Agent-Oriented Software Patterns for Rapid and Affordable Robot Programming

Antonio Chella

Dipartimento di Ingegneria Informatica, University of Palermo, Italy

Massimo Cossentino

Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) - Consiglio Nazionale delle Ricerche (CNR), Palermo, Italy

Salvatore Gaglio

Dipartimento di Ingegneria Informatica, University of Palermo, Italy

Luca Sabatucci

Dipartimento di Ingegneria Informatica, University of Palermo, Italy

Valeria Seidita

Dipartimento di Ingegneria Informatica, University of Palermo, Italy

Abstract

Robotic systems are often quite complex to develop; they are huge, heavily constrained from the non-functional point of view and they implement challenging algorithms. The lack of integrated methods with reuse approaches leads robotic developers to reinvent the wheel each time a new project starts. This paper proposes to reuse the experience done when building robotic applications, by catching it into design patterns. These represent a general mean for (i) reusing proved solutions increasing the final quality, (ii) communicat-

Email addresses: chella@unipa.it (Antonio Chella), cossentino@pa.icar.cnr.it (Massimo Cossentino), gaglio@unipa.it (Salvatore Gaglio), sabatucci@dinfo.unipa.it (Luca Sabatucci), seidita@dinfo.unipa.it (Valeria Seidita)

ing the knowledge about a domain and (iii) reducing the development time and effort. Despite of this generality, the proposed repository of patterns is specific for multi-agent robotic systems. These patterns are documented by a set of design diagrams and the corresponding implementing code is obtained through a series of automatic transformations. Some patterns extracted from an existing and freely available repository are presented. The paper also discusses an experimental setup based on the construction of a complete robotic application obtained by composing some highly reusable patterns.

Key words: multi-agent systems; design patterns; pattern oriented design; robotics systems

---

1. Introduction

The process of building robotic systems is a complex task principally because these are intricate systems where different categories of problems have to be faced. A robotic system encapsulates algorithms that frequently derive from artificial intelligence and the architecture often includes distributed and heterogeneous components and must cope with real-time efficiency trade-offs. Designing a robotic architecture implies not only modelling the robot hardware and managing its sensors and actuators, but also modelling knowledge about the environment, and the ability to perform intelligent behaviours.

Software development for robotic systems is still today more an art than an engineering discipline. A few system developers have complete control all over the software, and typically write it all by themselves. There is an emerging demand for reuse techniques, with the twofold aim of maintaining software quality factors across projects [33, 44, 21] and of easily communicating and disseminating knowledge about robotic development issues [44]. The current state of the art in the development of robotic applications suffers from several problems:

- Robotic application variability makes hard to create ad-hoc standards, unified architectures and methods, as well as to profitably import them from other application domains [34, 33, 21].
- Responsibilities and boundaries among applications, frameworks and middleware are not universally defined [44]. Reuse of code across projects may easily fail if not complemented with design techniques.

- Several frameworks support the reuse of components, but a standard model to create robotic components still lacks because of the difficulty to find out a unified way to represent data and processes [33, 44]. Moreover, it is not clear the level of granularity to be used for building such components in order to promote the reuse across varying frameworks [34]. Therefore a documentation process would help the component integration process [33, 18, 21].

- Development time and resource limits, typically occurring for experimental robotic systems, demand for environments and tools for fast prototyping of applications and for verifying and testing the system [33, 18]. These tools would minimize the effort spent for secondary aspects of the system, like component integration or system documentation, and would maximize the effort for research objectives.

By now the agent paradigm seems to be one of the most interesting choices for developing a robotic application by following a rigorous design process [3, 21, 9]. Autonomous agents offer powerful instruments for decomposing, abstracting and organizing such complex, distributed and evolving systems. Several works consider robotic software as a collection of agents, where each of them is responsible for a specific functional area of the robot. These agents independently manage robot devices and collaborate in order to exhibit a collective synchronized behaviour, thus achieving a collective goal that is the robot mission.

This structure creates a decoupling between hardware and software, that is a necessary feature of an engineering design in which mission and global requirements, take priority over details about the implementation and deployment platforms.

This paper presents design patterns for agents, defined as a complement of the PASSI (Process for Agent Societies Specification and Implementation) design process [14] for developing multi-agent systems (MAS). The contribution of this paper is a pattern-based reuse method supported by a specific tool for automatic code and documentation production. Although other works exist in this field, the specific innovations proposed by this approach are multifold. First of all, they regard the integration between the pattern reuse practice and the PASSI design process; then there is the successful adoption in the development of complex systems like robotic ones. The generation of the system code from pattern reuse is another relevant element; this code is not the common skeleton produced by several design tools nor

the behavioural code obtained by applying transformations to dynamic diagrams (like it can be done for state-charts) but it is a complete and fully functional portion of code (skeletons and inner code of methods) reused from a repository and adapted to the specific problem or produced by processing available system specifications. Finally another contribution is in the definition of a repository of patterns that can be widely applied to the design of robotics systems but also in many other developing scenarios.

The paper is organized as follows: Section 2 presents common approaches for building robotic systems underlining the growing need for frameworks and methods for a rapid prototyping of these applications. Section 3 describes a well-known architecture adopted for a robotic case study throughout the paper. Section 4 illustrates the proposed engineering process consisting in a reuse technique based on design pattern composition. In addition, a tool (Agent Factory) is presented for supporting pattern selection, reuse and composition; it also provides automatic code and documentation generation. Section 5 presents some patterns from a repository for agents. They have been identified as a solution to typical and recurring robotic design problems. The section illustrates both pattern features and their usage. Section 6 is focused on the reuse and composition process applied to the proposed robotic application. Section 7 discusses the reusability of this approach and the quality of the produced system. Finally, some conclusions are drawn in Section 8.

2. Robot Programming Techniques and Methodologies

This section explores possible approaches from literature to the development of robotic applications. The analysis starts with specific architectures and frameworks for reusing robotic components. Successively some methodologies for designing multi-agent robotic systems are illustrated and finally design pattern reuse is discussed.

2.1. Component-Based Frameworks

In the last few years several different platforms have been proposed for robotics programming. These are mainly based on the principle of modularity and make an extensive use of component-based software engineering practices. Several frameworks exist where a set of components specialized for robotic applications can be customized and integrated, thus the process of building a robotic system is made easier.

The CLARAty framework [49] proposes a representation of the system based on two layers: the *Decision Layer* is the strategic level that drives the *Functional Layer*. The upper level provides components for the reasoning engine while the lower level is layered and can represent different levels of system abstractions.

The COOLBOT framework [21] explicitly considers software reuse, modular composition and third-party software integration. This framework provides means to design and to build units to be reused and to be composed (hierarchically and dynamically) by using finite state machine diagrams.

The Chimera Methodology [46] addresses the design of dynamically reconfigurable real-time systems and robotic applications. Components are specified by describing their interfaces. The result is a software model for objects that can be reused, statically integrated and dynamically reconfigured, it supports real-time applications, and it can be used in a distributed shared memory environment.

2.2. Multi-Agent System Methodologies

Multi-agent systems represent a means for introducing autonomy, distribution, collaboration, and other advanced features in robotic (and non-robotic) programming. Many design methodologies have been proposed for designing agent systems and most of them can be adopted for the design of robotic applications. In [47] authors propose to specify agent behaviour for robotic applications by using UML state-charts. Model checking techniques are employed to formally analyze behavioural properties of finite state systems and other issues like concurrency.

The Organizational-based Multiagent Systems Engineering (MaSE) methodology [19] has been conceived for engineering practical multiagent systems. It prescribes a top-down approach where the key concept is the *Goal*, a system-level objective that can be assigned to agents. MaSE has been used to design a team of autonomous, heterogeneous search and rescue robots. Analysis and design models proved to be helpful in the maintenance and modification of the cooperative robotic systems. A tool (agentTool) is provided with the methodology that supports the designer during system development.

The Cassiopeia methodology provides a method to proceed from a collective task global specification to the specification of the local behaviours, which are to be provided to the agents. The methodology has been successfully adopted in order to design and implement the organization of a robot team for the *RoboCup*.

A totally opposed approach is defined in ADELFE [7] that assumes agents totally ignore system goals and the environment where they live. It employs cooperative agents whose design is aimed at avoiding non-cooperative situations descending from incomprehension, ambiguity, incompetence, unproductiveness, concurrency or other conflicts. ADELFE was employed to implement a multi-robot resource transportation system [38].

2.3. Reuse with Design Patterns

It is commonly recognized that reuse cannot be limited to the development phase [4, 27, 28]. Design patterns are commonly considered the ultimate way for introducing reuse in a design process [31]. They also allow for overcoming main limitations of components reuse: (i) libraries of components usually address specific needs and lack of generality; (ii) people require information on how to use each component and how to correctly integrate them with the system under development; (iii) simply announcing the existence of a library, a component, or a framework, will not cause their usage; people requires trusting it.

The Tropos methodology [25] gives a great emphasis to early requirements analysis by adopting the Eric Yu's i* modelling framework [51]. This methodology includes a native support for design patterns reuse [20]. In Tropos, social patterns are idioms inspired by social and intentional characteristics used to design the details of a system architecture. The process is supported by a code generator that helps the programmer in choosing the right pattern, and then it generates the multi-agent system skeleton according to the employed patterns. The repository of patterns is quite generic and does not comprise robotic specific patterns.

Gaia [50] is another methodology for multi-agent system that includes a catalogue of design patterns, mainly focused on the social perspective [26]. This methodology uses roles as a key mechanism for social interoperability. Authors address that robots can use social roles to enable behaviour and to adapt to new domains. Roles are used as a concrete tool to assign and allocate behaviours, showing that a role-based system can effectively administer behaviour in an artificial system.

Agent based systems present great advantages in terms of level of abstraction, design methodologies and reuse frameworks. Next section introduces the architecture we adopted in several robotic applications. This architecture is perfectly suitable to be implemented by a multi-agent system and following

sections will show how the proposed reuse approach supports and simplifies the whole development process.

3. The Adopted Robotic Architecture

Robotic applications are complex systems affected by different categories of problems. Robotic systems encapsulate algorithms from artificial intelligence working on distributed architectures where components are developed to face real-time efficiency trade-offs.

The architecture adopted in this work [9][8] revealed to be highly generic, since it has been used across several heterogeneous robotic applications. The architecture is based on symbolic and behavioural processing of data coming from robot sensors. The integration of these two types of data processing is realized in three levels of data representation: (i) sub-conceptual, (ii) conceptual and (iii) linguistic. The subconceptual level is a repository containing reactive behaviour modules that are generally connected to the actuators of the robot. These modules process data coming from sensors and they send results to the conceptual space that is a metric space composed by a certain number of cognitive dimensions corresponding to environment qualities. Finally, the linguistic level acts as a central engagement module that controls the whole behaviour of the robot. The control of the robot is driven by the generation of expectations about relevant aspects of the environment. Expectations trigger actions and decisions, thus determining the rationale and reactive behaviour of the robot.

Autonomous agents represent a powerful instrument for decomposing, abstracting and organizing such complex, distributed and evolving systems. In this view, the robot may be considered as a collection of agents, where each one is responsible for specific functionalities:

- Agents in the sub-conceptual level access to robot sensors and actuators and realize reactive behaviours. They also collaborate with agents in the conceptual level, providing data to process.

- Agents in the conceptual organization create a structured representation of the surrounding environment.

- Agents in the linguistic level act as a reasoning engine, by using allowable information to generate expectations that drive the actions of the robot, thus determining its behavioural strategy.

From a software engineering point of view, this architecture maintains a decoupling between hardware and software, that is a necessary feature of an engineering process in which the mission of the system, or the global requirements, take priority over details about the implementation and deployment platforms. This architecture also enables to change robot configurations without modifying high-level components that handle the behaviour.

3.1. Case Study: A Robotic Guide

This section presents the case study that will be adopted throughout this paper: the architecture of the CiceRobot project, whose mission was performing guided tours of the Agrigento's Regional Archaeological Museum [32, 11].

The whole system is a project of considerable size whose complete description will be omitted because of space concerns. The paper will only detail a specific functionality of this robotic system: the indoor motion planning. The path-planning problem consists in the identification of the trajectory the robot has to follow in order to reach a desired position while operating in an environment whose map is a-priori partially given and it should be completed by exploration. The general architecture of such a robotic navigation system is based on the decomposition of the knowledge and reasoning process in three different levels: (i) Linguistic, working with symbolic representations of information, on which high level reasoning is possible (ii) Conceptual, working with geometrical knowledge, typically diagrammatic representation of an area and (iii) Sub-Conceptual, working with rawest representation of the environment around the robot [10, 8, 9]. The proposed architecture encompasses a society of agents in order to handle different aspects of robot management. An analogy has been created between knowledge decomposition and agent organization. Each knowledge level is assigned to one autonomous agent.

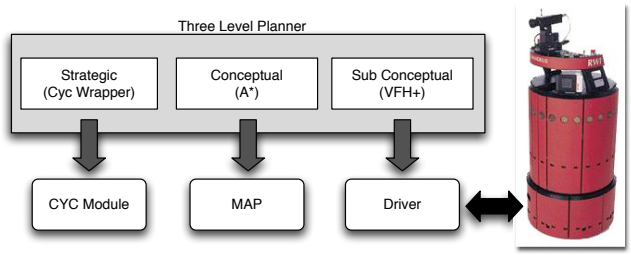


Figure 1: Architectural view of the robotic navigation system

Namely they are Strategic agent, Conceptual agent and Sub Conceptual agent.

The Strategic agent is responsible for reasoning on the symbolic information that represents robot's missions. This agent performs two high level tasks: (i) interacting with human users and (ii) operating strategic reasoning. The human user can assign qualitative targets, expressed in a natural language, to the robot. For instance a command may sound like "go to the left of Tom's desk". The strategic planner is the reasoning ability owned by this agent in order to elaborate a strategy for a single robot, or to coordinate the behaviour of a team of collaborating robots. From the implementing point of view, this agent contains an expert system that is able to reason on a huge ontology (provided by the OpenCyc reasoning engine) built over a natural language (common English).

The Conceptual agent manages a geometrical knowledge of the environment where the robot operates. This information is extracted from a 2D or 3D map that may be incomplete or that may contain errors. Knowledge about the environment is fixed during navigation by using sensorial data. The reasoning process on the map is enacted when a mission is elaborated by the Strategic agent: (i) the environment is decomposed in rooms and openings, then (ii) it is represented as a graph where rooms are nodes and openings are represented by arcs connecting them; finally (iii) an algorithm calculates the best navigational path in the graph in order to reach the target. A typical plan includes a list of the rooms to be navigated, and the openings to pass through. This plan does not care about details of in-room navigation.

The Sub Conceptual agent maintains a raw representation of the environment that surrounds the robot during the navigation. The motion space is described by using a Cartesian grid, where each cell stores a value for indicating the expectation of the presence of an obstacle at specific coordinates. This information is calculated on the basis of sensorial data, and the degree of confidence is due to noise and errors that typically affect this data. The agent is delegated to deal with this type of data and to elaborate a precise trajectory across the room by using a reactive run-time obstacle avoidance algorithm.

In the next section, the proposed approach for the development of robotic software with patterns will be discussed. The section will briefly introduce the PASSI process and then the definition and documentation style adopted for patterns.

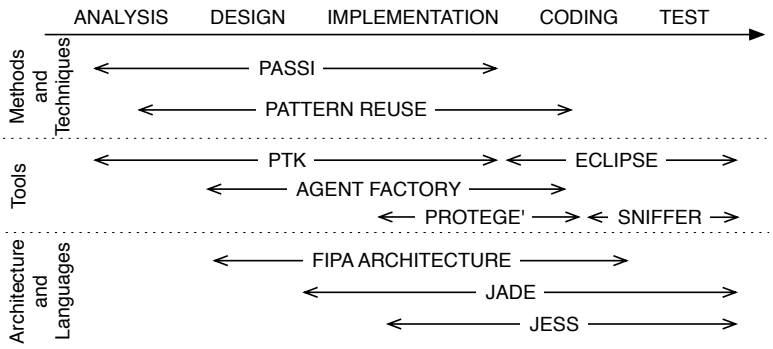


Figure 2: Overview of the framework for the development of robotic multi-agent systems.

4. Developing Agents with PASSI and Patterns

Designing a robotic architecture implies not only modelling the robot hardware and managing its sensorial outputs, but also modelling its knowledge about the environment, and providing the robot with the abilities required to perform intelligent behaviours.

Previous works (see Section 2) already addressed possible approaches for designing and developing multi-agent robotic applications by using agents. The approach that will be discussed in this paper is composed of a complete framework (Figure 2) that adopts the following components: (i) PASSI (Process for Agents Specification and Implementation) [14], a methodology specifically conceived for multi-agent systems design, (ii) PTK (PASSI ToolKit) a CASE tool that supports the design activity in PASSI, including the AgentFactory plug-in for design pattern reuse, (iii) the IEEE FIPA (Foundation for Intelligence Physical Agents) Abstract Architecture [22], providing a set of standard specifications for agent architecture, platforms and interaction protocols, (iv) the Protege'[36] tool for designing and implementing the system ontology and, finally, (v) Jade [6], a Java middleware providing a set of APIs for agent development and deployment easily integrable with (vi) Jess [1] a rule based engine for realizing the symbolic knowledge level.

All these elements provide a framework for developing multi-agent robotic systems from design to implementation. The framework is completed by a repository of design patterns for agents (that will be discussed in the next section), and a specific technique for pattern reuse during systems development. Design patterns encompass a steady way for introducing rapid prototyping of multi-agent systems in PASSI. Section 6 shows some benefits of the employ-

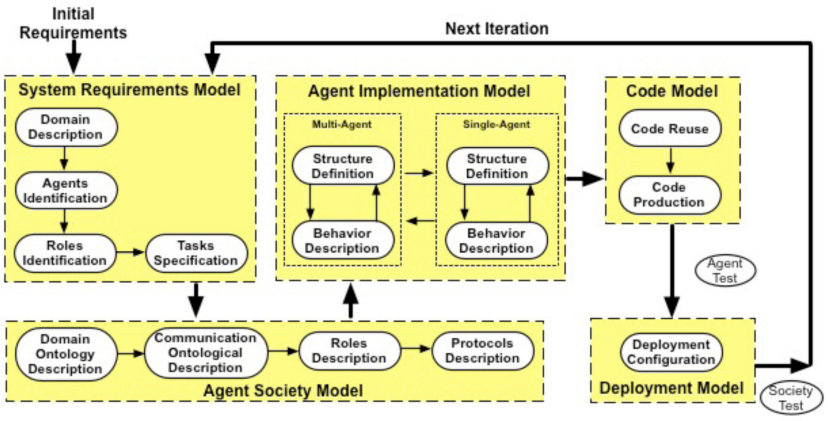


Figure 3: The PASSI design process life-cycle

ment of pattern reuse and composition in the development of a robotic application. In this approach design patterns are considered as building blocks, even if the difference with component integration must be clear; the reuse of each pattern enriches the system with new functionalities and specific issues. The resulting incremental development technique is the consequence of the mix of the top-down system functional decomposition (typical of the PASSI design process) and the bottom-up pattern reuse approach.

4.1. Design Patterns for Agents

Patterns for agents presented in this work are explicitly defined as an extension of the PASSI design process [14][15]. The PASSI methodology leads the designer from requirements analysis to the implementation of a multi-agent system (see Figure 3). The work is carried out through five phases composed by twelve sequential and iterative activities. The *System Requirements* phase produces a functional description of the system-to-be. The 'agent' and 'role' concepts are mainly used in order to realize a functional decomposition and an early assignment of responsibilities. The *Agent Society* phase analyzes the 'social' perspective of the system, thus focusing on interactions, ontology used to exchange information, and dependencies among agents and roles. The *Agent Implementation* phase expresses the solution architecture in terms of classes and objects by modelling the system from the static and dynamic perspectives. The *Code* phase is concerned with automatic code production (mainly from reused patterns) and manual code

completion. Finally, in the *Deployment* phase, agent deployment concerns find a solution.

As for traditional software design patterns, a pattern for agent is a proven solution to a recurring problem. The main difference is that pattern specific forces [24] are identified during the development of a multi-agent system, and the solution is described by using PASSI phases and models in terms of agents, responsibilities, roles, tasks, communications and so forth. According to the PASSI process, the formalization of patterns in the repository encompasses three layers: the first layer addresses problem description and it is based upon text (no formal language is used). This layer describes the specific problem where each pattern might be successfully employed and it aims at explaining how the pattern could help the designer in solving a problem while hiding the details required for implementing the solution. An interesting template for describing agent-oriented patterns is illustrated in [37], with the aim to improve the communication and the comprehension of patterns for agents by proposing a consistent template structure. The description of patterns in the proposed repository is done in terms of: i) motivations for pattern application, ii) context and forces that occur in the problem iii) system pre-requisites, and iv) post-conditions for the system where the pattern is instantiated.

The second layer contains the definition of the solution in terms of models of the multi-agent system. In this layer the description addresses the agent paradigm in order to describe how to solve a specific problem. A set of PASSI design models is generally employed to describe static and dynamic aspects of: (i) the involved agents or the agent organization, (ii) the roles played by agents, (iii) the services provided or exchanged by agents, (iv) the communications used for coordinating agent activities, and finally (v) the ontology used to store agent knowledge or to enrich communicative acts with specific meanings.

Different PASSI design activities and diagrams are affected by the peculiarities of the instantiated pattern. Figure 4 reports the list of diagrams affected by the instantiation of a few patterns. It is worth noting that some diagrams are automatically compiled by the PASSI design tools thus improving design quickness and ensuring a high design consistence.

The last layer of pattern formalization involves the implementation phase and it represents the detailed description of the solution in terms of implementation elements. This layer actually depends on the specific programming language or agent platform selected for the development. For this reason it

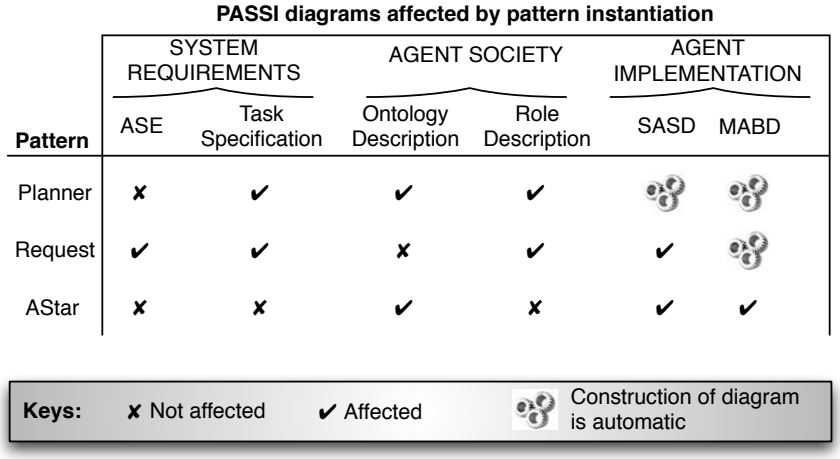


Figure 4: The PASSI diagrams affected by pattern instantiation

is maintained quite generic, thus avoiding to bind the methodology to implementation details. The implementation description is expressed in terms of: i) the static structure of each agent specifying tasks, messages, services and actions and ii) the dynamic behaviour of the whole system described by flows of activities, and interactions involving two or more agents.

Each pattern provides a set of functionalities and responsibilities that can be applied to the system in an isolated fashion, or more profitably, composed with: (i) elements of the system where the pattern is instantiated and/or (ii) functionalities and responsibilities coming from other patterns. This composition is a bottom-up process where design patterns represent the building blocks to be assembled in order to incrementally develop the system. An example of pattern composition will be discussed in section 6.

An example of design pattern description is reported in the following (design diagrams and example code are omitted for the sake of brevity but they are available on the repository website[1]):

Name: Planner
Classification: organization/behaviour
Intent: This pattern is originated from robotic and artificial intelligence applications. A planner system is a complex software that is able to perform some kind of reasoning in order to build a plan for reaching a target.

[1]http://www.dinfo.unipa.it/sabatucci/pattern/

Motivation: Consider a MAS (Multi-Agent System) designed for supporting a robotic system, such as a *RoboCup* soccer team. In this scenario, each robot of the team has to be fully autonomous and independent, and a strong coordination between team members is required. Thus, several algorithms and architectures are involved, regarding communications, environment exploration, information sharing, and mission planning. In a cooperative robotic context, each robot has a specific mission, which contributes to the realization of the common goal. Several planning architectures can be taken into account for this purpose. This pattern implements a distributed planner architecture based on the decomposition of the reasoning process on three or more levels.
Applicability: This pattern can be adopted only within an agent execution platform that provides a Yellow Pages service. The *Planner* participant needs to register the service to Yellow Pages and allows the agent to receive service requests.
Agent Solution: The solution is described in terms of static structure and dynamic behaviour. For a complete description of the formal language used in the solution description see [43]. The list of participants, with their goals and responsibilities of this pattern includes:

- Planner Agent. The agent that is responsible for the planning activity.
- Upper Level Agent. The agent used for coordinating the execution of a mission. This agent is responsible to send a sequence of missions (targets to be reached) to the Planner Agent. Each mission requires the elaboration of a new plan.
- Lower Level Agent. Agent that executes atomic commands. This agent is the executor of the plan generated by the *Planner Agent*.
- Planner Task. Task assigned to the Planner Agent in order to elaborate a plan. It is responsible for using sensible information to elaborate a plan for reaching a target.

The dynamic description of the *Planner* pattern encompasses a PlanListener task that waits for a mission incoming from the Upper Level Agent. When a mission is received, the Planner Task elaborates a plan for reaching the imposed goal, details of the specific algorithm are not part of this pattern. Then the PlanExecutor task decomposes the plan in atomic commands to be executed and sends them to the Lower Level Agent. If the planning algorithm fails (e.g. a path is not found), the DeadlockCommunicator task informs the Upper Level Agent of this event (deadlock). In turn, when the deadlock message arrives from a lower level it means that this agent has not been able to execute the mission. In this case the agent executes the RePlanner Task that tries to elaborate an alternative solution.
Implementation Solution: This pattern is implemented by using a class. Because of the generality of the pattern, the Planner task is an abstract class that does not perform any action. A concrete task must be developed in order to implement the desired behaviour.
Related Patterns from the Repository: This pattern divides the planning operation among three different agents but it does not give any indication about the specific algorithm to be applied in each level. The *VFHPlanner* and the *AStarPlanner* are two suitable patterns for solving the problem at the second and third levels.

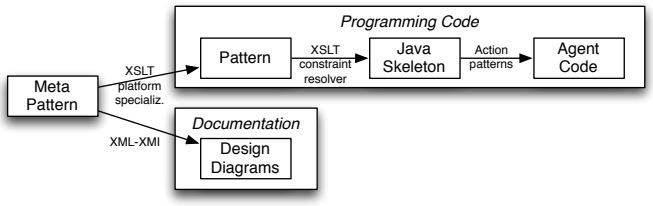


Figure 5: Representation of the pattern code/documentation generation process

Further details about dynamic aspects of this pattern are presented in subsection 5.1.1. The next subsection discusses the support provided to the designer by the AgentFactory tool for pattern reusing. This is a fundamental element of the proposed pattern reuse approach and it is responsible for pattern selection and retrieval from the repository, its introduction in the current system and finally code and documentation production.

4.2. Tool Support and Code Generation

The AgentFactory tool[2] [17] has been developed in order to support the use of patterns for agents during the design phases of PASSI. It provides a graphical interface for the selection of a pattern from the repository and it drives the whole process of pattern instantiation into the model. The tool also provides the: (i) automatic generation of the programming code deployable on the selected agent platforms (Jade [6] or FIPA-OS [39]), (ii) automatic documentation of the system portions obtained by using patterns, and (iii) reverse engineering of the agent source code with automatic identification of design patterns that have been previously instantiated in the system.

The Implementation Phase of the PASSI process exhibits a specific support for the development of multi-agent systems that are compliant to the FIPA Abstract Architecture specifications [22].

Figure 5 shows the code/documentation generation process [16], that is composed of several succeeding transformations. A language based on XML is used for describing generic patterns, or meta-patterns, where the solution is sketched by using a high level of abstraction, thus avoiding the specification of any implementation issue. When a meta-pattern is instantiated, it is processed by a succession of transformations. The first transformation is

[2]http://af.pa.icar.cnr.it/

conceived to specialize the meta-pattern for a specific agent platform (Jade or FIPA-OS). A different XSLT transformation is selected whether the system is to be developed by using Jade or FIPA-OS. The transformation adds all the details required by the specific implementing platform (for instance behavioural components at the class level are labelled tasks or behaviours according to the platform specifications); as a result, the meta-pattern is transformed into a pattern. The second XSLT transformation aims at generating the skeleton of the classes composing the architecture of the solution. This is an object-oriented (Java) code, where only attributes and operation interfaces are defined. Finally, the last transformation (that is not based on XSLT) introduces lines of code belonging to body methods inside class operations, thus producing a final code that is richer than the code generated by classical CASE tools. Such lines of code are stored in a repository of action patterns and are selected on the basis of the pattern that has been instantiated into the system.

An analogous transformation process, based on XSLT, is executed for generating the documentation for the pattern. The transformation generates the XMI description of the model that is shown by a viewer integrated in the tool. Documentation is provided as a set of UML diagrams representing portions of the system that are affected by the pattern instantiation. Figure 6 shows some screenshots, in particular a code view, a class diagram view and an activity diagram view, that are generated by the tool as a consequence of the *Planner* pattern instantiation.

5. Pattern Repository

The case study reported in this paper, a robotic application developed as a guide for the Agrigento's Regional Archaeological Museum, encompasses the architecture represented in Figure 1. During the development of this system, several patterns have been applied. The repository of design patterns for the PASSI methodology includes 21 patterns (listed in Figure 7). A complete description of all the patterns reported in our repository would require too much space and therefore we will only report their intent in Table 1; a complete description can be found in the repository website[3].

The list of the patterns that have been reused and composed for the navigation system described in Section 3 includes:

[3]http://www.dinfo.unipa.it/sabatucci/pattern/

Table 1: A short description of the patterns in the repository.

Pattern	**Intent**
Log Agent	A log agent is a social agent that owns the ability to store information about all interactions with other agents, for testing or administration aims.
Persistence Agent	A persistence agent pattern has the capability to restore information or data structures independently from the given architecture or platform, by means of memorization on non-volatile storage. Persistence allows an agent to save, load, freeze and thaw its state.
Memento Agent	As for the Persistence Pattern with the additional capability of recording agent belief base history in order to realize undo/redo functionalities.
Social Agent	A social agent owns the capability to use the yellow page service, when available from the platform, for registering/deregistering the services it offers to other agents and for searching services provided by other agents.
Resource Caching	When an agent owns the exclusive access to a resource, the caching mechanism can increase performances by minimizing the data exchange from/to the resource.
A* Planner	The A* Planner agent relies on a generic architecture for informed graph search by employing a heuristic estimation. The use of an ontology allows the customization of the architecture for a specific search context.
VFH Planner	The VFH Planner agent is an agent with a reactive layer that relies on the Vector Field Histogram method, based on a statistical representation of the environment, thus allowing the agent dealing with uncertainty.
Explorer	In a distributed open environment, a couple of base-explorer agents is able to collect data from 1..n remote platform(s) while maintaining the centrality of the reasoning process over this data.
Sequential Resource Share	A SequentialResource agent owns the exclusive access to a resource and provides services related to this resource. The sequential mechanism implies that each service request corresponds to a resource access.
Parallel Resource Share	As for the SequentialResource agent, but the parallel access mechanism implies that resource access and service providing are asynchronous processes.
Planner	This multi-part architecture derives from robotic and artificial intelligence applications. A planner system is able to scatter the reasoning process across several interacting agents.
Query/ Request/ Inform	Social agent communications are ruled by interaction protocols in which intentions and data are communicated by the means of a standard set of speech-acts like Request, Query, Inform (each one originating a different interaction protocol).
Secure Query/Request/Inform	As for Query/Request/Inform protocol, but these communications ensure a protection layer that includes means for avoiding other agents or humans can intercept or modify exchanged messages.
Contract Net	In open societies agents require to contract the performance of their services. This protocol provides means for operating negotiations among an undefined number of agents.
Publish-Subscribe	In a distributed resource management system, the publish-subscribe protocol defines means for two agents to synchronize a portion of their knowledge.
Information Agent	An information agent is capable of accessing one or multiple, heterogeneous and distributed information sources, proactively maintaining relevant information or services on behalf of its human users, or other collaborating agents, at any time and anywhere.
Holonic Society	A holon is a model for an agent society inspired to biologic and social systems, where elements can be at the same time 'whole' and 'parts'. Holonic societies are dynamic hierarchical structures that share a common goal.
Supply Chain	A Supply Chain is the system of entities (organizations, resources, and so on) involved in the delivery of a product from the manufacturer to the consumer. This pattern provides a solution for managing a supply chain in a multi-agent environment.

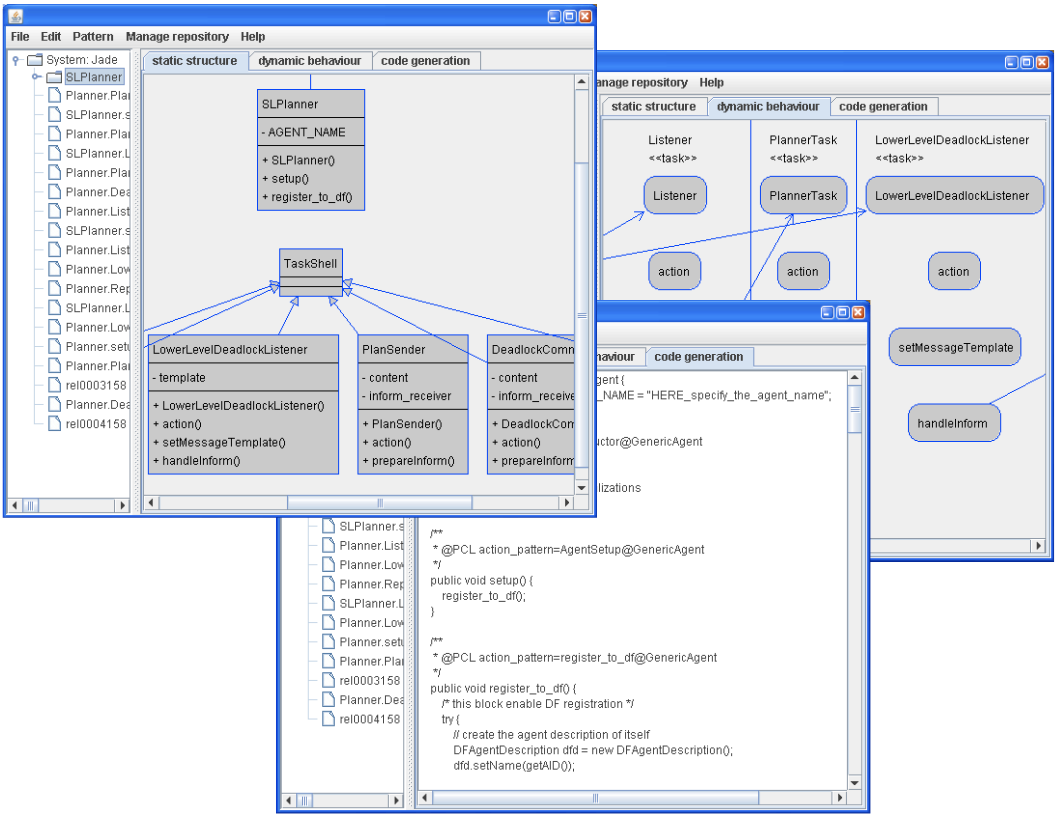


Figure 6: Screenshots of the AgentFactory application.

- *Planner* pattern; as already discussed in subsection 4.1, it encapsulates a generic intermediate level of the multi-level architecture for planning; this pattern may be applied to the three planning agents that have been employed in the system: the FirstLevel agent, the SecondLevel agent and the ThirdLevel agent.

- *AStarPlanner* pattern; it is used together with the *Planner* pattern in the second level of the architecture (SecondLevel agent) in order to implement the A* algorithm for informed graph searching [35]. This algorithm is used in order to search for the best trajectory among the rooms in the operating environment.

- *VFHPlanner* pattern; it may be used together with the *Planner* pattern in order to implement a reactive robot motion algorithm for avoiding

Pattern Repository		Application Context		
		Internal Architecture	Behavior	Role Definition
Functionality	Utility	LogAgent, Persistent-Agent, Memento	Secure-Request, Secure-Query	
	Knowledge Sharing	Generic-Agent	Explorer	Query, Inform-Protocol, Publish-Subscribe
	Resource Management	Resource-Caching	Sequential -Share, Parallel-Share	Information-Agent
	Organization	A*Planner, VFHPlanner	Planner	Request, ContractNet, Holonic-Society, SupplyChain

Figure 7: The list of patterns in the repository

obstacles inside each room by using the VFH* algorithm [30].

- *SequentialResouceSharing* pattern; it is used to assign the control of a resource to an agent. The agent is responsible to provide an indirect access to the resource by external requests. In the CiceRobot project this pattern is used to give the ThirdLevel agent the ability to access some robot drivers.

- *Request* pattern; it is used to implement a communication among two agents by using the FIPA Request agent interaction protocol [23]. This protocol is used to implement a service-based form of cooperation among agents.

The rest of this section illustrates these design patterns; planning patterns are discussed in details, whereas patterns for resource management and for agent interactions are only briefly introduced for space limits. A complete documentation can be found in the already cited website.

5.1. Design Patterns for Planning

Three patterns specifically address planning problems: (i) the *Planner*, (ii) the *AStarPlanner* and (iii) the *VFHPlanner*. This section discusses context, motivations and forces for each of these patterns, whereas in Section 6 an example of composition is reported.

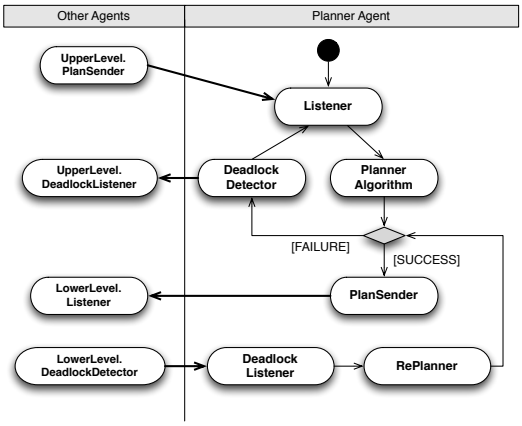


Figure 8: The behavior of the *Planner* pattern

5.1.1. The Planner pattern

This pattern proposes a general solution for generating a multi-agent architecture that is suitable for distributing a planning algorithm across several agents. The *Planner* pattern describes: (i) the structure of an intermediate level of a planning system, (ii) the internal flow of actions and (iii) the interactions occurring among the agents with other levels of the organization in order to coordinate their activities and to achieve the planning goal.

The proposed solution is conceived for a single agent whose dynamic description is reported in Figure 8, by using a PASSI *Task Specification* diagram. This is an UML activity diagram where the right swimlane focuses on the agent's tasks (represented as activities) and the left swimlane reports other tasks belonging to agents which interact with the focused agent. The right swimlane is used to represent the flow of actions of the Planner Agent, whereas the right swimlane is used to show interactions with the other levels of the planning architecture (Upper Level Agent and Lower Level Agent agents).

The *Planner* pattern prescribes that the agent receives (from the Upper Level Agent) a mission to be accomplished by means of the Listener task. The PlannerAlgorithm is the first activity to be executed (shown in Figure 8). The pattern does not specify how to execute this task; this has been done in order to raise the generality of the solution and to allow its reuse in all the three planning levels.

The PlanSender task is responsible to execute the plan. This operation generally needs to orchestrate the capabilities of the Lower Level Agent, to ana-

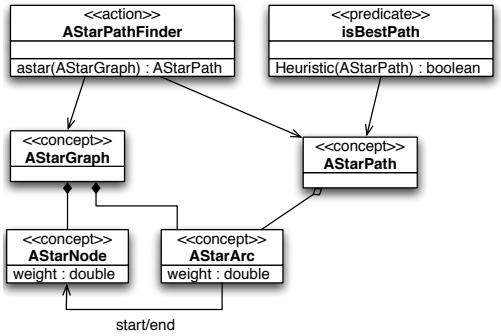


Figure 9: The ontology description of the AStarPlanner pattern

lyze the plan, and to communicate to the Lower Level Agent, step by step, each single sub-mission to be executed.

The architecture proposed in this pattern is fault tolerant. When the Lower Level Agent is not able to execute the assigned mission, a deadlock event occurs. In this case the DeadlockCommunicator task communicates to the Upper Level Agent a failure; in this situation the responsibility of the Upper Level Agent consists in deciding how to react to this exceptional event. An example is provided by a closed door along the path; this is discovered by the Sub Conceptual agent by using sensorial perception; as a consequence, it produces a deadlock signal to the Conceptual agent. This operation is achieved by using the Replanner task that elaborates an alternative solution.

Other benefits of this pattern are: i) decomposition of a complex planning algorithm in collaborating set of simpler algorithms; ii) possible distribution of the planning algorithm in a network with an improvement of the architecture scalability and efficiency; iii) independence of the proposed architecture from the specific algorithm, thus promoting the interchangeability, often required when building experimental portions of robotic applications.

5.1.2. The AStarPlanner pattern

An indoor environment can be represented as an unidirectional graph where nodes are rooms and arcs are the openings that connect rooms. As a consequence, the navigation problem can be seen as a graph exploration problem. In this case it could be useful to work with informed search algorithms because they reduce the searching space and generally offer solutions in a lower time. This type of algorithms needs some specific knowledge on the application context in order to improve the efficiency [35].

The *AStarPlanner* pattern is principally devoted to implement the A* algorithm. In order to abstract the algorithm from the specific application context (though remaining still reusable), Figure 9 reports the ontological description of the general data structure required for the execution. Ontology is described by using the *Domain Ontology Description* PASSI diagram, a class diagram where *Concepts*, *Predicates* and *Actions* are represented by using classes with specific stereotypes. The pattern implements the planning algorithm as a service that consists in elaborating an AStarPath by applying the A* algorithm on the AStarGraph (that is composed by instances of AStarNode and AStarArc). This structure contributes to maintain the pattern solution quite generic, so to be reused in different contexts. In the CiceRobot project AStarNodes are assigned to rooms of the building, whereas AStarArcs correspond to doors. Another example of application for this pattern is the search of routing-paths in a computer network, where AStarNodes are routers and AStarArcs are cables that physically connect routers.

From a structural point of view (see Fig. 8), the AStarPlanner pattern introduces two tasks in the agent where it is applied: the SetInitialConditions and the AStarPlanner. The first task is responsible for creating a match between pattern abstract concepts (nodes and arcs) and concrete elements that are specific of the domain where the agent should operate (for instance rooms and openings). This task should be customized each time the pattern is reused. The second task is concrete and implements the A* algorithm that works on nodes and arcs of a graph. The empty method named heuristic must be manually implemented in order to provide an ad-hoc metric for comparing different solutions discovered by the algorithm.

5.1.3. The VFHPlanner Pattern

The *VFH* (Vector Field Histogram) algorithm [30] is often used for real-time obstacle avoidance. It allows for the detection of unknown obstacles, by using heterogenous data coming from different kinds of sensors. The *VFHPlanner* pattern is an algorithmic pattern that encapsulates the VFH method. The robotic architecture, presented in Section 3, contains a reactive obstacle avoidance behaviour, enacted by the Sub Conceptual agent. This agent represents the operating environment by using a bi-dimensional grid that is constantly updated in according to sensor data perception. The VFH algorithm works through two consecutive steps of data reduction: (i) the first step consists in the generation of a polar histogram containing the obstacle density corresponding to a specific direction; (ii) the second step is the selec-

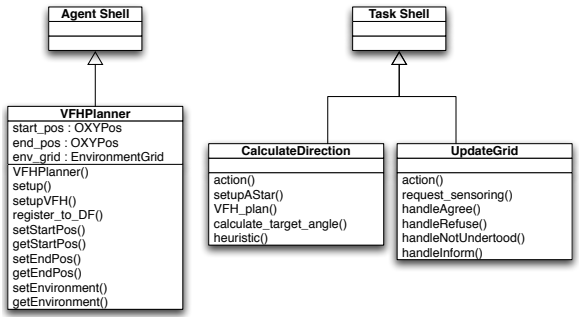


Figure 10: Static structure of the VFHPlanner pattern

tion of the sector presenting the minor probability of encountering obstacles, while maintaining stable (as much as it is possible) the direction to the final target.

The structural representation of the pattern is shown in Figure 10; the agent owns a set of attributes, the starting_pos, the ending_pos, and the env_grid, that respectively represent the robot position, the target position and the grid describing the environment around the robot. This grid contains information about free cells that can be navigated or cells that are occupied by obstacles.

This pattern comprises two tasks: (i) the UpdateGrid that cyclically interrogates sensors for updating the env_grid attribute, and (ii) the CalculateDirection that encapsulates the VFH algorithm for determining the best direction.

5.2. Information Patterns

The repository also comprises some patterns for managing information that is strictly related to external resources. Patterns in this category are named Information Patterns and they are: (i) the *SequentialShareResource* pattern, (ii) the *ParallelShareResource* pattern and (iii) the *CacheResource*. These patterns share common intents and motivations, but have different application contexts. The agent obtained by applying one of these patterns is capable of accessing heterogeneous and distributed resources and proactively maintaining relevant information on behalf of collaborating agents. Differences among these patterns are identifiable in the architecture used to provide the resource to other agents. The *SequentialShareResource* updates the resource status only when required, e.g. when the service is requested by another agent. The *ParallelShareResource* cyclically updates the resource status in order to maintain it always updated. The *CacheResource* pattern is generally used to decrease the number of accesses to a resource (both in

reading or writing). A typical case in which this behaviour proves useful is the access to a database.

5.3. Interaction patterns

Interaction is one of the most important aspects of multi-agent systems. Agents may communicate while collaborating in order to pursue the same goal, exchanging information, delegating responsibilities and so on. Interaction patterns are specifically conceived for implementing Agent Interactions Protocols (AIPs) as defined by FIPA [23]. A great number of the patterns in the repository use interaction patterns, in order to deal with communications.

Examples of interaction patterns are: the *Request*, the *Query* and the *ContractNet* patterns, that are frequently reused across projects. The *Request* pattern incorporates the FIPARequest protocol that allows an agent to request another one to perform some action. A typical scenario, useful to illustrate this pattern, is the delegation of a task. It occurs when an agent is not able or it does not possess the rights to perform an action by itself. In this case the agent may request another agent to perform the action. Because of the agent's autonomy, the participant agent can always refuse the request according to its personal intentions. The *Query* pattern is typically employed for exchanging information or believes about the state of a concept of the domain. The question is usually expressed by using a predicate. The *ContractNet* pattern provides a software solution for negotiation contexts. It is often used for e-market and supply-chain applications.

6. Pattern Reuse and Composition

This section illustrates an example of the process for building a system with pattern composition. The example is excerpted from the robotic system that has been described in Section 3.1 and uses the design patterns that have been illustrated in Section 5. The project aim was to realize a robotic system able to provide guided tours of the Agrigento's Regional Archaeological Museum [32, 11]. The whole system required an effort of about 24 person months.

6.1. Reuse in the PASSI Design Process

This subsection illustrates the outcome of reusing a single design pattern (the *Planner* pattern) along the design and development of the navigation

sub-system in the robotic application. This technique is provided (and integrated) with the PASSI design process, that is not discussed in details because of space limits. A complete description of the PASSI methodology can be found in [14, 15].

The identification of the patterns to be reused involves the early phases of the methodology, where the problem is decomposed in functional areas in order to understand and dominate it. The identification of these basic functionalities and the sketch of a high level agent-oriented architecture (as briefly described in Section 3) provide means for choosing patterns to be employed. In the case study, the reuse of the *Planner* pattern derives from the need to realize the three-levels architecture, where the decisional process is decomposed in three sub-processes according to different conceptualizations of the information about the environment. The design choice has been to assign each level of this architecture to a different agent thus creating a social organization responsible for handling different kinds of data and the algorithms working on them: (i) sensorial data coming from heterogeneous sources, (ii) diagrammatic representations of the environment and (iii) symbolic representation of the state of the world. The intent and the motivation provided by the *Planner* pattern candidate this latter as a good solution to this problem.

When reused, patterns for agents spread their effects in a wide part of the project. Once a pattern is identified and introduced in a specific diagram, the consequences of this pattern reuse are: (i) the current diagram is modified by pattern instantiations; (ii) changes also propagates forward on other parts of the design (an automatic support is provided by the tool in the code and documentation generation).

In the Conceptual agent, the *Planner* pattern introduces new elements in the design: (i) several *tasks* for dealing with specific data conceptualization, (ii) some new ontological *concepts* for handling plans, missions and commands, (iii) two *communications* for receiving the mission from the higher level agent and informing the lower level agent about the results of the planning activity and finally (iv) two other *communications* for managing exceptional and fault situations. All these details are shown in Figure 11.
In order to complete the design of this agent, some elements introduced by this pattern must be refined and integrated with existing elements in further phases of the methodology. For instance, the planner algorithm must be defined for dealing with a Cartesian map representing the environment, each communication must be detailed with a protocol and all generic concepts

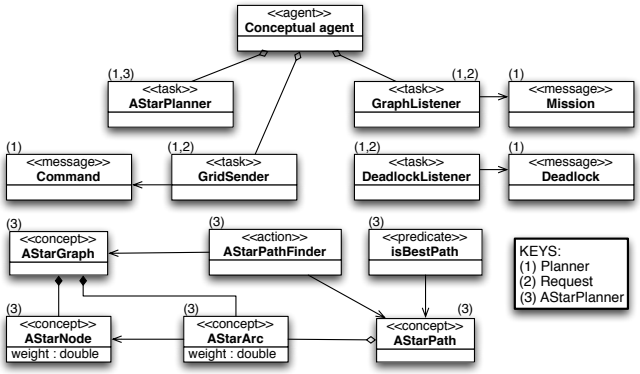


Figure 11: Single Agent Structure Diagram (SASD) for the SLPlanner agent. This PASSI diagram shows how patterns selected for this agent affect the final result. Numeric labels placed near classes indicate the application of more than one design patterns. When more patterns affect the same element, a composition is required.

of the ontology (like Mission) must be matched with concrete elements of the domain (Mission is, in the proposed case study, matched with a Target, a position in the map).

Figure 11 shows a static structure of the package describing Jade implementation classes for the Conceptual agent. Classes' compartments, detailing attributes and operations, were concealed in order to obtain a compact class diagram; numbers were attached to classes in order to indicate which pattern affects each specific element. As it can be seen, three patterns were employed in this agent, and several classes affected by more than one pattern. Specifically, the application of the *Planner* pattern in the Conceptual agent caused the introduction of a large part of the classes shown in Figure 11, and mainly, it imposed the internal architecture of this agent. It is worth to note that this diagram was automatically generated by our design tool (and successively annotated with numbers for the purpose of this paper).

6.2. Pattern Composition Technique

The PASSI diagram shown in Figure 11 implies that during the building process couple (or more) of patterns may affect the same piece of the system. When more patterns affect the same element of the system, a composition is occurring and a specific technique for handling this scenario can be useful. This technique ensures that a composition produces a perfect synergy of intents, and design conflicts are avoided. This issue is discussed in details in

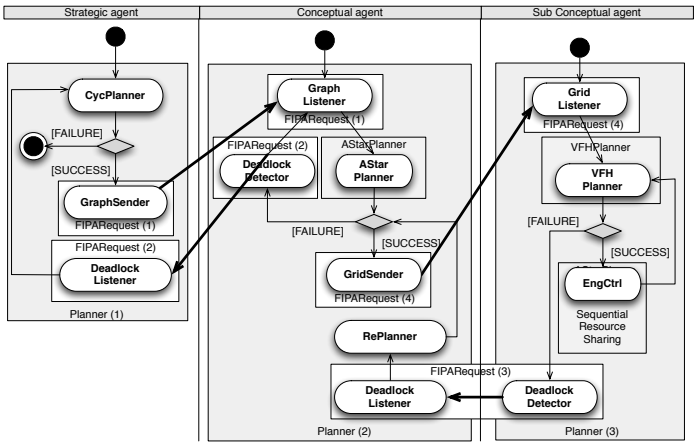


Figure 12: Multi Agent Behaviour Diagram (MABD) for the planning navigation subsystem.

[42] while here only a brief description of a pattern composition is provided to give a complete vision of the approach.

More in details, the *Planner* pattern is composed with more instances of the *Request* pattern. The intent of these compositions is to let the Second Level agent to receive messages from the First Level agent by using the FIPA Request interaction protocol.

The unification of roles provided by the two patterns is the rationale of the composition. The *Planner* pattern provides three roles: (i) the Upper Level role, (ii) the Planner Agent role and (iii) the Lower Level role. These roles are detailed in Section 5.1.1. On the other hand the *Request* pattern provides other two roles: (i) the Initiator role, played by the agent responsible to start a communication and (ii) the Participant role, played by the agent who desires to participate in the communication. The Strategic agent plays both the Upper Level role (from *Planner* pattern) and the Initiator role (from *Request* pattern), whereas the Conceptual agent plays the Planner Agent role (from *Planner* pattern) and the Participant role (from *Request* pattern), thus two couple of roles are unified: (a) Upper Level+Initiator and (b) Planner Agent+Participant.

The result of composition (a) is that the Strategic agent structure is enriched with a Request Initiator task (instantiated as Graph Sender) that is conceived to initiate a Set Mission communication by sending a Mission message containing the definition of the goal to accomplish (this is an abstract ontology element that must be manually refined by the designer).

The result of composition (b) is that the Conceptual agent structure is enriched with a Request Participant task (instantiated as Graph Listener) that is conceived to handle incoming Set Mission communications. Therefore the task waits for Mission messages and it is able to extract the goal to accomplish from them. An additional task (instantiated as Decision) is responsible for evaluating whether to accept or refuse the mission. The Planner task is obtained by the unification of the Planner Algorithm (from the *Planner* pattern) and the Execute Service (from the *Request* pattern); this task is abstract, because it does not contain any algorithm to be executed. The Plan Sender task (namely Grid Sender) is added by the *Planner* pattern in order to execute the plan that is elaborated (actually the plan execution is delegated to the Sub Conceptual agent).

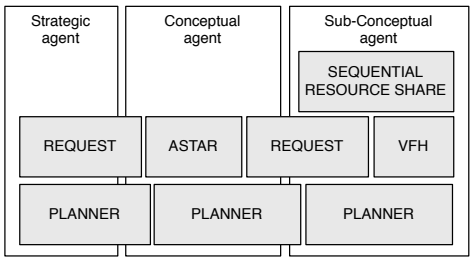
Another instance of the *Request* pattern is composed with the current structure in order to handle deadlock situations. The elements added to the two agents are the same of those previously described even if different names are given to avoid conflicts.

6.3. Incremental Assembling of the System

Previous subsections discussed the composition of only a couple of patterns and its consequences in the system. This subsection focuses on the construction of the navigation sub-system of the CiceRobot's application, where several patterns have been reused and composed. Figure 13 shows a couple of alternative ways for documenting pattern instances reused in the case study. The table on the left reports pattern (in columns) used for each agent (in rows). It is possible to have a double interpretation of this table: reading it by following rows, the table provides responsibilities assigned to each agent, coming from patterns; reading it column by column, the ta-

	Planner	**Request**	**Sequential Share**	**AStar**	**VFH**
Strategic agent	planner (1), upper level (2)	initiator (1)			
Conceptual agent	lower level (1), planner (2), upper level (3),	participant (1), iniziator (2)	client	planner task	
Sub-Conceptual agent	lower level (2), planner (3)	participant (2)	resource handler		planner task

a)



b)

Figure 13: Two alternative documentations of pattern instances reused in the navigation subsystem of the proposed case study.

ble provides how pattern instances influences the system. Figure 13.b is an abstract overview of pattern reuse and composition in three agents of the system. The three white boxes in the background represent the agents of the architecture. Patterns are shown as grey boxes that overlap with the agent boxes; these intersections represent the areas of influence of the patterns. Several patterns in this schema affect more than one agent at the same time. In particular the *Planner* pattern, responsible for defining the whole architecture, affects all the three agents (with three different instantiations).

Patterns have been applied to the project in an incremental way: the first step involves the *Planner* pattern, that is instantiated in all the three agents in order to realize the three-levels architecture. As a consequence these agents are organized in a simple social structure where the Strategic agent is the head and its mission consists in elaborating a high level strategy for accomplishing a user-defined mission; usually, this also involves calculating the target position in Cartesian coordinates (the user can indicate that by using a qualitative description that is converted in coordinates by also using the Open-Cyc inference engine). The Conceptual agent is responsible for planning a trajectory across an indoor environment map, and, finally, the lower level agent, Sub Conceptual agent, is a reactive agent responsible for managing robot motion and perceptions and at the same time performing obstacle avoiding behaviours along the trajectory.

A modification in the model is required in order to fit these pattern instances in the specific context. The internal architecture of the Strategic agent does not require the Listener and DeadlockCommunicator tasks (because this agent does not receive any command from an upper level agent). The architecture of the TLPlanner agent must be adapted too, since it does not require the PlanSender and the LowerLevelDeadlockListener tasks (this agent directly controls robot drivers).

At this stage, the base for the architecture of the navigation subsystem is defined but it is still incomplete: agents cannot (still) communicate and planning algorithms are not specified. Also, the Strategic agent, positioned at the linguistic level of the previously discussed robotic architecture, still needs the capability of understanding human language [32]. In order to accomplish this requirement, the CycWrapper component is now used to create a bridge between the Strategic agent and the OpenCyc knowledge base and inference engine employed in this project. The original off-the-shelf OpenCyc ontology has been extended with a set of custom rules and assertions concerning our specific domain in order to allow for common sense reasoning and qualitative

designation of our targets. CycWrapper is not a pattern since the adopted solution is greatly dependent on a specific technology (Cyc) and this violates the principle of generality of design patterns.

In the second level of the architecture, the *AStarPlanner* pattern is composed with the *Planner* pattern in order to implement the planning algorithm used for elaborating a path between two points of the map. Concepts included in the *AStarPlanner* pattern description (see Figure 9) are generic enough to be reused in several contexts. When the pattern is instantiated in our system, these concepts have to be manually related to the specific concepts of this domain: (i) AStarGraph is connected to EnvironmentMap, (ii) AStarNode is connected to Room and (iii) AStarArc is connected to Opening.

In the last level of the architecture the *VFHPlanner* pattern has been composed with the *Planner* pattern. This composition provides the implementation of the specific planning algorithm adopted at this level. This agent requires a reactive behaviour and therefore the VFH algorithm can be profitably used for that since it is well known and diffused. The only customisation work the designer has to do, consists in defining the heuristic function, in order to specify a metric for measuring distances in a cartesian space.

Another problem that can be tackled with the proposed patterns is the access to robot engines and sensors. They are usually considered an unshared resource that can be accessed with specific constraints. In order to manage multiple accesses, in our approach, the resource is assigned to the Sub Conceptual agent that is responsible for coordinating their use. The *SequentialResourceSharing* pattern is a good solution to provide access to a resource to the other agents of the system. This pattern encompasses two roles: the *Resource Manager* that handles a resource and the *Client* that requires to access to the resource. Once the pattern is assigned to the Sub Conceptual agent, this agent becomes responsible for managing all resource access requests, eventually using queues and flags.

Finally the subsystem is completed with the specification of communications among agents. The *Request* pattern is composed with the *Planner* pattern and with the *SequentialResourceSharing* pattern in order to allow agent communications by using the Request agent interaction protocol.

Figure 12 illustrates the dynamic behaviour that has been generated as a consequence of pattern composition and instantiation. The diagram is an activity diagram where swimlanes correspond to agents and activities represent agents' behaviours. As it happens in Figure 11, numbers address

elements (activities) that have been affected by pattern reuse. Activities in the three agents start in parallel. The Strategic agent selects the user-defined mission and it elaborates the coordinates to reach. These are assigned as a mission to accomplish to the Conceptual agent who is waiting for a command from the upper level. This agent has a pre-loaded environment map from which it generates a graph where rooms are edges and openings are arcs. This graph is explored by using the A* algorithm in order to select the best path to reach the target. The resulting path is decomposed in rooms to be crossed, and the navigation inside each room becomes a mission for the lowest level agent (performing obstacle avoidance behaviours). The third level agent is ready for new commands. Inside each room, the VFH planner works by using a 2D cell array that is filled up by using sensorial data; when an opening fitting the right movement direction is found, then the robot moves through that. If one of the three levels fails in elaborating the mission, a deadlock message is sent to the upper level, and a new planning activity is executed in order to discovery an alternative solution (if it exists).

7. Discussion

Literature agrees on the important role of design patterns and reuse techniques in the development of industrial applications [12, 13]. Design patterns affect several aspects of a system, introducing tangible benefits in the process. The main issue is the quality of produced code and documentation. The design of the system is well structured and greatly communicative because of the common language introduced by design patterns. Companies that introduce reuse techniques in their software development process notice a general improvement of the productivity.

These arguments have driven the definition of the reuse technique presented in Section 6. The technique is conceived for developing the system by reusing and composing a set of design patterns from a repository for agents. The reuse process considers design patterns as bottom-up bricks for building multi-agent systems. The current section discusses benefits coming from the employment of the whole development process and assesses the generality of the patterns in the repository and the proposed approach.

The quality of final code is easily recognized as a key aspect to estimate the goodness of the reuse technique, but its quantitative estimation is not trivial. It is generally measured by the minor density of errors recovered in

the source code. The tangible consequence of the use of design patterns in a project is the minor effort spent during the development process.

This section discusses three of the key aspects that are commonly considered to represent the quality of a reuse approach: (i) the rapidity and the affordability of the resulting development process, the (ii) traceability of the system and (iii) the reusability of patterns in the repository. These aspects will be detailed in the following subsections by referring to the case study reported in Section 3.1. The CiceRobot system has been conceived as a multi-agent system and it was designed by using the iterative and incremental phases of the PASSI methodology.

7.1. A rapid and affordable process

The use of a structured design process is traditionally considered inadequate to the robotic domain because modelling is a demanding task that only a few research groups really want to afford [21]. The effort required in modelling every aspect of the system is typically set against time and resource constraints [44]. The PASSI design process is composed of twelve activities that model the system from different abstraction levels and different points of view. However the methodology is provided with a reuse technique based on design patterns and a CASE tool that offers the execution of several automatic steps during the development (as discussed in Section 4). The reuse technique, discussed in this paper lets the developer easily build the system by using design patterns and a specific language is used to glue blocks together avoiding design conflicts. This speeds up the developer during his/her design work by enabling the reuse of a great part of (already tested and therefore affordable) design pieces thus (consequently) improving the correctness of the final result.

Two experiments have been conducted for evaluating the worthiness of the pattern reuse approach, against a traditional development lifecycle in which reuse is not adopted. This benchmark does not replicate an experiment for comparing the correctness of the resulting code with and without design patterns since similar works already exist and are independent from the design process or the reuse framework. Prechelt et al. [41] conducted an empirical study proving that, usually, the development of a system is completed with fewer errors if patterns are included in the process.

Several authors in literature assert that the use of design patterns in the development of a system improves the whole correctness of the result. Similar conclusions are provided by Vokac et al. [48] that proved the correctness of

a program developed with pattern reuse, by using regression models and an estimation method that took into account the correlations present in the raw data.

Table 2: Two experiments for evaluating the number of man/weeks saved by using the pattern reuse approach against the traditional approach.

Metronotte Project	**Manual** (p/w)	**Pattern Reuse** (p/w)	**saved %**
Requirement Analysis	4	4	0.00
Design (PASSI)	12	8	33.33
Development (architecture and interaction)	2	1	50.00
Development (ontology and algorithm)	6	5	16.67
Deployment and Testing	8	4	50.00
TOTAL	**32**	**22**	**31.25**

Our experiment is indeed aimed at comparing the number of man/weeks, as an estimation of the changes in the effort caused by the adoption of patterns. Table 2 summarizes results of the experiment, by showing different values for the time required in the various activities. It is worth noting that the pattern reuse approach implied a 30% of time less than the traditional development. Analyzing the single rows, it is possible to note that: (i) the reuse of pattern eases the design phase, in which many diagrams are filled in by the patterns; (ii) the development of the architecture is strongly affected by the reuse of patterns, because most of them concern architecture and interaction issues; (iii) a few differences exist for completing the development of the agents, and this is due to the automatic introduction of pieces of ontology; (iv) finally effort is considerably less for the deployment of the system and the testing phase.

In order to estimate the amount of automatic generated code, Table 3 reports reuse details for 6 agents and the system ontology. The number of interactions among these agents is high: 12 different types of communications are included. The table is organized in two sections, regarding the design and implementation phases. For the first section the columns report: (i)

Table 3: Summary of the agents involved in the CiceRobot project

Agent	Design		Implementation			
	n. pattern reused	n. diagrams generated	total LOCs	generated LOCs	manual LOCs	code reuse
Sensor Reader	2	3	105	87	18	82,86%
Sub-conceptual	5	8	486	258	228	53,09%
Engine Controller	2	4	314	143	171	45,54%
Conceptual	4	7	752	253	499	33,64%
Knowledge Manager	2	3	232	65	167	28,02%
Strategic	3	3	523	443	80	84,70%
Ontology	2	1	1458	587	871	40,26%

the number of instances of the employed patterns, and (ii) the number of diagrams reused for each agent.

Analyzing this data, it is worth to note that 12 design patterns were adopted for building the three-level architecture (the FirstLevel, SecondLevel and ThirdLevel agents). The consequent number of automatically generated code lines becomes even more relevant when considering that more instances of the same pattern are used in the same agent.

The code is generated by the Agent Factory application and it is comprehensive of attributes, operation interfaces and body methods (see Figure 14).

The rightmost part of Table 3 details the implementation phase, reporting (i) the total number of LOCs, (ii) the number of LOCs generated by the tool, (iii) the number of manually produced LOCs, and (iv) the percentage of automatically generated code with regards to the total amount of code (of each agent).

The size of the source code for this case study is about 3,8 MLOC for the 6 different types of agents. The contribution of automatically generated code to the development of these agents is relevant, spreading in a range from 28% to 84%, providing in total almost half of the total number of LOCs of the system.

The amount of generated code could vary according to the specific agent and its functionalities. For instance, the KnowledgeManager agent (responsible to store and manage indoor maps describing the environment) was built by using two patterns only and therefore it has a poor percentage of reuse. On the other hand the reuse percentage for the SensorReader agent is significantly high (82.25%). The motivation is that te structure and behaviour of this agent are simple and focused on handling and communicating acquired sensorial data. As a consequence, this agent has been mainly realized by composing and reusing interaction and information patterns.

7.2. Traceability and maintenance of the system

The quality improvement of the system and of the resulting documentation is an important outcome of adopting a design process and a reuse technique for modelling a robotic system.

Indeed, it is widely understood that the use of design patterns introduces many quality aspects into the system; among them, an increased productivity throughout almost all the design and development life-cycle, a better documentation [2] and an easier maintenance of the system in the future[5, 40].

```
public class SecondLevel extends Agent {
...
    public class Listener extends SimpleBehaviour {
        private MessageTemplate template;
        /**
         * @PCL action_pattern=constructor@InformParticipantTask
         */
        public Listener(Agent owner) {
            super(agent_ref);
            MessageTemplate m1 = MessageTemplate.
                  MatchPerformative(ACLMessage.INFORM);
            MessageTemplate m2 = MessageTemplate.MatchProtocol(
                  "inform-protocol");
            setMessageTemplate(MessageTemplate.and(m1, m2));
        }
        /**
         * @PCL action_pattern=setup@InformParticipantTask
         */
        public void action() {
            ACLMessage msg = myAgent.receive(template);
            if (msg != null) {
                handleInform(msg);
            }
        }
       ...
}
```

Figure 14: A portion of the automatically generated code for the SecondLevel agent

The use of patterns for the documentation of a system lets programmers handle new changes without having to understand system working details [29].

In the presented approach patterns are precisely documented by using models that adopt a precise notation (aligned with that adopted in the PASSI process). PASSI diagrams contained in pattern solutions can be integrated with the current model, thus raising the quality of the whole documentation with a minimal effort, thanks to the integration between Agent Factory and the PASSI ToolKit (PTK). Every design choice of the system is motivated by a diagram that illustrates an aspect or a functionality.

In addition pattern reuse is traced along the whole process. The pattern selection and integration with the system is traced by information like the one reported in Figure 13.a, from which it is possible to figure out the rationale for pattern applications into the model. In addition, the language used for the pattern specification and composition (see [42]) traces the synergy among patterns that affect the same slice of system. Finally, tool support is fundamental to forward the traceability of pattern reuse down to the programming code.

This precise documentation obtained by pattern reuse reduces the time to understand the whole system and to test or modify its components in future.

7.3. Reusability of the proposed patterns

Section 4.2 presented the transformation process for generating the programming code of a design pattern. By now the tool does not operate any integration control over patterns when they overlap; this activity is entirely demanded to the designer. As a consequence, the correctness of the whole code generated by the tool depends on the correctness of the design patterns employed in the project. It is widely understood that the quality of a pattern is assessed by the experience rather than by testing it [45], and in a similar fashion, the quality of automatically produced code is provided by the refinement and consolidation of design patterns in the repository across several years of work and across various low/mid-size projects. In the past few years the repository of patterns for agents has been tested on several heterogeneous domains. Some of the projects involved in this analysis are reported in Table 4. They are all academic mid-size case studies:

1. CiceRobot: the already discussed robotic application that aims at providing guided tours at the Agrigento's museum[11].
2. Metronotte Simulator: a system that emulates a surveillance B21r robot in a realistic indoor 3D environment.
3. Koala: a robotic navigation system for a small car robot, where vision is provided by an external camera positioned well above the operating field (bird-eye camera).
4. Meeting Scheduler: an agenda manager and meeting scheduler for a company workgroup.
5. Iron Manufacturer: an application for supporting a B2B scenario involving an iron manufacturing company.
6. Bike Production: an application for supporting a B2C scenario for a bike manufacturing company.
7. Exam Manager: an application for managing exam calendars and student enrolment.
8. SDBE Sim: a software system that simulates business evolution of regional small and mid-size companies.

Table 4 summarizes statistics of reuse for a collection of patterns from the repository, including those presented in this paper. The name of projects is reported in the first column, whereas other columns indicate the number of pattern instances in each project. The last column and the last row respectively report: (i) the whole number of reuse occurrences of a specific pattern

across all the projects, and (ii) the total number of patterns reused for each project. This table is intended to prove the generality of the proposed approach and of the pattern repository since it shows the extent of pattern reuse and composition across several heterogeneous applications.

Table 4 shows that the most reused patterns are the interaction ones (in particular the Request and the Query patterns are reused 74 times). This result is not surprising because of the key role played by communications in agent societies. They are often instantiated as isolate elements of reuse (26 instances), but most times they are composed with other ones (48 instances). Interaction patterns are very useful in combination with other patterns in order to provide collaborative abilities. The category of patterns for planning is strongly reused in robotic architectures (13 instances), but their usage is not limited to this domain: they may be employed in resource-based contexts. As an example the meeting scheduler application uses 3 instances of planning patterns for the meeting-scheduling task. To conclude, information patterns have been steadily reused across all the applications, showing that their reuse is independent from the specific domain. The management of resources is another common issue in the development of several kinds of applications. This category represents another good example of pattern generality thus proving that patterns in the proposed repository catch solutions in a really generic way that is easy to reuse in various cases.

Table 4: Summary of pattern reuse statistics related to some of the projects developed with the presented approach in the last years

PATTERN REUSE SUMMARY	SDBE Sim	CiceRobot	Metronotte Simulator	Koala	Meeting Scheduler	Iron Manufacturer	Bike Production	Exam Manager	TOT
AStarPanner		1	1		1				**3**
Parallel ShareResource						1		1	**2**
Planner		3	3	2	2				**10**
Query	3	4	2	2	6	7	5	6	**35**
Request	5	5	8	3	2	5	5	6	**39**
Sequential ShareResource	3	1	1			2	4	3	**14**
VFHPlanner		1	1	1					**3**
TOT	**11**	**15**	**16**	**7**	**11**	**15**	**14**	**16**	

8. Conclusions

This paper presented an approach for designing and building robotic systems by reusing and composing design patterns. Such patterns are proved solutions to recurrent problems occurring during the development phase of complex and distributed systems. The reuse technique is integrated with a comprehensive framework for developing multi agent systems that includes the PASSI methodology for designing the system, a set of tools to support the development and an agent platform for implementing and integrating the robotic agents.

The contributions of this paper lie in the integration between the pattern reuse practice and the PASSI design process, the (successful) adoption of a pattern reuse extensive practice in the development of robotic systems, the generation of relevant portions of system code (skeletons and inner code of methods), and finally the definition of a repository of patterns that have been tested in the design of robotics systems but may be fruitfully applied in many other developing scenarios.

Patterns in this repository can be instantiated as isolate elements of reuse in a project, but they revealed a greater usefulness when composed together in order to build a larger portion of the system to be. Finally a specific tool for automatic code generation is provided to designers to ease the agent development phase. This tool uses a meta-description of pattern solutions for generating, as a consequence of various transformations, the system code and the corresponding (although partial) documentation. A case study is discussed across sections of this work in order to illustrate the whole process of pattern reuse and code generation.

9. Acknowledgements

This work has been partially supported by the EU project FP7-Humanobs.

References

[1] Jess, the rule engine for the java platform. `http://herzberg.ca.sandia.gov/`, 2008.

[2] E. Agerbo and A. Cornils. How to preserve the benefits of design patterns. *SIGPLAN Not.*, 33(10):134–143, 1998.

[3] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An Architecture for Autonomy. *The International Journal of Robotics Research*, 17(4):315, 1998.

[4] B. Barnes and T. B. Bollinger. Making reuse cost effective. *Software*, 8(1):13–24, January 1991.

[5] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 139–149. Springer-Verlag London, UK, 1994.

[6] F. Bellifemine, A. Poggi, and G. Rimassa. Jade - a fipa2000 compliant agent development environment. In *Agents Fifth International Conference on Autonomous Agents (Agents 2001)*, Montreal, Canada, 2001.

[7] C. Bernon, V. Camps, M.-P. Gleizes, and G. Picard. Engineering adaptive multi-agent systems: the adelfe methodology. In *Agent Oriented Methodologies*, chapter VII, pages 172–202. Idea Group Publishing, 2005.

[8] A. Chella, A. Frixione, and S. Gaglio. A cognitive architecture for artificial vision. *Artificial Intelligence*, 98(1-2):73–111, 1997.

[9] A. Chella, A. Frixione, and S. Gaglio. An architecture for autonomous agents exploiting conceptual representations. *Robotics and Autonomous Systems*, 25:231–240, 1998.

[10] A. Chella, S. Gaglio, and R. Pirrone. Conceptual representations of actions for autonomous robots. *Robotics and Autonomous Systems*, 34:251–263, 2001.

[11] A. Chella, M. Liotta, and I. Macaluso. CiceRobot: a cognitive robot for interactive museum tours. *Industrial Robot: An International Journal*, 34(6):503–511, 2007.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] P. Coad, D. North, and M. Maryfield. *Object Models: Strategies, Patterns, and Application*. Yourdon Press, New Jersey, NJ, 2nd edition, 1997.

[14] M. Cossentino. From requirements to code with the passi methodology. In B. Henderson-Sellers and P. Giorgini, editors, *Agent-Oriented Methodologies*. Idea Group Inc., Hershey, PA, USA, 2005.

[15] M. Cossentino and L. Sabatucci. *Agent System Implementation in Agent-Based Manufacturing and Control Systems: New Agile Manufacturing Solutions for Achieving Peak Performance*. CRC Press, 2005.

[16] M. Cossentino, L. Sabatucci, and A. Chella. Patterns reuse in the PASSI methodology. In *ESAW*, pages 294–310, 2003.

[17] M. Cossentino and V. Seidita. Composition of a New Process to Meet Agile Needs Using Method Engineering. *Software Engineering for Large Multi-Agent Systems*, 3:36–51, 2004.

[18] C. Cote, D. Letourneau, F. Michaud, J. Valin, Y. Brosseau, C. Raievsky, M. Lemay, and V. Tran. Code reusability tools for programming mobile robots. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, 2004.

[19] S. DeLoach. Engineering organization-based multiagent systems. *Software Engineering for Multi-Agent Systems IV*, 3914:109–125, 2005.

[20] T. Do, M. Kolp, T. Hoang, and A. Pirotte. A Framework for Design Patterns for Tropos. *Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES 2003), Maunas, Brazil, October*, pages 3–343, 2003.

[21] A. Dominguez-Brito, D. Hernandez-Sosa, J. Isern-Gonzalez, and J. Cabrera-Gamez. Integrating robotics software. *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, 4, 2004.

[22] Foundation for Intelligent Physical Agents. *FIPA Abstract Architecture Specification*, 2000.

[23] Foundation for Intelligent Physical Agents. *FIPA Interaction Protocol Library Specification*, 2000.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[25] P. Giorgini, M. Kolp, J. Mylopoulos, and J. Castro. Tropos: A requirements-driven methodology for agent-oriented software. In *Agent Oriented Methodologies*, chapter II, pages 20–45. Idea Group Publishing, 2005.

[26] J. Gonzalez-Palacios and M. Luck. A Framework for Patterns in Gaia: A Case-Study with Organisations. *Agent-Oriented Software Engineering V: 5th International Workshop, AOSE 2004, New York, NY, USA, July 19, 2004: Revised Selected Papers*, 2005.

[27] M. L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, 32(4):548–566, 1993.

[28] J. W. Hooper and R. Chester. *Software Reuse - Guidelines and Methods*. Plenum Press, New York, 1991.

[29] R. Johnson. Documenting frameworks using patterns. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 63–76. ACM New York, NY, USA, 1992.

[30] J.Ulrich and J.Borenstein. Vfh*: Local obstacle avoidance with look-ahead verification. In *IEEE International Conference on Robotics and Automation*, pages 2505–2511, San Francisco (CA), USA, April 2000.

[31] R. Lajoie and R. K. Keller. Design and reuse in object oriented frameworks: Patterns, contracts, and motifs in concert. In V. Alagar and R. Missaoui, editors, *Object-Oriented Technology for Database and Software Systems*, pages 295– 312, Singapore, 1995. World Scientific.

[32] I. Macaluso, E. Ardizzone, A. Chella, M. Cossentino, A. Gentile, R. Gradino, I. Infantine, M. Liotta, R. Rizzo, and G. Scardino. Experiences with CiceRobot, a Museum Guide Cognitive Robot. *AI* IA 2005: Advances in Artificial Intelligence: 9th Congress of the Italian Association for Artificial Intelligence, Milan, Italy, September 21-23, 2005: Proceedings*, 2005.

[33] I. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I. Shu, et al. CLARAty: Challenges and Steps Toward Reusable Robotic Software. *International Journal of Advanced Robotic Systems*, 3(1):023–030, 2006.

[34] I. Nesnas, R. Volpe, T. Estlin, H. Das, R. Petras, and D. Mutz. Toward Developing Reusable Software Components for Robotic Applications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2001.

[35] N. J. Nilsson. *Artificial intelligence: a new synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[36] N. Noy, M. Sintek, S. Decker, M. Crubézy, R. Fergerson, and M. Musen. Creating Semantic Web Contents with Protégé-2000. *IEEE INTELLIGENT SYSTEMS*, pages 60–71, 2001.

[37] A. Oluyomi, S. Karunasekera, and L. Sterling. Description templates for agent-oriented patterns. *The Journal of Systems & Software*, 81(1):20–36, 2008.

[38] G. Picard. Cooperative Agent Model Instantiation to Collective Robotics. *Engineering Societies in the Agents World V: 5th International Workshop, Esaw 2004, Toulouse, France, October 20-22, 2004, Revised Selected and Invited Papers*, 2005.

[39] S. Poslad, P. Buckle, and R. Hadingham. The fipa-os agent platform: Open source for open standards. In *5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, Manchester, UK, 2000.

[40] L. Prechelt, B. Unger, and M. Philippsen. Documenting design patterns in code eases program maintenance. In *Proc. ICSE Workshop on Process Modeling and Empirical Studies of Software Evolution*, pages 72–76, 1997.

[41] L. Prechelt, B. Unger, W. Tichy, P. Brossler, and L. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, pages 1134–1144, 2001.

[42] L. Sabatucci, A.Garcia, N. Cacho, M. Cossentino, and S. Gaglio. Conquering fine-grained blends of design patterns. In H. Mei, editor, *10th International Conference on Software Reuse (ICSR 2008)*, volume LNCS 5030, pages 294–305. Springer-Verlag, 2008.

[43] L. Sabatucci, M. Cossentino, and S. Gaglio. A semantic description for agent design patterns. In *6th International Workshop "From Agent Theory to Agent Implementation"*, Estoril, Portugal (EU), May 13 2008.

[44] C. Schlegel. Communication Patterns as Key Towards Component-Based Robotics. *International Journal of Advanced Robotic Systems*, 3(1):49–54, 2006.

[45] D. Schmidt. Using design patterns to develop reusable object-oriented communication software. 1995.

[46] D. Stewart and P. Khosla. The Chimera Methodology: Designing Dynamically Reconfigurable and Reusable Real-Time Software using Port-Based Objects. *International Journal of Software Engineering and Knowledge Engineering*, 6:249–277, 1996.

[47] F. Stolzenburg and T. Arai. From the Specification of Multiagent Systems by Statecharts to Their Formal Analysis by Model Checking: Towards Safety-Critical Applications. *Multiagent System Technologies: First German Conference, MATES 2003, Erfurt, Germany, September 22-25, 2003: Proceedings*, 2003.

[48] M. Vokac, W. Tichy, D. Sjøberg, E. Arisholm, and M. Aldrin. A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns—A Replication in a Real Programming Environment. *Empirical Software Engineering*, 9:149–195, 2004.

[49] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. *Aerospace Conference, 2001, IEEE Proceedings.*, 1, 2001.

[50] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–315, 2000.

[51] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97)*, pages 226–235, January 1997.