemap$ that biases action selection towards the actions of one role relative to the preferences of the actions of a number of other roles of the agent. As soon as the deactivation condition becomes true, the situated commitment is deactivated and will no longer affect the behavior of the agent. An example of a situated commitment in the case study is the `working` commitment that commits an AGV agent to handle a particular transport. The relation set of the working commitment contains the identity of a transport agent. The context contains information about the transport (id, type, etc.). The commitment is activated when the vehicle picks the load, and deactivated when the load is dropped. In the working commitment, the AGV agent will select actions in the `work` role, transporting the load to its destination.

5.4.2. Design Rationale. Behavior–based action selection enables agents to behave according to the situation in the environment, and flexibly adapt their behavior with changing circumstances. The notions of a role and situated commitment enable agents to set up collaborations. Agents may explicitly communicate when the conditions for a committed cooperation no longer holds, or the local context in which the involved agents are placed may regulate the duration of the commitment. E.g., when an AGV runs out of energy, the `charging` commitment will be activated that guides the vehicle to a charging station. As soon as the battery is recharged, the commitment will be deactivated and the AGV will start looking for work. This approach fits the general principle of situatedness in situated MAS and improves flexibility and openness. An agent adapts its behavior when the conditions in the environment change or when agents enter or leave its scope of interaction. For an extensive discussion of behavior–based action selection with roles and situated commitments, we refer to [18].

5.4.3. Related Patterns.

Roles and commitments are generally acknowledged to be valuable abstractions for interaction design of MAS. However, their use for structuring interactions among situated agents is less well studied. For a recent overview, we refer the interested reader to [19].

5.5. Protocol-based Communication

The primary presentation of the protocol-based communication pattern is shown in Fig. 8.

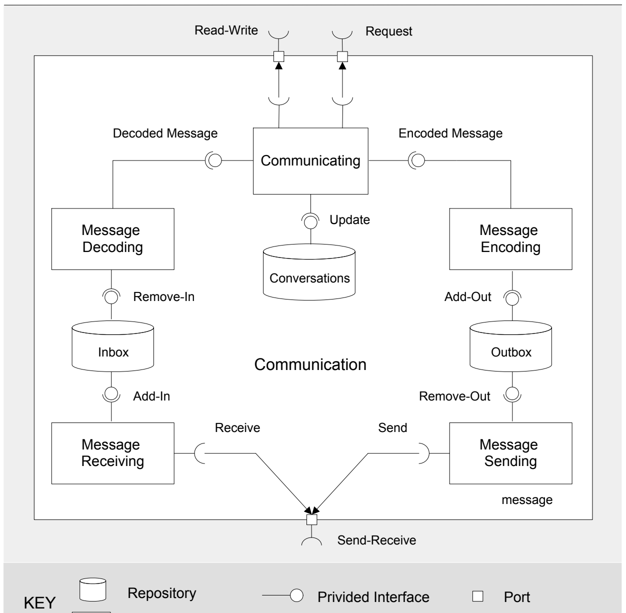

Figure 8. Protocol-based communication pattern

5.5.1. Architectural Elements.

The **Conversations** repository maintains a set of *conversations*. A conversation is an ongoing communicative interaction following a well-defined *communication protocols*. A communication protocol consists of a series of protocol steps. Each protocol step is characterized by a condition–effect pair. The condition determines whether the step is applicable. Conditions take into account the agent's current knowledge and data from ongoing communicative interactions. The effect is the actual result of executing the protocol step. A conversation is initiated by the initial message of a communication protocol. At each stage in the conversation there is a limited set of possible messages that can be exchanged. Terminal states determine when the conversation comes to an end.

An example of a communication protocol in the case study is the DynCNET protocol [8] which extends the well-known CNET protocol [20]. DynCNET consists of five basic steps: (1) the transport agent sends a call for proposals; (2) the AGV agents in a certain area respond with proposals; (3) the transport agent notifies the provisional winner; (4) while the AGV of the provisional winner moves towards the load, both the transport agent and the AGV agent may abort the provisional commitment if a more suitable assignment becomes available, and (5) the selected AGV agent informs the transport agent when it picks the load. DynCNET enables the agents to reconsider the situation in the environment and adapt the assignment of tasks when circumstances change, improving flexibility and openness. The tradeoff is an increase of required bandwidth.

The **Communicating** component provides a dual functionality: (1) it interprets decoded messages and reacts appropriately; (2) it initiates and continues a conversation when the necessary conditions hold. During the execution of a protocol step, the communicating component may initiate a perception request. The execution of a protocol step will produce the data to encode a new message and update the corresponding conversation. Furthermore, the agent's current knowledge may be modified, possibly affecting the agent's selection of actions. A typical example is the activation or deactivation of a situated commitment.

Outbox and **Inbox** are messages buffers. They buffer outgoing and incoming message respectively.

Message Encoding encodes a newly composed message. Message encoding is based on a shared communication language that defines the format of the messages, i.e. the subsequent fields the message is composed of. The message content is based on an ontology that defines a shared vocabulary of words that agents use to represent domain concepts and relationships between the concepts.

Message Sending selects a pending message from the outbox buffer and passes it to the communication service of the virtual environment. **Message Receiving** accepts messages from the communication service.

Message Decoding selects a received message from the inbox buffer and decodes the message according to the given communication language and ontology.

5.5.2. Design Rationale. Direct communication allows situated agents to exchange information and set up collaborations. Coordination through message exchange is complementary to indirect coordination via marks in the virtual environment. The various components in the communication component are assigned clear-cut responsibilities and coupling amongst components is kept low.

Communication defined in terms of protocols puts the focus of communication on the relationship between messages. In each step of a communicative interaction, conditions determine the agent's behavior in the conversation. Conditions not only depend on the status of the ongoing conversations and the content of received messages, but also on the actual conditions in the environment reflected in the agent's current knowledge, in particular the status of the agent's commitments. This contributes to the flexibility of the agent's behavior.

5.5.3. Related Patterns. A communication service for message transport between agents is generally considered as a basic service for MAS. The Foundation for Intelligent Physical Agents has defined a standardized reference model for an agent message transport service [21]. The Protocol-Based Communication pattern is compatible with this model.

6. Conclusions

In this paper, we presented a pattern language for MAS that supports the design of a family of decentralized self-adaptive systems. The pattern language ties five patterns together. Situated agent and virtual environment are the central patterns of the pattern language. Selective perception, roles & situated commitments, and protocol-based communication zoom in on the three main concerns of a situated agent. For each pattern, we provided a primary presentation that shows the constituent architectural elements of the pattern, a catalog that explains the responsibilities of the element, and a design rationale that explains the underlying design choices and the quality attributes associated with the pattern. In addition, we provided extracts of a formally founded specification of the pattern elements and how they are used with one another. We illustrated the patterns with examples of a decentralized control architecture for an industrial AGV transportation system, and referred to related work.

The presented pattern language integrates the design knowledge we have acquired from developing numerous MAS. We hope that this systematic knowledge contributes to a better understanding of MAS architectures and their value for designing complex self-adaptive systems.

Acknowledgement

This research is supported by the Foundation for Scientific Research in Flanders (FWO-Vlaanderen), the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and the Research Fund K.U.Leuven.

References

[1] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE '07: Future of Software Engineering*. IEEE Computer Society, 2007.

[2] M. Shaw and P. Clements, "The golden age of software architecture," *IEEE Software*, vol. 23, no. 2, pp. 31–39, 2006.

[3] D. Weyns, Homepage: http://www.cs.kuleuven.be/∼danny/.

[4] A. Rao and M. Georgeff, "BDI Agents: From Theory to Practice," in *1st International Conference on Multiagent Systems*. The MIT Press, 1995.

[5] F. Oquendo, "Pi-ADL: an Architecture Description Language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 3, pp. 1–14, 2004.

[6] D. Weyns, "Formal Specification of a Pattern Language for Multi-Agent Systems in Pi-ADL," 2009, http://www.cs.kuleuven.be/∼danny/pi-adl/.

[7] P. Clements and et al., *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2002.

[8] D. Weyns, N. Boucké, and T. Holvoet, "A field-based versus a protocol-based approach for adaptive task assignment," *Autonomous Agents and Multi-Agent Systems*, vol. 17, no. 2, pp. 288–319, 2008.

[9] S. Brueckner, *Return from the Ant, Synthetic Ecosystems for Manufacturing Control*. Ph.D Dissertation, Humboldt University, Berlin, Germany, 2000.

[10] M. Mamei and F. Zambonelli, *Field-based Coordination for Pervasive Multiagent Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[11] A. Omicini, A. Ricci, and M. Viroli, "Artifacts in the a&a meta-model for multi-agent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 17, no. 3, pp. 432–456, 2008.

[12] J. Saunier and F. Balbo, "Regulated multi-party communications and context awareness through the environment," *Multiagent Grid Syst.*, vol. 5, no. 1, pp. 75–91, 2009.

[13] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

[14] P. Maes, "Situated Agents can have Goals," *Designing Autonomous Agents, MIT Press*, 1990.

[15] J. Ferber, *An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[16] "CArtAgO," http://apice.unibo.it/xwiki/bin/view/CARTAGO/.

[17] J. Odell, H. V. D. Parunak, and M. Fleischer, "The Role of Roles," *Journal of Object Technology*, vol. 2, no. 1, 2003.

[18] D. Weyns, *Architecture-Based Design of Multi-Agent Systems*. Springer, 2009.

[19] V. Dignum, Ed., *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. Information Science Reference: Hershey, PA, USA, 2009.

[20] R. Smith, "The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver," *IEEE Transactions on Computers*, vol. 29, no. 12, 1980.

[21] FIPA, "Foundation for Intelligent Physical Agents," *http://www.fipa.org/*, (7/2009).