An MDA Approach For Multi-Platform Agent Design Patterns

Massimo Cossentino (2,3), Luca Sabatucci (1), Salvatore Gaglio (1,2) *Member IEEE*

(1) DINFO - Dipartimento di Ingegneria Informatica, Università degli Studi di Palermo - Viale delle Scienze, 90128 Palermo, Italy
(2) Istituto di Calcolo delle Reti ad Alte Prestazioni, Consiglio Nazionale delle Ricerche;
(3) SET - Université de Technologie Belfort-Montbéliard - 90010 Belfort cedex, France
cossentino@pa.icar.cnr.it; sabatucci@csai.unipa.it; gaglio@unipa.it

***Abstract*—The object-oriented paradigm enriched with the definition of design patterns proved successful in lowering the development time and number of errors in produced software; now a similar phenomenon is occurring for multi-agent systems, where this is related to the great effort that has been currently spent in methodology definitions by several researchers. In this work we describe our experiences in the identification, description, production and application of agent patterns. Upon our pattern definition, we base a reuse process that can be considered as a crosscutting phase of the entire PASSI design methodology, from analysis to development. A classification criteria and a documentation template was defined in order to help user in selecting a pattern from the repository. The pattern solution is described using an MDA multi-level approach allowing us to automatically produce both source code (for multiple agent platforms) and UML diagrams (usually almost a structural and a dynamic diagram) useful for documenting the process. A concrete case study is reported in order to illustrate our pattern reuse approach, and some experimental results are reported for supporting the theory.**

***Index Terms*—Multiagent systems, patterns, process, reuse models and tools.**

I. INTRODUCTION

In the last years, multi-agent systems (MAS) have achieved a remarkable success and diffusion in employment for distributed and complex applications; experiences of industrial applications have been done, for instance, in e-commerce/e-market contexts where usage scenarios require high quality of design as well as secure, affordable and well-performing implementation architectures. In our research we deal with design process of agent societies; this activity involves a set of implications such as capturing the ontology of the domain, representing agent interactions (social aspects), and modelling the ability of performing intelligent behaviours. Several scientific works can be found in literature dealing with the same basic elements; they adopt several different approaches, they sometimes use different notations/languages and, above all, give different emphasis to different aspects of the process (for example the design of goals, communications, roles). In the following, we are going to pursuit a specific goal: lowering the time and costs of developing a MAS application without forgetting the necessary attention for quality of the resulting software and documentation.

We think that a fundamental contribution to this field could come by the adoption of proper reuse techniques and tools providing a strong support during the design phase. In pursuing these objectives we defined a reuse technique based on design patterns; this approach is integrated with the PASSI methodology [11], a step-by-step requirements-to-code methodology for developing multi-agent software.

Our work conceives the support for pattern reuse during almost all the PASSI design process: this practice is a sort of crosscutting phase occurring during the phases of PASSI. We define a pattern as a representation and implementation of some kind of (a part of) the system behaviours that solves a recurrent problem. In order to support the localization of our patterns for two of the most diffused agent platforms (JADE [6] and FIPA-OS [20]) we based our solution on the MDA architecture, using languages based on XML and transformations based on XSL. In this way designers can automatically generate not only the agent source code for the two selected agent-platforms but also an XMI representation of the UML diagrams representing the pattern solution. It is worth to note that although we actually worked with only FIPA-OS and Jade, our approach is general and the introduction of another platform in our repository is possible.

This paper is organized as follows: section II quickly overviews the agent paradigm from the software engineering point of view and shortly introduces the PASSI methodology; section III illustrates design patterns state of the art from literature, considering approaches used for both objects and agents. Section IV is the core of the paper: it presents our definition of patterns, based on a three-levels architecture (problem, solution, implementation); subsection IV.A discusses the process adopted for generating the source code for a specific execution environment; subsection IV.B presents AgentFactory, a tool developed for supporting pattern reuse. Section V describes the repository and the classification we adopted to describe our patterns. Finally,

Manuscript received October 9, 2001. (Write the date on which you submitted your paper for review.)

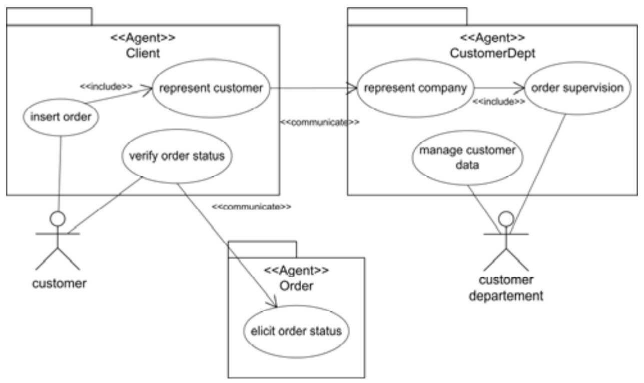

Figure 1 - A portion of the Agent Identification Diagram for the proposed case study system

section VI discusses experimental results obtained for a proposed case study and some conclusions are presented in section VII.

II. The Agent Paradigm

Several authors, nowadays consider the agent paradigm as the key for the implementation of flexible and scalable applications; Jennings argued that agents can be a successful solution for two major problems of contemporary design and development approaches: rigidity of components interactions and limitedness of available system's organizational structures [27]. We agree with these arguments and, in the following, we report a brief description of the concepts we refer in our work.

Agent. The traditional meaning of agent derives from Artificial Intelligence where *an agent is an entity capable of perceiving its own environment through sensors and acting through effectors* [49]. Wooldridge [60] introduces that *an agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous actions in order to meet its design objectives*. Jennings [27] speaks about *agents as a new theoretical model of computation that more closely reflects current computing reality than Turing Machines*.

Afterwards agent is endowed with additional characteristics [18]: its behaviour tends towards satisfying its own goals (proactivity), taking into account resources and skills in accordance with its internal knowledge and external events.

The intelligent behaviour, the ability to learn, and the mobility are skills assigned to agents depending on the nature of applications in which they are used [22][40].

Multi-Agent Systems. Interactions are one of the most important features of agents. An agent may communicate with other agents in order to collaborate for achieving some common goal. Multi-Agent Systems (MAS) are abstractions for dealing with complex and open problems: organization facilitates managing complexity by determining structures, norms and dependencies [63][64][65]. In some cases these may be established during design time but, in certain approaches organization emerges at run time.

A. PASSI

Autonomous agents represent a powerful instrument for decomposing, abstracting and organizing complex, distributed and evolving systems [61]. A new branch of the software engineering [27][60], the Agent-Oriented Software Engineering (AOSE, sometimes also referred to agent based software engineering, ABSE) is dealing with using agents to manage and conquer the complexity in a design process. Many approaches looked at the problem of requirements specification and design of agent-oriented systems with a formal approach [55][56][64] [13]. Non-formal specifications were adopted by several agent-oriented design methodologies (such as ADELFE [7], Gaia [65][66], Ingenias [45], MaSE [14], MESSAGE [9], Prometheus [44], ROADMAP [28], SODA [43], and Tropos [8]).

In our work we will refer to the PASSI methodology that we have been using for a few years. It will be the starting point and the natural context of our pattern definition and application.

PASSI [11] (Process for Agent Societies Specification and Implementation) drives the designer from the requirements analysis to the implementation phase in the construction of a multi-agent system. The work is carried out through five models composed by twelve sequential and iterative activities. Briefly, the models and activities of PASSI are:

- **System Requirements**. It is composed of four different activities and produces a description of the functionalities of the system-to-be; it allows an initial decomposition of the system according to the agent paradigm. The four activities are: (i) the Domain Requirements Description, where the system is described in terms of the required functionalities; (ii) the Agent Identification where agents are introduced for dealing with requirements; (iii) the Role Identification where agents' interactions are described by using traditional scenarios; (iv) the Task Specification where the plan of each agent is draft.
- **Agent Society**. It is composed of four activities producing an ontological view of the domain and the specification of the society. In the Domain Ontology Description the system domain is represented in terms of concepts, predicates, and actions. The Communication Ontology Description focuses on the agents' communications that are explained in terms of referred ontological elements, content language and protocol. In the Role Description the distinct roles played by agents and the tasks involved in playing each role are detailed.
- **Agent Implementation**. It is a model of the solution architecture in terms of classes and objects. It is composed of two main streams of activities (structure definition and behaviour description) both performed at the single-agent and multi-agent level of abstraction.
- **Code**. It is a model of the solution at the code level. It is largely supported by patterns reuse and automatic code generation.
- **Deployment**. It is a model of the distribution of the parts of the system across hardware processing units; it describes the allocation of agents in these units and any

constraint on migration and mobility.

- **Testing** has been divided into two different steps: the Agent and the Society tests. In the first one the behavior of each agent is verified with regards to the original requirements; during the Society Test, integration verification is carried out together with the validation of the overall results of iteration.

B. *A Case Study*

Throughout this paper we will refer to a case study that is the result of a relevant effort in MAS design and implementation. It regards a manufacturing system working in a market environment characterized by a globalization movement, where the rapid change of customer's desires and the mutable requirements of market require a good internal organization and a decrease in production time. Although we developed the system for a real company, the problem is very similar to the one studied in [37].

Agent technology assures the possibility to implement a flexible, scalable and decentralized architecture where autonomous entities could interact and organize themselves to achieve a general objective. MAS features such as autonomy, cooperation and mobility allow best fitting to the market changes [53].

The proposed manufacturing system was designed using the PASSI methodology; the real case from which we originated this case study regards an iron parts production chain (the products are directed to industry, carpentry, and other markets). Generally speaking, such a system may often be decomposed in three areas: i) a management sub-system for company trading affairs (relationships with suppliers and customers), ii) a flexible production chain management sub-system (storehouse and production control) and iii) an administrative sub-system to manage data and policies for the different areas of the whole system.

In the system requirements analysis, according to the PASSI process prescriptions, we identified 9 agent categories (more than one agent belonging to each category may exist in our MAS, for instance 1..n *Client* agents may interact with multiple customers at the same time). During the Agent Implementation phase we enriched the system with other agent categories (discovered later because they are depending from implementing choices), such as database wrapper agents, data caching agent and user agents; the whole system finally aggregates 27 agent categories. For the sake of brevity we will report and describe only a selection of the whole system. Figure 1 is a PASSI Agent Identification Diagram showing responsibilities of customer interaction area. In this diagram each agent is modeled as an UML package containing use cases (functional requirements): in this way the responsibility of accomplishing these requirements is partitioned among agents. A "communication" stereotype is introduced (this is not compliant to UML specifications) for representing interactions among use cases that are assigned to different agents. More in details, Figure 1 introduces two actors and three agents:

- The *Customer* actor is interested in placing an order (*insert order* use case) and it is represented by the *Client* agent that presents an interface where the *Customer* may insert a new order and verify the status of existing ones (v*erify order status* use case).
- The *Customer Department* is the area of the company specialized in interacting with customers. It is represented by the *CustomerDept* agent that is responsible for handling customer data (*manage customer data* use case) and negotiating with the *Client* agent the agreement on order parameters (*order supervision* and *represent company* use cases).
- The *Order* agent is responsible for eliciting the status of an order during its production; this is done by collecting data from other agent that are not shown in the figure for the sake of space.

III. Design Patterns and Agents

Designing and developing software is a rough duty because of the growing complexity of modern software systems but this activity might be simplified with an appropriate support of CASE tools and reuse techniques [58][26].

In this paper we focus on both of these aspects using a tool (AgentFactory, introduced later) that supports the reuse of agent design patterns. Design patterns born in a context that is far away from computer science: Alexander [2], a building architect, recognized common structures in cities, communities, and buildings that he considered to be "alive"; he called *patterns* these recurring themes: "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core solution to that problem*". Alexander intended those patterns as answers to questions such as "*Where should I place a terrace?*", "*How should I design the front entrance?*" or even more abstract such as "*How should I organize my community?*". His research resulted in the creation of a language [3] that he believed would enable people to design almost any kind of building and community.

A pattern is a three-part rule that expresses a certain relationship between a certain context, a problem, and a solution: this generic definition is the core of the work of Gamma et al. [17] that applied Alexander's idea to computer science and more specifically to the object-oriented paradigm. They used design patterns to describe best practices, good designs, and capture experience in such a way that it is possible for others to reuse them. Design patterns allow experts to systematically document, reason and discuss about solutions applied to specific problems. These solutions are validated by the experience rather than from testing and a project results more robust and simpler to modify with respect to traditional projects [46]. Design patters also provide a comprehensible way of documenting complex software architectures by expressing the structure and the collaboration of participants at a level that is higher than source code [50] [24].

The recent growing of methodologies for the design of multi-agent systems focused the attention of several researchers towards pattern concepts applied to agents. An evolution of design patterns should generate solutions rather than only describe them [15][59]: moreover, pattern description should be useful not only for understanding a pattern and its usage, but it should also indicate how the pattern is related to (or combined with) the others.

One of the first studies in agent patterns come from Kendall that [29][30] in illustrates an architecture based on role modeling: Kendall's patterns are object-oriented solutions based on a layered structure for implementing agents composed by: i) *Sensory*, for perceiving the environment, ii) *Beliefs*, representing agent's knowledge, iii) *Reasoning*, addressing goals, plans and capabilities, iv) *Action*, addressing agent's intentions (plans instantiated from the reasoning layer), v) *Collaboration*, including protocols, competitive bidding and coalition formation, vi) *Translation*, in which the agent formulates a message for another agent, and finally vii) *Mobility*, required for message transmission and reception.

In [38][54] several patterns for agents are presented, these are inspired by the real world and are described at a very high level of abstraction. The proposed structure for these patterns may be viewed as a specialized hierarchy of agents with the description of their communication mechanisms. These works do not investigate how patterns should cope with a specific implementing architecture and therefore remain quite abstract. Several authors consider design patterns as crosscutting with respect to the entire development process (from design to implementation); therefore the solution introduced moves from different abstraction levels that are generally clearly separated in the development process. Another approach proposes the use of Aspect Oriented Programming [31] for reducing the gap among design and implementation phases [23]. This work identified a set of aspects that are crosscutting with respect to the high-level elements of a MAS by exploring the influence of each aspect on the effective implementation (in an object-oriented language)

Another interesting source for patterns for agents comes from a well known AOSE methodology. In [39][16][32] the authors introduce a framework for using design patterns within the Tropos methodology. The proposed approach is "requirement driven": a problem is decomposed in a set of goals and their inter-dependencies. Patterns, in this context, are defined as design idioms based on social and intentional behaviours and are described from different points of view: i) the *social* dimension specifies agents and their interactions using a sort of second order logic language; ii) the *intentional* dimension is focused on services (seen as a functional link among agents); iii) the *structural* dimension explores the internal composition of agents in terms of Believes, Events and Plan [48][47]; iv) the *communication* dimension focuses on agent interaction protocols, using AUML [5] for describing communications; finally v) the *dynamic* dimension uses activity diagrams for defining what operations are involved in intentional and social actions.

IV. The Three-Levels Proposed Architecture

Our design patterns approach was initially conceived during the development of PASSI with the objective of introducing a viable reuse technique for the development of MAS. Classically, software design is structured in two domains [25]: the problem context and the solution context that are separated entities located in two different conceptual positions. The solution stays *in the computer and in its software* (machine domain) whereas the problem is *in the world outside from it* (application domain). During the solution discovery phase, the problem analyst's duty is to understand the problem exploring the context in which the machine will fit. Complex problems are faced using problem decomposition techniques.

A relevant part of the previously discussed literature proposes an abstract implementation for patterns that can be situated in the application domain [38] or a concrete architecture principally located in the machine domain using object-oriented elements for the solution [30][34][54].

Our approach to the definition of agent patterns spreads across both of the application and machine domains. However when using agents as a design paradigm the solution is more abstract than it is when expressed in object oriented terms; so we prefer to split the machine domain in two sub-domains, introducing the "agency domain" between the problem and implementation domains. In this way designing a multi agent system passes through three different levels of abstraction: (i) the "problem domain" catching the problem description; (ii) the "solution domain" giving a solution in terms of high level concepts coming from the agent paradigm (agents, communications, ontology, tasks, and so on); (iii) the "implementation domain" containing the effective implementation (often in object oriented terms). This three-levels architecture is the base for introducing our agent pattern definition (shown in Figure 2); in the following we will detail each of the domains of our pattern structure.

Pattern problem. A fundamental part of a pattern is the description of the problem (see the "problem domain" compartment of Figure 2) for which it may be useful. It is composed by: (i) *motivation*, an explanation of how (and why) the pattern works, and why it is good, putting into evidence

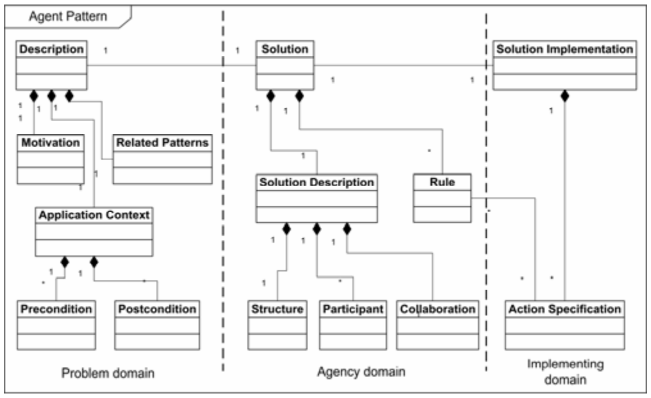

Figure 2 – Architecture of our Agent Pattern

steps and rules required to solve the problem (a concrete scenario used as an example of the pattern application is frequently employed); (ii) the *application context* describes the conditions under which the problem and the solution seem to recur, and for which the solution is desirable (pattern applicability); it is composed by *preconditions* (condition to verify before pattern application) and *postconditions* (conditions to verify after pattern application); finally (iii) *related patterns* element describes other patterns that could solve a similar problem. All these elements belong to the problem domain; they are expressed using PASSI artifacts like scenarios, requirements and domain ontology.

Pattern solution. It represents the solution (introduced when adopting the pattern) in terms of agent-oriented elements (see the "agency domain" compartment of Figure 2). The element "Solution" aggregates a textual and a formal description of the solution introduced by the pattern. The textual description illustrates the static structure and the dynamic behaviour introduced by the pattern in terms of structure, participants and collaborations. The formal description introduces some rules expressed by using an XML-based language that will be detailed in subsection IV.A.

Pattern solution implementation. This represents the lower level of the solution containing the effective implementation in an object-oriented language (it is very common in agent implementation to realize agency concepts using some object-oriented language like Java, although we acknowledge the limits coming from this practice, this remains a merely technological issue and its analysis is out of the scope of this paper). This phase uses diagrams from the *Agent Implementation Model* of PASSI; in *Agent Structure Definition* the involved agents are represented in terms of classes, attributes and methods using conventional UML class diagrams. The *Agent Behaviour Description* depicts the behaviour of agents involved in interactions using activity or state-chart diagrams.

A. Patterns and Meta Patterns: an MDA based approach

In this subsection we focus on the effective implementation of our problem-to-solution approach for agent patterns; during the definition of our architecture and the development of the supporting tools we chose to be compliant to some industrial standards; more specifically we adopted: i) MDA [41] (Model Driven Architecture) for implementing our three-levels architecture; UML (Unified Modeling Language) [42] for the graphical semiformal specification of pattern solutions; and

TABLE 1 – A PORTION OF THE CIM VIEW OF THE PARALLELSHARERESOURCE PATTERN (METAPATTERN LEVEL).

```
<Agent name="ParallelShareResource">
   <Resource name="sharing_resource"
             type="UserDefined"/>
   <Task name="ResourceServiceListener"
         type=”RequestParticipant”>
      <Action name="sendAgree"
             category="communication"
             act="send"
             performative="Agree"/>
         ...
   </Task>
   ...
</Agent>
```

XML/XSL [57] as a language for the effective implementation of patterns and transformations.

The Model Driven Architecture (MDA) has been proposed by the Object Management Group (OMG) as an open, vendor-neutral approach for separating business and application logic from underlying platform technologies. The main goal of MDA is to improve the quality of software products and the development process by allowing the reuse of models and transformations.

MDA proposes a definition of three models concerning different viewpoints of a system and addressing different levels of abstraction; the *Computation Independent Model* (CIM) describes the system in the environment in which it will operate, and what it is expected to do: it is an abstract representation of the system-to-be, independent from any computational aspect. The *Platform Independent Model* (PIM) defines the computational components that will satisfy the requirements, independently from the specific target platform. Finally, the Platform Specific Model (PSM) defines platform-specific concerns providing more or less details, depending on its purpose (for implementation purposes it can provide all the information needed to build the system and to put it on operation).

The interpretation of these MDA models depends on the specific meaning assigned to the term "platform"; in OMG specifications it is defined as "a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented“ (MDA Guide V1.0.1 [41], p. 13). In our context we consider the agent programming language and the deployment infrastructure as lower levels of our system, that is, the "platform" according to the previous

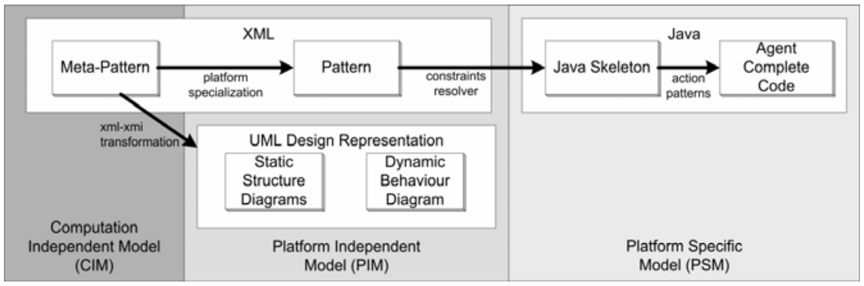
XML
Meta-Pattern
platform specialization
Pattern
constraints resolver
Java
Java Skeleton
action patterns
Agent Complete Code
xml-xmi transformation
UML Design Representation
Static Structure Diagrams
Dynamic Behaviour Diagram
Computation Independent Model (CIM)
Platform Independent Model (PIM)
Platform Specific Model (PSM)
Figure 3 - MDA view of the proposed pattern architecture

TABLE 2 - A PORTION OF THE PIM VIEW OF THE PARALLELSHARERESOURCE PATTERN (PATTERN LEVEL)

```
<Class name="ParallelShareResource"
       extends="AgentShell">
  <Attribute name="sharing_resource"
             type="UserDefined"/>
  <Class name="ResourceServiceListener"
             extends="TaskShell"
             type="RequestParticipant">
    <Method name="sendAgree" type="void">
      <Argoment name="msg"
                 type="FIPAACLMessage"/>

  <Code>sendAgree@FIPARequestParticipantTask</Code>
  </Method>
  ...
```

TABLE 3 - A PORTION OF THE PSM VIEW FOR THE PARALLELSHARERESOURCE PATTERN (JADE COMPLETE CODE)

```
public class ParallelShareResource extends Agent {
    private UserDefined sharing_resource;
    public class ResourceServiceListener
                 extends AchieveREResponder {
        public void sendAgree(ACLMessage msg) {
            // This method can be used to
            // prepare the msg
            // to reply with an Agree
            msg.setPerformative(ACLMessage.AGREE);
        }

    }
...
```

definition.

- Platform Specific Model. We selected some FIPA-compliant platforms for producing our agents: JADE [6] and FIPA-OS [21], which together represent a relevant part of the installed platforms in the Agentcities EU initiative [1]. In this way the source code of agents (that is principally Java-based) represents the *Platform Specific Model* of our architecture.
- Platform Independent Model. Generalizing the elements of the PSM level of our architecture we deduced the PIM one. We extracted some common features from platforms we considered: i) same coding language (Java), ii) compliance to the IEEE FIPA Abstract Architecture [19] and iii) several similarities in the structure and behaviour of agent-classes. Therefore we produced a "meta" representation of the analyzed platform that is our MDA *Platform Independent Model;* we use this generic platform to describe and abstract solution: the elements involved in this solution are different from those coming from the PSM solution, but a simple transformation allows to move between these two levels.
- Computational Independent Model. From these considerations we moved to a more abstract level: the agent solution (expressed in terms of the agent domain) corresponds to the *Computation Independent Model* (CIM) because it provides a view of the system that is independent from any computational aspects: the details of the structure of the solution in object oriented terms are hidden or as yet undetermined.

According to this decomposition we structure our pattern-to-solution process in three levels and in their correspondent transformations. In the following we will discuss about the **multi-platform implementation** of our pattern solution schematized in Figure 3.

The meta-pattern contains a generic description of the solution in terms of agency domain and therefore corresponds to the CIM. As an instance of meta-pattern, Table 1 shows a portion of the XML representation of the *ParallelShare-Resource* pattern; we specifically report a rule composed by a resource and a task. The resource is identified by a name and a type (that the user must define) whereas the task has a name and a set of possible actions (for sake of brevity only *sendAgree* is shown).

It is possible to perform a transformation from a meta-pattern model to a **pattern** model using a query/transformation language (we used a style-sheet based on XSL, XQuery and XPath). The transformation produces a description of the solution in object oriented terms. Table 2 shows the result of this transformation when applied to the *ParallelShareResource* pattern. The root element is a class defining the agent and containing an attribute (corresponding to the agent's resource) and an inner class (corresponding to the agent's task); the action was transformed in a method of the task class.

Using an opportune XML-XMI transformation we can also represent a meta-pattern solution or a pattern solution using the UML notation**;** resulting diagrams are useful as a documentation for designers; we prefer to dynamically generate diagrams instead of manually create them because in this way any successive maintenance operation on the meta-pattern has an immediate effect on its documentation.

Until now we have not yet chosen the desired agent platform. The solution is expresses in object oriented terms but it is still quite generic. Next step is the specialization for a specific platform; this transformation produces the **agent complete code** corresponding to the PSM. While localizing a pattern for a specific platform it is time to consider all the implementing details we have ignored up to the moment. For instance in FIPA-OS, an agent that wants to communicate must have a listener task, that is a specific class registered as a message dispatcher and containing an *handleX* method for each type of messages to catch (where the X has to be substituted with a specific communicative act [51]). The Jade framework provides not only a Task superclass (as it happens in FIPA-OS) but the suitable superclass has to be selected from a hierarchy of behaviour classes; besides, all of them may handle communications (they do not need a registration and/or a specific message dispacher).

The process for obtaining the final code is decomposed in three consecutive sub-steps: i) a transformation replaces all meta-level placeholders (specifying generic features as described before) with specific elements of the selected platform; this intermediate result is still expressed using a language based on XML but it is compatible with only one agent platform; ii) another transformation generates a first instance of the source code: it is only an empty skeleton

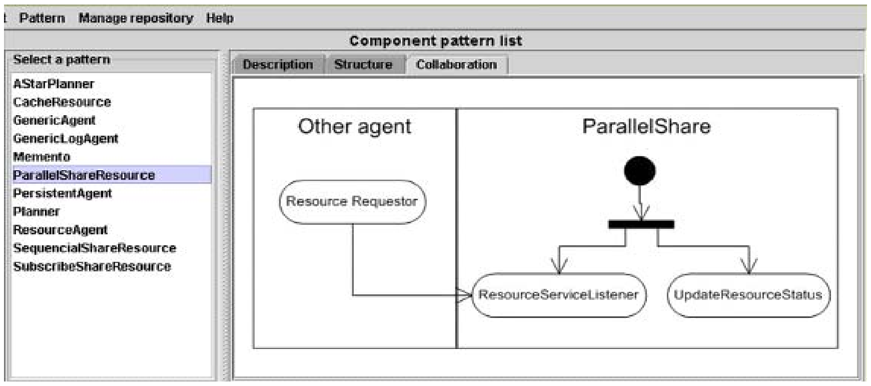

Figure 4 – An screenshot of the AgentFactory tool when browsing the repository. The ParallelShareResource pattern is selected in the left panel and the right panel shows its documentation.

defining the structure of classes constituting the agent; classes contain attributes and methods, but bodies of these methods are empty. A final transformation introduces the remaining code in the skeleton: in this phase, action-specification patterns are introduced (they are portions of code realizing some specific behaviour, like for instance the registration to the system yellow pages service).

In Table 3 the result of this transformation chain applied to the *ParallelShareResource* pattern is shown for the Jade deployment platform.

For summarizing the entire process we could track the sequence of changes from meta-pattern solution to "code" focusing on the root element: the agent. In the meta-pattern solution (Table 1) an agent is described by an *Agent* tag containing *Resource*, *Task* and *Action* tags. In the pattern solution (Table 2) the agent is described by a *Class* tag with an *extends* attribute set to the *AgentShell* string; this is an abstract placeholder indicating a generic super-class realizing the agent. The final transformation produces a Java class (Table 3) where the placeholder *AgentShell* has been substituted by *Agent* (as defined in Jade).

B. *Agent Factory: a tool for patterns reuse*

Patterns can contribute to significantly enhance the quality of software and this is one of the reasons that justify their diffusion. We think that under precise hypothesis, patterns can also provide another important contribution to the development process: they reduce the amount of work done when designing a system; in particular we aim at enhancing the amount of code automatically produced by a CASE tool during the development phase. In order to concretely reuse our patterns we developed the AgentFactory[1] tool; it can be used as a standalone application, as a web-based applet and as plug-in of the PASSI PTK tool (thus introducing in it the support for agent patterns according to the prescriptions of the PASSI methodology).

The development of AgentFactory motivates the complex multi-level and multi-platform architecture we previously described. The problem description is useful when the user browses the repository searching for a best solution to a specific design problem. The formal description of the solution, addressing the design phase, is useful because AgentFactory automatically modifies the artefacts under development introducing the changes required by the application of a pattern. Finally the previously described transformation chain generates the source code for the system. The main features of AgentFactory are:

- **Repository management**. AgentFactory manages a catalogue of patterns (categorized as presented in subsection IV.B); for each pattern an accurate documentation is provided for user navigation. In Figure 4 we can see a screenshot of the tool during the selection of a pattern from the repository.
- **Automatic Pattern Reuse**. AgentFactory handles projects where users can build their MAS selecting patterns from the repository; when a pattern is chosen a new agent is created or an existing one is modified according to the proposed solution.
- **Code Generation.** Diagrams may be transformed in source code by using the MDA-based transformation process described in subsection IV.A. Agent Factory supports both Jade and FIPA-OS implementation platforms.
- **UML Static and Dynamic diagrams generation.** The tool uses transformations for generating descriptions of patterns in XMI format and this may be useful both for documenting a single pattern or the entire developed system.
- **Reverse Engineering.** AgentFactory may parse the generated source code (even if modified by programmers) for rebuilding agents' internal representation. This feature is exploited in the Agile version of the PASSI methodology [12] that is principally oriented to coding and testing rather than designing. The building of a multi-agent system using Agile PASSI is an iterative process. In each iteration the developer uses patterns for modelling its system and then AgentFactory generates the code for the prototype (that is manually completed by the programmer). When this phase ends, the code may be imported (using the reverse engineering feature) so that a new iteration is possible (changes and more patterns may

[1] Website: http://mozart.csai.unipa.it/af/

be applied).

- **Import/Export as XMI.** In order to be integrated in other CASE tools (in particular with the PASSI Toolkit) AgentFactory supports the generation and the acquisition of an XMI representation of the designed multi-agent system.

V. Pattern Repository and Reuse

In section IV we presented our definition of agent patterns and our repository. We are now going to discuss the proposed approach for pattern classification based on functional and structural criteria, and to present a template for documenting each pattern. In subsection V.A, we will apply our reuse technique to the proposed case study.

A typical approach to patterns classification can be found in [17], where design patterns are classified according to two criteria: purpose and scope. With "purpose" authors refer to what a pattern does: they enumerate *creational* patterns (dealing with the process of object creation), *structural* patterns (dealing with the composition of classes and objects) and *behavioral* patterns (describing the interactions of classes/objects). The "scope" is directed to clarify patterns according to the different kind of elements they can be applied to (i.e. classes or objects). A different, agent-oriented, classification is proposed by Lind in [34] where patterns are classified in accordance to the views defined in the MASSIVE methodology [33] that are: i) Interaction view, ii) Role view, iii) Architecture view, iii) Society view, iv) System view, v) Task view, and vi) Environment view. Another (functional oriented) classification has been proposed in [36] where patterns are clustered in three categories: i) traveling (dealing with agent mobility issues), ii) task (regarding the breakdown of agent's tasks and the delegation of them from one agent to another) and iii) interaction (dealing with agent communications).

In our approach, we summarize these classifications using two criteria: the first criterion is the *application context* regarding the structural aspects of the solution. We enumerate four kinds of patterns in this first category:

- Multi-Agent patterns. Concerning collaborations among two or more agents; they can be thought as an aggregation of roles (played by several agents) and rules to observe during the interaction.
- Single-Agent patterns. They are entire-agent patterns; these patterns propose a solution for the internal structure of an agent together with its plans for realizing specific services.
- Behaviour patterns. They propose solutions addressing specific agent capabilities, introducing features to agent behaviours; we can look at each of them as a collection of actions.
- Action Specification patterns. They address an atomic functionality of an agent; their granularity can be resembled to a method of a class.

The second criterion is *functionality*; we consider four categories in it:

- Resource management patterns. They deal with information retrieval, manipulation of data sources, access to external resources.
- Communication patterns. They represent solutions to the problem of interaction among agents using an interaction protocol.
- Internal Architecture patterns. They deal with deliberation, plan management, message dispatching, knowledge management and other internal agent's basic functionality.
- Mobility. These patterns describe the possibility for an agent of moving from one platform to another, maintaining its knowledge.

Our repository actually contains 27 patterns (among multi-agents, single-agent and behaviour patterns), and about 170 action-specification patterns (many of them available for both Jade and FIPA-OS). In Table 4 we can find a summary of our pattern classification whereas Figure 5 shows the two types of relationships among patterns in the repository: *generalization* and *use*. The *generalization* relationship is used when a pattern extends another by adding new properties or elements; for instance the *ParallelShareResource* and the *SequentialShareResource* patterns are both specialization of the *ResourceAgent (*useful for assigning a resource to an agent). Both the *ParallelShareResource* and the *SequentialShareResource* patterns give an agent the ability to share its resource as a service for the community; the difference is in the mechanism used for updating the status of the resource. The second type of relationship (*use*) occurs when a pattern includes another in its solution for solving a sub-problem; for instance, the *ParallelShareResource* pattern uses the *FIPARequestInitiator* pattern for handling incoming FIPA Request communications.

The documentation schema we used in our repository is composed of a few keys derived from [17], however their use and meaning are different because of the specific needs of the agent-oriented paradigm.

- **Name**. The name of the pattern (preferably a single word or short phrase).
- **Classification**. The classification of the pattern addressing the already discussed categories (see Table 4).
- **Intent**. A short description of the pattern solution, its rationale and intent.
- **Motivation**. A description of pattern relevant *forces*, how they interact/conflict with one another, and goals they achieve. A concrete scenario which serves as the motivation for the pattern is frequently employed. Forces [17] reveal the intricacies of a problem and define the kinds of *trade-offs* that must be considered in the presence of the tension or dissonance they create.
- **Preconditions**. The *preconditions* under which the problem and its solution seem to recur, and for which the solution is desirable. This shows us the pattern

applicability context. It can also be thought of as the initial configuration of the system before the pattern is applied to it.

- **Postconditions**. It describes the state or configuration of the system after the pattern has been applied, including the consequences of applying the pattern, and other problems and patterns that may arise from the new context.
- **Solution** (Structure, Participants and Collaboration). Static relationships and dynamic rules describing how to realize the desired outcome. This is often equivalent to giving instructions which describe how to construct the required work products. The description of this solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting a concrete implementation.
- **Implementation availability**. Availability of the implementation code for the FIPA-OS/JADE platforms and UML diagrams of the solution for importing them in the existing system design.
- **Implementation description**. Comments on the most significant code fragments for illustrating the pattern implementation in the specific agent platforms
- **Related Patterns**. The static and dynamic relationships between this pattern and the others if any. Related patterns often have an initial or resulting context that is compatible with the resulting or initial context of another pattern. Such patterns might be *predecessor* patterns whose application leads to another one; *successor* patterns whose application follows from the current one; *alternative* patterns that describe a different solution to the same problem but under different forces and constraints; and *codependent* patterns that may (or must) be applied simultaneously with this pattern.

As an instance of pattern documentation we report the description of the *GenericAgent* pattern, frequently used as a starting point for building our agents.

Name: GenericAgent

Classification: internal architecture/single-agent

Intent: this pattern may be used as the root for applying all single-agent patterns because it gives to an agent the ability of registering/deregistering to/from the platform services (white and yellow pages).

Motivation: this pattern is useful for agents who want to discover whether the system offers a specific service and who provides it. The GenericAgent pattern adds the ability of registration to the platform so that the agent is reachable for conversations.

Preconditions: none.

Postconditions: the agent will be able of registering and de-registering to the white and yellow pages.

Solution: the agent is enriched with an attribute for listing the description of all its services offered to the community. A *registerDF()* and *registerAMS()* methods with their

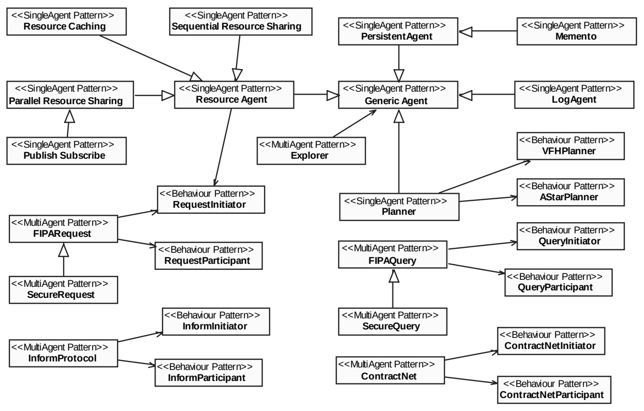

Figure 5 - Relationships among patterns in the repository

correspondent *deregisterDF()* and *deregisterAMS()* are provided.

Related Patterns: this pattern may be the predecessor for all single-agent patterns. The LogAgent pattern is a variant of the GenericAgent which may be used specifically for debugging/testing aims.

Other examples will be provided in the following sections.

A. Pattern Identification from PASSI models

Using the PASSI methodology for developing a multi-agent system, the designer faces three main different levels of abstraction (corresponding to three of the five PASSI models): i) System Requirements, ii) Agent Society and iii) Agent Implementation .

In the *System Requirements model* the designer analyzes and simplifies the problem with a top-down decomposition discovering system goals and interactions with the environment. Some high-level problems recur during this phase; these are classified in three categories: i) functional/non-functional requirements of the system, ii) scenarios and responsibilities, iii) components and services of the system.

The *Agent Society model* defines social interactions and dependencies among the agents involved in the solution. In this phase the agent is the central element of the analysis; it is considered as an autonomous entity, capable to play one or more roles in a social context. During its life an agent is able to communicate with other agents, to execute tasks and actions for achieving its own goals, and to provide services to the community. A solution described in these terms is a high level definition of “how” the system will work.

The *Agent Implementation model* is strongly related to the final coding activities and almost all of its phases receive a great input from selected patterns. This model is composed of a structural and behavioural definition of the MAS that is performed at both multi- and single-agent levels. As a result we have two structural diagrams and two behavioural diagrams: i) the Multi-Agent Structure Diagram (MASD) that is a diagram representing all the agents of the society as a class and communications among agents as relationships, ii) the Single-Agent Structure Diagram (SASD), reporting all the

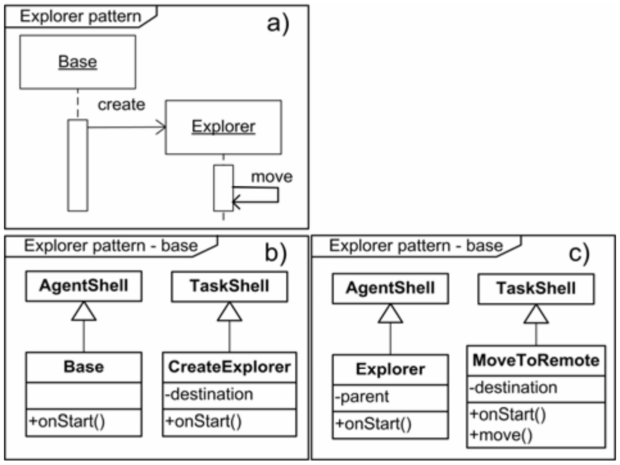

Figure 6 - Solution for Explorer pattern: a) role description, b) structure of Base role and c) structure of Explorer role

agent implementation details in one different class diagram for each agent,, iii) the Multi-Agent Behaviour Diagram (MABD), that is an activity diagram representing all the agents and their behaviours, and iv) a Single-Agent Behaviour Diagram (SABD), that is a diagram used to represent the algorithmic aspects of the solution.

The designer does not need to build this model from scratch because, as we described before, a great portion of it is encapsulated in pattern solutions.

The Agent Identification diagram (A.Id.) is the center of the functional requirements description; Figure 1 shows a portion of the A.Id. diagram for the manufacturing case study; this is an UML use case diagram used for representing agent functionalities and dependencies. In this type of diagram, packages are used for enclosing use cases that will be assigned to the responsibility of each agent.

In this diagram we might already discover patterns for solving multi-agent and single-agent problems at a high level of abstraction. An example is described in the following scenario from our case study (see Figure 1); let us consider the customer department area of a company which purpose is ensuring the acquisition of new orders for the production: the *customer* actor contacts the *customer-department* actor (responsible for order acquirement); as a consequence a *Client* agent is created for representing the *Customer* during the interaction. This agent moves itself to the remote host where it shows an interface for the introduction of new orders (*insert order* use case). Then the *CustomerDpt* and the *Client* agents bargain the best combination of price and deliver data that is advantageous for both the customer and the company. This negotiation is described in *represent customer* and *represent company* use cases (Figure 1). From this scenario we may use the *Explorer* pattern for realizing the basic part of the *Client* agent; this pattern allows to an agent the exploration of remote platforms with the intent of performing some kind of operation in them. The pattern is applicable to a couple of agents: an agent playing the role of *Base* and an agent playing the role of *Explorer*. The *Base* agent receives the ability of creating one or more *Explorer* agents (assigning them a mission) who has the ability to move from a platform to another.

Figure 6-a is a Role Identification diagram showing interactions proposed for these two roles: the *Base* role creates an agent who plays the *Explorer* role, moving itself to a remote platform. Figure 6-b represents the structure for the *Base* agent: this agent has the ability to activate one or more *Explorer* agents using the *CreateExplorer* task. Figure 6-c reports the structure of the *Explorer* agent who has a *MoveToRemote* with a *destination* attribute.

In our case study we applied the Base role of the pattern to the CustomerDept agent and the Explorer role to the Client agent. It is interesting to note how relevant the backlash of this pattern in the other phases of PASSI can be.

After the A.Id. phase, the designer performs the Roles Identification phase where he/she describes scenarios. When a scenario allows the identification of a pattern, the designer receives a significant help in designing it from the documentation that illustrates the collaborations of the pattern.

Figure 7 reports an example of a Role Identification diagram (describing the "insert new order" scenario), obtained after the application of the *Explorer* pattern. The sequence diagram starts with a client who contacts the customer department for a new order. The *Remote Interface* role moves to the client host and visualizes a graphical interface for collecting user input. Then the Customer Department, after using the *Customer Data Interface* role, activates a *Seller* and a *Mediator*; these agents will contract for establishing the better combination of parameters for both the customer and company. In this scenario the *Customer Data Interface* role is

TABLE 4 - THE PATTERNS IN OUR REPOSITORY

		Application Context			
		Multi-Agent	Single-Agent	Behaviour	Action Spec.
Functionality	Resource Management		Resource Sharing, Parallel Resource Sharing, Sequential Resource Sharing, Publish-Subscribe, Resource Caching		53*
	Communication	Request, Query, Inform, ContractNet, SecureRequest, SecureQuery		Request (I/P), Query (I/P), Inform (I/P), ContractNet (I/P)	66*
	Internal Architecture		GenericAgent, LogAgent, Planner, Memento, PersistentAgent	AStarPlanner	44*
	Mobility	Explorer			7*

* Names of these patterns have been omitted because the list would be too long and not really significant to the purpose of this paper

the *Base* role of the pattern whereas the *Remote Interface* role is the *Explorer* one.

Several candidate communications can be identified in the complete (Agent Identification) A.Id diagram and these can be realized using the corresponding communication/multi-agent patterns that are the most frequently used because of the peculiar agent feature to be strongly interactive and collaborative. For instance the FIPA Query and Request protocols are used in a great variety of contexts: in our case study the *FIPAQuery* pattern was used in eight agents whereas the *FIPARequest* in six agents. For instance we used the *FIPA Request* pattern for implementing the communication among the *Client* agent and the *Order* agent for implementing the functional dependencies among *order supervision* and *order status* use cases; this occurs when a client wants to verify the status of his/her order. The effect of applying this pattern is that *order status* is seen as a service offered by the *Order* agent to the society. This becomes more evident in the Task Specification (T.Sp.) phase of the PASSI methodology where one different activity diagram is drawn for each agent (an example is reported in Figure 8). In this type of diagram the designer studies the plan and tasks of each agent with the aim of defining a first hypothesis of the internal architecture of an agent and of its interactions with the other agents.

In these diagrams it is quite easy to find a great variety of patterns to reuse (principally behavioral ones). For instance for implementing a service three patterns (from our repository) may be considered: the *ParallelShareResource* pattern, the *SequentialShareResource* pattern and the *Publish-Subscribe* pattern; all of these give a solution to the same problem, but they starts from different contexts, preconditions and produce different postconditions. Here we report the description for the *ParallelShareResource* pattern (Figure 9) chosen for implementing the needed service.

Name: ParallelShareResource

Classification: resource management/single-agent

Intent: this pattern solves a problem of coordination: while continuously reading the status of a resource the agent has to provide a service based on this resource.

Motivation: when an agent provides a service depending

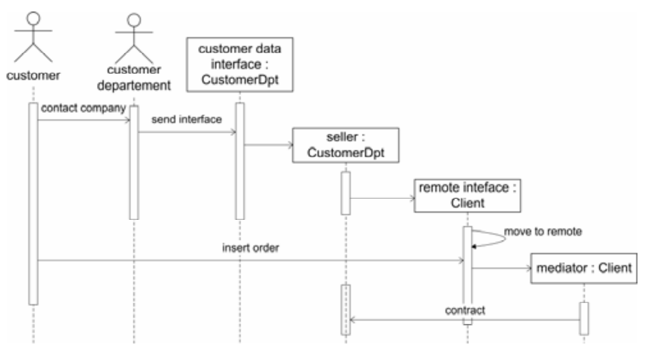

Figure 7 - Example of PASSI Role Identification

from a resource (resource sharing) which status changes in time, there is a problem of coordinating the update activity with the resource sharing. The pattern solves this problem putting in parallel these two activities: a listener task is always ready for offering the service, and in the same time a cyclic behaviour continuously reads the status of the resource. In this way when a request incomes, a reply is sent without any delay (thus obtaining a good response time). Two side effects are: i) the cyclic behaviour requires a lot of computational resources (it is always running), so other agents deployed in the same node may suffer of a significant slow down; ii) incorrect or older information may be given as a response because of the independency among service providing and updating cycle.

Pre-conditions: none

Post-conditions: the pattern uses the GenericAgent pattern for registering the service to the yellow pages and the FIPARequest Participant pattern for the incoming communications. The pattern specifically introduces a resource handling ability (to read/change the resource status) a synchronization mechanism to access the resource and a cyclic task for updating the resource status readings.

Solution: the solution proposed by this pattern is composed by only one participant: the *ParallelShare* agent. Figure 9-a shows the structure of this agent: it inherits the ability of registering itself to the platform (from the *GenericAgent* pattern) and the ability of manipulating a resource (regarded as an abstract element of the ontology domain). The agent also

TABLE 5 - STATISTICS FOR MAIN AGENTS OF THE MANUFACTURING SYSTEM CASE STUDY

Agents	Lines Of Code			Percentage (%)	
	Manually	Generated	Method Body	Generated	Method body
Agenda	1334	662	358	50	54
Customer	1306	746	392	57	53
CustomerDpt	1398	612	364	44	59
Order	502	410	226	82	55
Production	680	606	400	89	66
ProductionAdmin	488	248	132	51	53
Repository	3634	504	162	14	32
Supplier	570	198	120	35	61
SupplierDpt	476	248	132	52	53
Total	**10388**	**4234**	**2286**		
Average				**41**	**54**

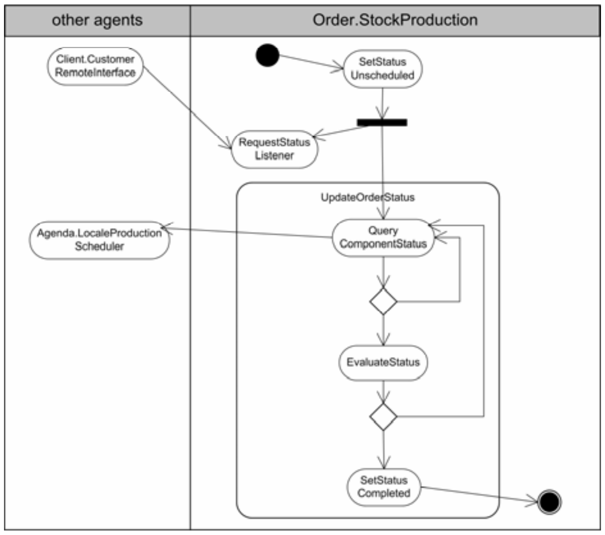

Figure 8 - Example of PASSI Task Specification for Order agent

receives the ability of participating to a conversation with other agents (using the *FIPARequest* pattern) and uses a cyclic task for updating the *ResourceStatus* value (see Figure 9-b); this task has an abstract method *updateStatus()* where the programmer has to put the code for reading the resource.

Related Patterns: the *SequentialShareResource* and the *Publish-Subscribe* solve the same problem using different approaches. A solution could be to read the status of the resource only when required thus avoiding the cost in computational time (*SequentialShareResource*). When several agents are interested on the same resource the *Publish-Subscribe* pattern may be useful: the agent responsible of the resource manages a list of subscribed agents to notify every time the resource status changes.

We used the *ParallelResourceSharing* pattern for implementing the architecture of the *order status* service: in Figure 8 we can observe the Task Specification diagram for the resultant *Order* agent; the agent's life begins with two tasks running in parallel: the *requestStatusListener* waiting for incoming requests (and accomplishing the service) and the *updateOrderStatus* responsible for evaluating the current status of the order; this is not an atomic operation, since an order may be composed by various components; therefore this task has to query for the status of all the components (information available via the *Agenda* agent that is not showed in reported diagrams) for building the complete information.

The *System Requirements* model for our case study has been completed with a massive use of patterns and the designer effort in this phase has been significantly reduced. Some work is still to be manually performed: as an instance the domain ontology (used by agents for their knowledge and communications) has to be completed with the specific concepts, predicates and actions of the application.

Finally the code is automatically generated by the AgentFactory tool, and the programmer can manually complete classes and methods with specific behaviours not contained in pattern solutions.

VI. Experimental Results

In this section we will report results obtained applying the PASSI methodology within our pattern approach to the manufacturing case study presented in subsection II.B.

The initial prototype has been designed without a significant reuse of patterns, since our repository was almost empty at that time. Now, in our experiment we reproduced it applying the patterns with the support of PTK and the AgentFactory tool thus obtaining good results in decreasing the production time and increasing the quality of code and documentation. As an instance it is worth to consider that with a few mouse clicks, selecting the *Explorer* and *FIPARequest* patterns we can produce an application composed of two agents, six classes and about 190 lines of code. The documentation is provided by UML class diagrams (structure of the system) and UML activity diagrams (behaviour of the system); these can be exported in XMI format and included in the design of the remaining part of the system.

The system we built was composed by 9 early agents (identified in the *System Requirements Analysis* phase) covering main functionalities of the system, plus other 18 agents identified later (during the *Agent Society* phase), used for implementing details of the system, such as database wrappers, data caching and user agents; the whole system is finally composed by 27 types of agent.

In order to quantify the contribution provided by the reused patterns, here we compare the number of lines of code (LOC) of the original agents with the LOC obtained by the patterns application. For the sake of brevity we will discuss statistics only for a selection of the whole system mainly focusing on the most important functionalities (agents representing the core of the system): in Table 5 we have summarized some data regarding nine agents. The first column reports the names of the agents we are studying. The "manually" column reports the number of lines of code manually produced for the first instance of the system (when the system was entirely manually developed). The "total" value (10.388 LOC) gives an indication of the dimension of the considered portion of the system (that is the core of the system). The "generated" column reports the number of LOC automatically produced by AgentFactory when the project was rebuilt using our pattern approach; we obtained a total of 4.234 automatically generated LOC. This "generated" code (as described in section IV.A) is the sum of code for the skeleton of classes (representing agents and tasks) plus the code inside the methods of these classes. The next column, labelled "method body", indicates the LOC automatically generated just for the body of method bodies.

The last two columns indicate (for each agent) the percentage of code automatically produced with respect to the total and the percentage of method body automatically produced with respect to the total automatically generated using patterns. As an average the pattern approach has produced a 40% of the final code of the system; best performances are obtained for agents that are involved in a

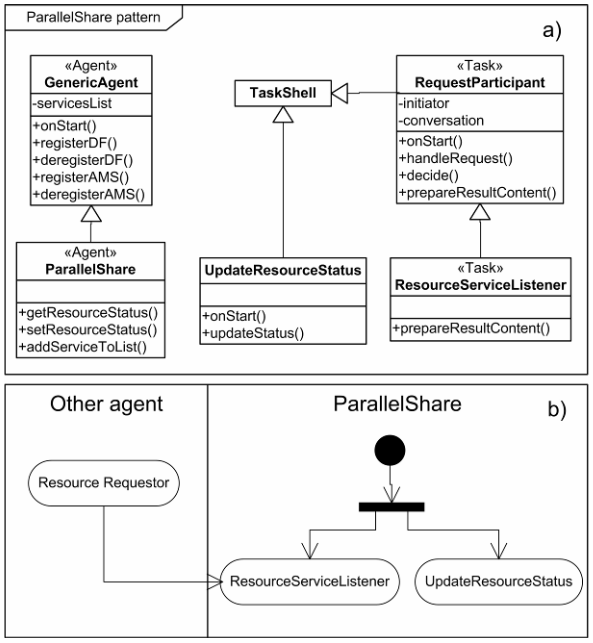

Figure 9 - Solution for ParallelShareResource pattern: a) structure and b) behaviour

great number of communications (we have a great number of patterns for that). Worst results are obtained for agents where tasks are essentially algorithmic.

About a half of the automatically generated code is about the internal part of methods (2286 over 4234 LOC). We think this is an important result because such a code is not generally produced by a conventional CASE tool.

We have here discussed only one third of the system agents, being the remaining part even more significant in size; nonetheless, percentage results remain nearly the same.

VII. Conclusions and Future Works

In this work we discussed the impact of pattern reuse in PASSI, a complete design methodology for multi-agent systems that is supported by PTK (PASSI ToolKit), an add-in for Rational Rose, and AgentFactory, a pattern reuse tool. The use of this methodology and the related tools allowed us the construction of significant projects with very good results in terms of automatically generated code and saving in time (as an example we reported here the rebuild of a part of an industrial application).

Our MDA oriented approach to the representation of patterns, based on XML/XSL, allows us the generation of code for agents for two different multi-agent FIPA-compliant platforms (FIPA-OS and JADE). In order to cover the entire process we use different representation languages (UML for diagrams, XML for patterns and JAVA for the final code) and we apply several transformations (mainly expressed using XSL style-sheets).

Experimental results have demonstrated the goodness of the approach that is however strongly affected by the number of patterns in the repository and the support the tool offers to designer in terms of automatically performed operations.

Our work has created new interesting issues we reserved to explore as future works. Basically we think that our pattern approach could be extended in order to become independent not only from the deployment platform (as it already is) but also from the adopted agent-oriented methodology. Also we are working on the support for the automatic generation of documentation about the generated solution in order to achieve a further improvement in the maintenance process.

References

[1] Agentcities Network Services - [Available on Internet] http://www.agentcities.net/
[2] Alexander C. 1979. *The Timeless Way of Building*. Oxford University Press
[3] Alexander C., Ishikawa S. and Silverstein M. 1977. A Pattern Language. Oxford University Press, New York
[4] Aridor Y. and Lange D. B. 1998. Agent design patterns: Elements of agent application design. In Proceedings of the second international conference on Autonomous agents (Agents'98) Minneapolis, Minnesota, United States, pp. 108 - 115
[5] Bauer B., Müller J. P., Odell J. 2001. Agent UML: A Formalism for Specifying Multiagent Interaction, In *Agent-Oriented Software Engineering*, edited by P. Ciancarini and M. Wooldridge, Springer-Verlag, Berlin, pp. 91-103
[6] Bellifemine F., Poggi A. and Rimassa G. 2001. Developing Multi-agent Systems with JADE. In proceedings of *The 7th international Workshop on intelligent Agents. Agent theories Architectures and Languages* (July 07 - 09, 2000) edited by C. Castelfranchi and Y. Lespérance, LNCS 1986, Springer-Verlag, London, pp. 89-103.
[7] Bernon, C., Camps V., Gleizes M. P., and Picard G. 2005. Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology, To appear in *Agent-Oriented Methodologies* edited by B. Henderson-Sellers and P. Giorgini, Idea Group Pub
[8] Bresciani P., Giorgini P., Giunchiglia F., Mylopoulos J., and Perini A. 2004. TROPOS: An Agent-Oriented Software Development Methodology, *Journal of Autonomous Agents and Multi-Agent Systems*, Kluwer Academic Publishers 8(3), pp. 203-236
[9] Caire G., et al. 2002. Agent Oriented Analysis using MESSAGE/UML. In *The Second International Workshop on Agent-Oriented Software Engineering (AOSE'01)*, LNCS 2222, Springer-Verlag, pp. 119-135
[10] Chella A., Cossentino M., Sabatucci L. and Seidita V. 2006. Agile PASSI: An Agile Process for Designing Agents, In *International Journal of Computer Systems Science & Engineering*. Special issue on "Software Engineering for Multi-Agent Systems" (in printing).
[11] Cossentino M. 2005. From Requirements to Code with the PASSI Methodology, In *Agent-Oriented Methodologies,* edited by B. Henderson-Sellers and P. Giorgini, Idea Group Inc., Hershey, PA, USA
[12] Cossentino M., Gaglio S., Sabatucci L. and Seidita V. 2005. The PASSI and Agile PASSI MAS Meta-models Compared with a Unifying Proposal, In proceedings of *4th International Central and Eastern European Conference on Multi-Agent Systems* (CEEMAS'05) 15-17 September, Budapest, Hungary, LNCS 3690, pp. 183 - 192
[13] De Giacomo G., Lespérance Y., Levesque H.J., and Sardina, S. 2004. On the Semantics of Deliberation in IndiGolog - From Theory to Implementation, In *Annals of Mathematics and Artificial Intelligence*, 41(2-4), pp. 259-299,
[14] DeLoach S. A., Wood M. F. and Sparkman Cl. H. 2001. Multiagent Systems Engineering. In the International Journal of Software Engineering and Knowledge Engineering, Vol. 11 (3), pp. 231-258.
[15] Deugo D. and Weiss M. 1999.A case for mobile agent patterns. In *Mobile Agents in the Context of Competition and Cooperation* (MAC3) Workshop Notes, at Autonomous Agents'99, pages 19-22
[16] Do T. T., Kolp M., Hang Hoang T. T. and Pirotte A. 2003. A Framework for Design Patterns for Tropos, In proceedings of the *17th Brazilian Symposium on Software Engineering* (SBES 2003), Maunas, Brazil
[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley
[18] Ferber J. 1999. *Multi-Agent Systems*, Addison-Wesley: Reading, MA

[19] FIPA Abstract Architecture – [Available on Internet] http://www.fipa.org/repository/architecturespecs.html

[20] FIPA ACL Specification – [Available on Internet] http://www.fipa.org/repository/aclspecs.html

[21] FIPA-OS Website - [Available on Internet], http://fipa-os.sourceforge.net

[22] Franklin S, and Graesser A. 1996. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents, In *Intelligent Agents III – Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, LNAI, 1193, Springer Verlag, pp. 21-35

[23] Garcia A., Silva V., Chavez C. et al. 2002. Engineering multi-agent systems with aspects and patterns. In *Journal of the Brazilian Computer Society*, July 8(1), pp. 57-72

[24] Greenfield J. and Short K. 2003. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM Press, New York, NY, pp. 16-27.

[25] Jackson M. 2001. Problem Frames: Analysing and Structuring Software Development Problems, Addison-Wesley

[26] Jacobson I. 1992. *Object-oriented software engineering*. ACM Press

[27] Jennings N. R. 2000. On Agent-based Software Engineering, In *Artificial Intelligence* 117, pp. 277-296

[28] Juan T., Pearce A., and Sterling L. 2002. ROADMAP: Extending the Gaia Methodology for Complex Open Systems, In Proceedings of the *1st Int. Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS'02)*. Bologna, Italy

[29] Kendall E. A. and Jiang C. 1997. Multiagent system design based on object oriented patterns. In the *Journal of Object Oriented Programming*, 10(3), pp 41-47

[30] Kendall, E. A., and Malkoun, M. T. 1996. The Layered Agent Patterns. Pattern Languages of Programs, PLoP'96.

[31] Kiczales, G. 1996. Aspect-oriented programming. *ACM Comput. Surv.* 28, 4es (Dec. 1996)

[32] Kolp M., Tung Do T., Faulkner S. and Hang Hoang T.T. 2003. Architectural Styles and Patterns for Multi-Agent Systems. In *Learning, communication and Coordination in Multi Agent Systems: Theory and Application* edited by L. Jain, World Scientific

[33] Lind J. 2001. Iterative Software Engineering for Multiagent Systems - The MASSIVE Method, LNCS 1994, Springer-Verlag

[34] Lind J. 2002. Patterns in agent-oriented software engineering. online paper. "http://www.agentlab.de/agent patterns.html".

[35] Luck M., and d'Inverno M. 2001. A Conceptual Framework for Agent Definition and Development, In *The Computer Journal*, 44(1), pp. 1-20

[36] Malyankar R. 1999. A pattern template for intelligent agent systems. In *Workshop in Agent-Based Decision Support for Managing the Internet-Enabled Supply Chain*, Seattle, WA

[37] Maturana F. P. and Norrie D. H. 1996. Multi-agent Mediator architecture for distributed manufacturing, In *Journal of Intelligent Manufacturing*, 7(4), pp. 257 - 270

[38] Meira N., e Silva I. C., and da Silva A. R. 2000. An agent pattern language for a more expressive approach. In Proceedings of EuroPLOP

[39] Mouratidis H., Weiss M. and Giorgini P. Modelling Secure Systems Using an Agent-Oriented Approach and Security Patterns, International Journal of Software Engineering and Knowledge Engineering, World Scientific (accepted for publication - in press)

[40] Odell, J. 2000. Agent Technology - Green Paper, OMG - Agent Platform Special Interest Group. http://www.objs.com/agent/index.html

[41] OMG Model Driven Architecture - [Available on Internet] http://www.omg.org/mda/

[42] OMG UML Specification - Object Management Group, 1999 - [Available on Internet] http://www.omg.org/uml/

[43] Omicini A. 2001. SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems, In *Agent-Oriented Software Engineering* edited by P. Ciancarini and M. Wooldridge, Springer-Verlag, pp. 185-194.

[44] Pandgham L and Winikoff M. 2002. Prometheus: A Methodology for Developing Intelligent Agents in proceedings of the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS'02.

[45] Pavón J., and Gómez-Sanz J. 2003. Agent-Oriented Software Engineering with INGENIAS, In *Multi-Agent Systems and Applications III*, (CEEMAS 2003) edited by V. Marík, J. Müller and M. Pechoucek, Springer-Verlag, LNCS 2691, pp 394-403

[46] Prechelt L., Unger B., Philippsen M. and Tichy W. 2002. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. Softw. Eng.* 28(6), pp. 595-606.

[47] Rao A. S. and Georgeff M. P. 1991. Modeling rational agents within a BDI-architecture, In *Readings in Agents*, edited by M. N. Huhns and M. P. Singh, Morgan Kaufmann Publishers, San Francisco, CA, pp. 317-328

[48] Rao A. S. and Georgeff M. P. 1995. BDI Agents: from Theory to Practice, In Proceedings of the *First International Conference on Multi-Agent Systems (ICMAS'95)*, pp. 312-319

[49] Russell S. and Norvig P. 1995. *Artificial Intelligence; A Modern Approach*, Englewood Cliffs, NJ: Prentice Hall

[50] Schmidt D. and Stephenson P. 1995. Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms, In proceedings of the *9th European Conference on Object-Oriented Programming*, LNCS 952, pp. 399 - 423

[51] Searle, JR, 1969, Speech Acts. Cambridge: Cambridge University Press.

[52] Serrano J. M. and Ossowski S. 2004. On the Impact of Agent Communication Languages on the Implementation of Agent Systems, In proceedings of the *Eighth International Workshop on Cooperative Information Agent (CIA 2004)*, LNCS 3191, pp. 92 - 106

[53] Shen W. and Norrie D.H. 1999. Agent-Based Systems for Intelligent Manufacturing: A State-of-the-Art Survey. In *Knowledge and Information Systems, an International Journal*, 1(2), pp. 129-156

[54] Shu S. and Norrie D. 1999. Patterns for adaptive multi-agent systems in intelligent manufacturing, In Proceedings of the *2nd International Workshop on Intelligent Manufacturing Systems*, Leuven, Belgium, pp. 67–74

[55] Singh M. P., Rao A. S., and Georgeff M. P. 1999. Formal methods in DAI: logic-based representation and reasoning. In *Multiagent Systems: A Modern Approach To Distributed Artificial intelligence*, G. Weiss, Ed. MIT Press, Cambridge, MA, 331-376.

[56] Smullyan R. M. 1968. *First-Order Logic*, Courier Dover Publications

[57] The World Wide Web Consortium (W3C) - [Available on Internet], http://www.w3.org/

[58] Wasserman A. I. 1990. Tool integration in software engineering environments. In proceedings of *the international Workshop on Environments on Software Engineering Environments* (Chinon, France), edited by F. Long, Springer-Verlag New York, New York, NY, pp. 137-149.

[59] Weiss M. 2003. Pattern-Driven Design of Agent Systems: Approach and Case Study, Lecture Notes in Computer Science, 2681, pp. 711 - 723

[60] Wooldridge M. 1997. Agent-based software engineering, *IEE Proc Software Engineering* 144, pp. 26-37.

[61] Wooldridge M. 2000. Reasoning about Agents, The MIT Press, Cambridge, MA

[62] Wooldridge M. and Jennings N. R. 1994. Agent theories, architectures, and languages: A survey. In proceedings of *The Workshop on Agent theories, Architectures, and Languages on intelligent Agents* (Amsterdam, The Netherlands). Edited by M. J. Wooldridge and N. R. Jennings, Springer-Verlag New York, New York, NY, pp. 1-39

[63] Wooldridge M. and Jennings N. R. 1995. Intelligent agents: theory and practice. In *The Knowledge Engineering Review* 10(2), pp. 115-152

[64] Wooldridge M., Fisher M., Huget M. P. and Parsons S. 2002. Model Checking Multi-agent Systems with MABLE, In proceedings of the *1st Int. Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS'02)*, Bologna, Italy, pp. 952 - 959

[65] Wooldridge M., Jennings N. R., and Kinny, D. 2000. The Gaia Methodology for Agent-Oriented Analysis and Design, *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), pp. 285-312

[66] Zambonelli F., Jennings N. and Wooldridge M. 2003. Developing Multiagent Systems: the Gaia Methodology, In *ACM Transactions on Software Engineering and Methodology*, 12(3), pp. 417-47