# The Reflective Blackboard Architectural Pattern

**Otavio Rezende da Silva Alessandro Fabricio Garcia**
**Carlos José Pereira de Lucena**

Grupo TecComm – SoC + Agents
Departamento de Informática – PUC-Rio
Rua Marquês de São Vicente, 225 – Ed. Pe. Leonel Franca, 10º Andar Rio de Janeiro – Brazil
{otavio, afgarcia, lucena}@inf.puc-rio.br
http://www.teccomm.les.inf.puc-rio.br

PUC-Rio Inf.MCC24/02 September, 2002

**Abstract.** Software architectures of large multi-agent systems (MASs) are inherently complex and have to cope with an increasing number of system-wide properties and their corresponding control policies. With the openness and increasing size and complexity of these systems a more sophisticated software architectural approach becomes necessary. In this context, we propose the *Reflective Blackboard* architectural pattern, which is the result of the composition of two other well-known architectural patterns: the *Blackboard* pattern and the *Reflection* pattern. The proposed pattern provides, early in the architectural design stage, the context in which more detailed decisions related to systemic properties and associated policies can be made in late stages of MAS development. The pattern allows a better separation of concerns, supporting the separate handling of control strategies by means of the computational reflection technique. Moreover these control activities are handled independently from the application data and agents, providing a better architecture for real-life multi-agent systems. An electronic marketplace architecture, with the goal of interconnecting providers and consumers of goods and services to find one another and transact business electronically, is assumed as a case study through the paper to clarify all the expressed concepts and to show the applicability of our proposal.

**Keywords**: Multi-agent systems, software engineering, architectural patterns, computational reflection, blackboard architectures.

**Resumo.** Arquiteturas de sistemas multi-agentes de larga escala (MASs) são inerentemente complexas e são associadas com um grande número de propriedades sistemicas e suas políticas de controle. A abertura e o aumento crescente do tamanho e complexidade destes sistemas requer uma abordagem arquitetural mais sofisticada. Neste contexto, nós propomos o padrão arquitetural *Reflective Blackboard*, que é definido pela composição de outros dois padrões conhecidos: o padrão *Blackboard* e o padrão *Reflection*. O padrão proposto introduz, na fase arquitetural de desenvolvimento, o contexto que dará suporte a decisões detalhadas relacionadas as propriedades sistemicas nas fases posteriores de desenvolvimento. O padrão promove uma melhor separação entre tais propriedades e respectivas estratégias de controle através da técnica de reflexão computacional. Além disso, essas atividades de controle são tratadas independentemente dos dados e agentes da aplicação, fornecendo uma arquitetura adequada para sistemas multi-agentes complexos. Um marketplace eletrônico, que tem como objetivos conectar fornecedores e consumidores de bens e serviços e dar supporte a transações comerciais eletronicamente, é usada como estudo de caso através do artigo para ilustrar os conceitos apresentados e mostrar a aplicabilidade da nossa proposta.

**Palavras-chave:** Sistemas multi-agentes, engenharia de software, padrões arquiteturais, reflexão computacional, arquiteturas blackboard.

# 1 Introduction

Software technology is undergoing a transition from monolithic architectures, constructed according a single overall design, into open architectures composed of conglomerates of collaborative, heterogeneous, and independently designed agents and multi-agent systems (MAS). These architectures are driven by additional system-wide properties, such as coordination [23, 29], adaptability [31], mobility [32], security [24, 42] and manageability [23]. Each of these system properties encompasses policies (or strategies) that control the application agents and data. Among the problems inherent in such architectural transition, none is more serious than the difficulty to incorporate and compose multiple control strategies, requiring a more sophisticated software architectural approach. The basic functionalities of agents already are quite complicated in large-scale multi-agent architectures, and so control strategies should be designed separately from the agents' basic behaviors. The degrees to which quality requirements (e.g. reusability and maintainability) are met on an MAS are largely dependent on its software architecture [14]. Hence, if an MAS architecture that includes suitable support for handling multiple control strategies is chosen from the outset, it is more likely that distinct quality attributes will be achieved throughout the development of multi-agent software.

Software architecture [14] has emerged as a central discipline for software engineers of complex systems in the last decade. This discipline is concerned with defining high-level styles and patterns for fundamental structure and organization of software systems. An architectural pattern [3] provides a solution to a recurring problem, defining a set of components as well as rules that organize the relationships between them. Architectural patterns are the building blocks of large-scale software architectures, which are likely to include instances of more than one of these patterns, composed in arbitrary ways [1]. A specific composition of architectural patterns, which occurs often in a given domain, is defined as another pattern. In the context of MAS, the blackboard architectural pattern has been widely used as a useful metaphor for communication and coordination of heterogeneous and separately designed agent organizations, providing low temporal and spatial coupling [4, 8].

The idea of blackboard architectures is not new, and they were first introduced in the Hearsay II project [27]. Nowadays they are experiencing a renaissance with various industry-strength tuplespace architectures, such as IBM TSpaces [11] and JavaSpaces [33]. The *Blackboard* pattern [3] encompasses the definition of components and rules of blackboard architectures: multiple *knowledge sources* or independent *agents* are the components that implement specific parts of the application logic, and interact with each other by using the *blackboard* component; the blackboard is a data structure that is used as the general communication and coordination mechanism for the multiple agents, and is managed and arbitrated by a *controller* component. However, the pattern does not specify explicitly how the controller component deals with distinct control strategies to manage the blackboard, and how to separate such strategies from the application agents and data, which leads to multi-agent software architectures that are difficult to maintain, understand and reuse.

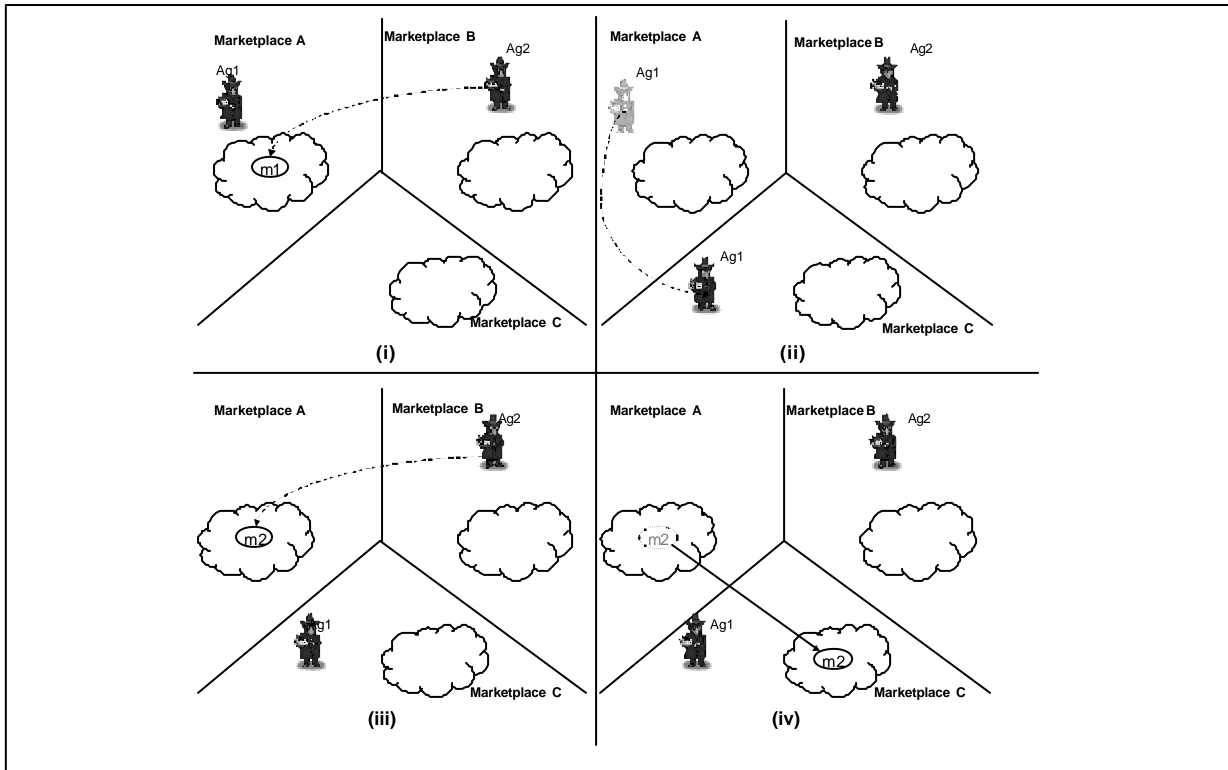


**Fig 1** Motivation example illustrated

In this paper we propose the definition of the *Reflective Blackboard* architectural pattern that is built from the composition of two well-known architectural patterns [3]: the *Blackboard* pattern and the *Reflection* pattern. As a result of the proposed composition, the components of the Reflection pattern are used to refine the overall structure of the Blackboard pattern and promote better separation of concerns. Separation of concerns is a fundamental principle of software engineering, and it is achieved in reflective architectures by separating the system in two levels: the *base level* and the *meta-level*. The Reflective Blackboard architectural pattern follows this organization: the controller is situated at the meta-level of multi-agent systems, while the application agents and data are encapsulated at the base level. Our primary claim is that systemic properties of an MAS are handled at the meta-level, completely separated from its basic functionality, and achieved by applying reflection mechanisms upon the blackboard operations and by invoking appropriate control strategies. The combination of the Reflection pattern with other patterns has already been successfully used to define new patterns for other complex domains [19, 20, 21].

The Reflective Blackboard architectural pattern is independent of programming languages and specific implementation frameworks, and its use can minimize the complexity caused by the presence of numerous system-level properties in MASs. The proposed pattern is targeted first of all to engineers of complex multi-agent applications who must define and implement the different control strategies that drive their systems. The proposed pattern can also be interesting for developers of different types of blackboard infrastructures and frameworks since they can decide to incorporate reflective capabilities directly into their products. The remainder of this

paper is organized as follows. Section 2 presents the Reflective Blackboard architectural pattern. Section 3 discusses the proposed pattern and a collection of other related patterns, together with guidelines for their implementation, combination, and practical use in MAS development. Section 4 points out some concluding remarks and directions for future work.

# 2 The Reflective Blackboard Architectural Pattern

## 2.1 Motivation Example: Electronic Marketplace

Consider a marketplace application where buyers and sellers negotiate products and services. Sellers advertise their desire to sell products or services, submitting offers to the marketplace. Buyers access the marketplace to submit bids in order to buy products and services, and simultaneously to find prospective sellers. Once the buyers have found an appropriate seller, they continue to communicate indirectly through the marketplace in order to negotiate and make proposals and counterproposals. Some buyers eventually join up with each other to buy products together and minimize costs. The marketplace is open, i.e. agents can join or leave it at any time, and agents are not initially aware of their counterparts. Buyers and sellers visit different marketplaces in the network in order to achieve their individual goals.

The blackboard architectural pattern is a natural solution for the marketplace problem and is widely used in practice to develop sophisticated marketplaces [35, 36, 37, 38]. Blackboards are the commonplace where commerce transactions are conducted and products or services are traded. Different blackboards represent distinct marketplaces (Fig 1) and work as a message exchange infrastructure, used by the agents to communicate and coordinate their activities. Buyer and seller agents are the knowledge sources that cooperate and compete to process sales transactions for their owners. Agents write and read messages on the blackboards, with each message encapsulating a bid, an offer, a proposal or a counterproposal. Each host holds one or more marketplaces (i.e., blackboards). The controller component manages the marketplace by ensuring its control policies.

In marketplace applications, one of the strategies must deal with the communication control in the presence of mobility. Distinct marketplaces are spread over the network and, as a consequence, buyer and seller agents move to different hosts to find products and services required by their owners. In the beginning of their conversation the negotiating agents know each other's locations and can send messages to the destination host and the target marketplace so that the receiver can read and process them. However, since buyer and seller agents must visit different marketplaces, the hosts where they exist are likely to change. After moving from a marketplace to another the agent needs to continue receiving all the messages that were addressed to it. On the other hand, the agent that will be sending messages does not necessarily know that the receiver has moved to another marketplace and thus can continue sending messages to the previous marketplace. Since every message must reach its eligible receiver, they must be forwarded to the receiver's new marketplace. This strategy for controlling the communication by forwarding

messages across multiple hosts should be seamless to both agents, so that they do not need to be aware of it.

This example is illustrated in Fig 1. In (i) Ag1 and Ag2 are agents that know each other's locations. Ag2 can thus send messages to Ag1 directly to the blackboard which represents Ag1's marketplace. In (ii), Ag1 has moved to a different marketplace, and in (iii) Ag2 has sent another message to the environment where Ag1 used to live. In this way, a control strategy that redirects the message to Ag1's new marketplace should exist. This control strategy is represented in (iv), and is termed *mobile communication strategy*. In addition to this communication strategy, robust marketplaces must contain control strategies for *coordinating* agent activities, *managing* the marketplace transactions, insuring s*ecure* commerce, providing *reliable* communication between agents and so forth. We use the mobile communication strategy to illustrate the use of the proposed pattern in the next section. Section 3 shows how our pattern provides a suitable structure for incorporating and integrating multiple control policies into a MAS based on a reflective blackboard architecture.

## 2.2 Problem

The blackboard architectural style already has been widely used to tackle problems that have non-deterministic solutions [3]. When MASs [13] are concerned, this architectural pattern is widely accepted to implement the agents' communication [8] and coordination [4]. Recent research also has achieved positive results in using blackboards to implement agents' mobility and persistence [16]. As described previously, the Blackboard pattern structure is divided into three components: the blackboard itself, a group of knowledge sources (or agents), and a controller component. The left upper side of Fig 2 shows the components of the blackboard pattern. The blackboard is the central data store of the MAS. Data elements of the blackboard are application data (like messages, information, and so on) and control data (or meta-data). The blackboard provides an interface that enables all agents to read from, remove (take) from and write data to it. Agents use these operations to communicate indirectly with each other, and coordinate their activities. Agents use effectors to issue operations on the blackboard, and use sensors to perceive changes in the blackboard (for simplicity, we overlooked sensors and effectors in Fig 2).

Although the blackboard structure has proven itself to be a proper communication interface between software agents, it lacks a more precise specification of its controller component. The control component proposed in [3,14] is simply defined as a loop that monitors the changes on the blackboard and decides what action to take next. However, real-life MASs encompass a number of application-dependent and -independent control policies used to manage various system-wide properties, like mobility, communication, coordination, and security. The problem is that the main liability of the blackboard pattern is the difficulty of dealing with multiple control strategies in large MASs [3]; the pattern does not provide architectural support for handling several control strategies separately. Finally, the blackboard pattern does not provide separation between application data and control data; the controller component is responsible for storing both kinds of data. However, access to control information should be prohibited to some agents.

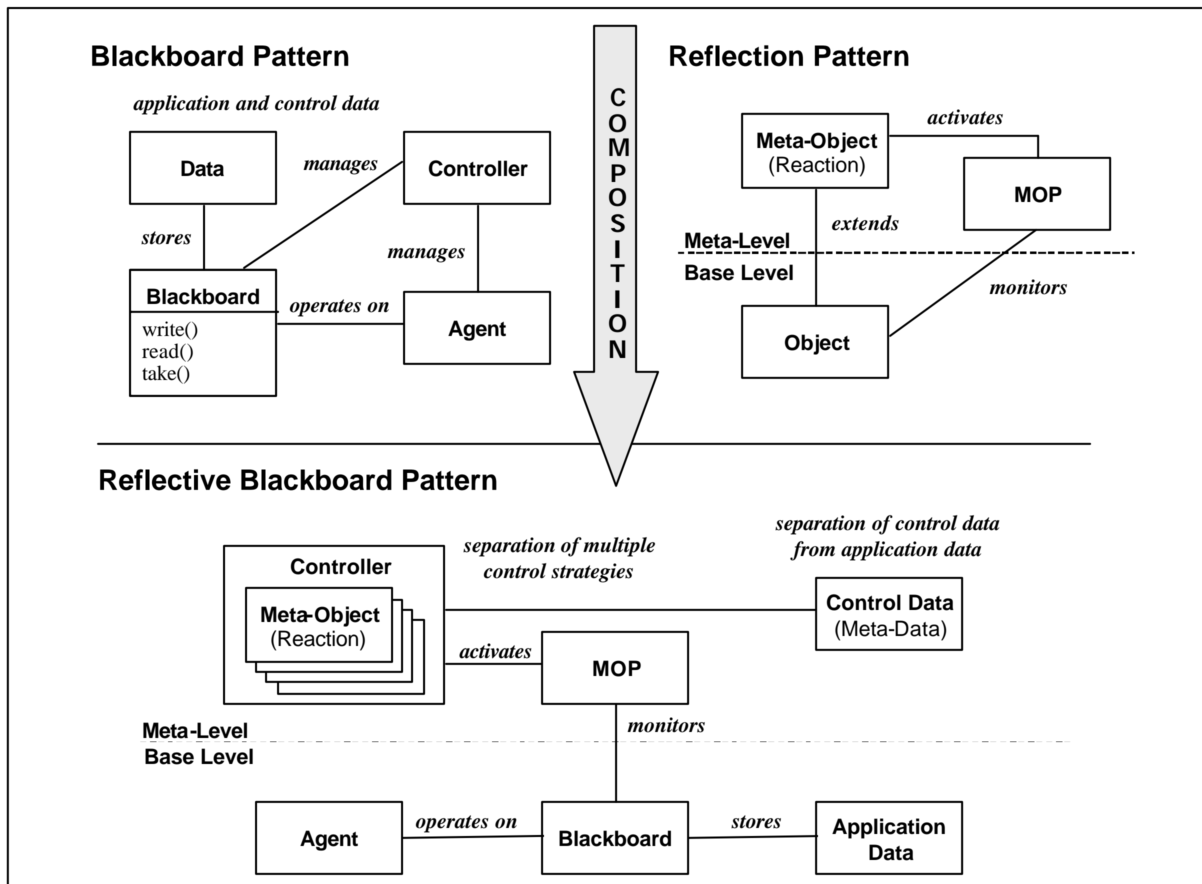


**Fig 2** The Composition of the Blackboard Pattern and the Reflection Pattern

*As far as the motivation example (Section 2.1) is concerned, the problem stated above is related to the difficulty associated with the definition of the mobile communication strategy in a way that is transparent to the buyer and seller agents. During negotiation processes, agents are moving across distinct marketplaces and should not keep control of their negotiation partners' location. In addition, the use of the blackboard pattern amalgamates control data – e.g. data informing about the agents' actual location - and application data – e.g. representing bids, offers, proposals and counterproposals. In addition, the pattern does not support the separate handling of mobility, reliability, management and security policies for the marketplace application.*

There are some *forces* associated with this problem:

- Control policies for some system properties are usually different in distinct execution environments. So the software architecture must be sufficiently flexible to enable adaptation to changes in the underlying environments, as well as to changes in application requirements related to control policies.
- MAS architectures must have a high degree of modifiability, i.e. facilitate the incorporation of changes once the nature of the desired change has been determined. In addition, the software architecture must support exchange, addition or removal of control strategies at run-time.

- The MAS architecture should guide the designer and the programmer on reusability of strategies across different projects when numerous control strategies are used.

### 2.3 Solution

We propose the composition of the blackboard architectural pattern with the reflection architectural pattern [3] to solve the problem stated in the previous section. The reflection architectural pattern provides a mechanism for changing the structure and behavior of a system dynamically [3]. The right upper side of Fig 2 illustrates the reflection pattern, which divides software systems into two different levels: base level and meta-level. The *base level* contains the application logic, which is implemented by *agents*; the *meta-level* is composed of *meta-objects*, which encapsulates data and behavior. Meta-objects' data is called metadata (or control data) that represent information about application data stored in the base level, while its associated behavior may be understood as the reaction to changes performed at the base-level [12]. The interface between the base-level and the meta-level is provided by a separate component called *Meta-Object Protocol* (MOP). The MOP is responsible for redirecting the control flow at the base-level to the meta-level in the execution points of certain systems.

The proposed composition results in three major changes to the blackboard solution: (i) the controller component and control data (metadata) are moved to the meta-level, (ii) the MOP intercepts the blackboard operations transparently, (iii) the controller semantics is distributed into separate meta-objects (i.e, reactions and metadata). According to these changes, data written in the blackboard may be associated with meta-objects located in the system's meta-level. The meta-objects behave like rules, which state how the system should behave when specific operations are performed in the blackboard. For example, a meta-object may specify that whenever a specific piece of data is taken out of the blackboard, the agent that wrote it will be notified of this data removal. In this way, the control of the agent communication, which is performed in the blackboard, allows us to inject system-wide properties transparently at the meta-level.

*The application of this solution to the marketplace example allows the mobile communication strategy be implemented at the meta-level controller, separated from the buyer and seller agents that are located at the base level. This is done by creating, meta-objects on the meta-level that specify that message forwarding strategies are created whenever an agent moves from one environment to another. These message-forwarding strategies are responsible for forwarding messages addressed to agents that have left their "home" marketplaces, to their destination marketplaces. The message pointers also are implemented as meta-level rules that state that every message addressed to the agent to which they are related is redirected to the destination environment. This control strategy is based on the same idea proposed by the mobile IP protocol [17], where data sent to mobile devices are always addressed to their home environment (home agent), which is responsible for forwarding the data to the environment where the device actually is. More details about the dynamics and implementation of this control strategy will be provided in the following sections.*

## 2.4 Structure

As it happens in the blackboard pattern the structure of the reflective blackboard pattern can be divided, as well, into three different subsystems: the blackboard itself, a group of knowledge sources and a controller. Fig 3 illustrates, using a UML component diagram [2], these subsystems, their main components as well as their dependencies. The blackboard behavior is almost the same as proposed in the Blackboard pattern. It is the central data storage structure where pieces of data are written, read or deleted by software agents. The main difference now is that every piece of data can be associated to meta-objects that are used in the controller component.

The controller subsystem is composed of a meta-object protocol (MOP) component that together with a collection of meta-objects implement the multi-agent system control strategies. Meta-objects are composed of data (metadata) and are responsible for associating specific behavior (reactions) to operations performed over specific pieces of data. These meta-objects can transparently modify the normal behavior of the blackboard, thus implementing the multi-agent system control strategies. Different agents can act over the blackboard by means of their sensors and effectuators, which can respectively sense and perform changes in the blackboard that can be considered their environment. The agents do not communicate directly; they only write and read data from the blackboard.

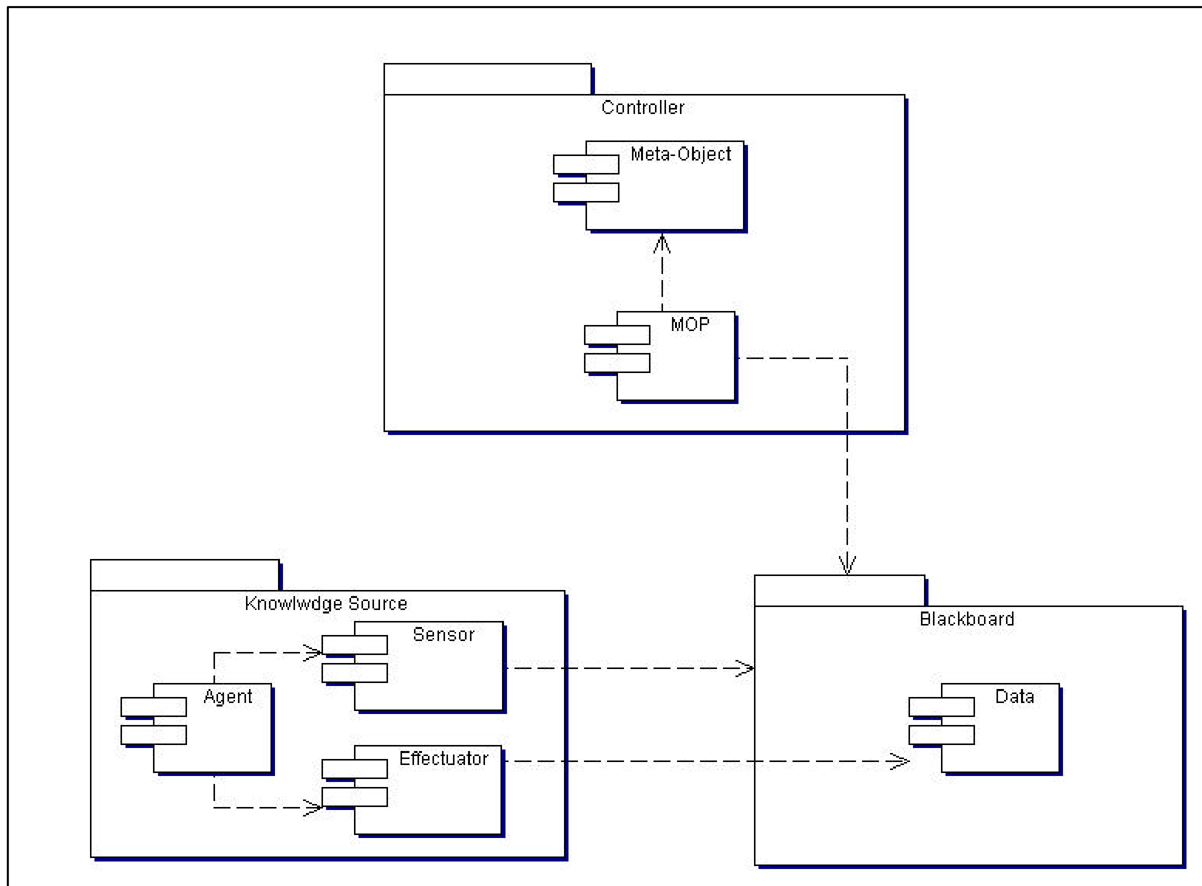


**Fig 3** The Reflective Blackboard Pattern Structure

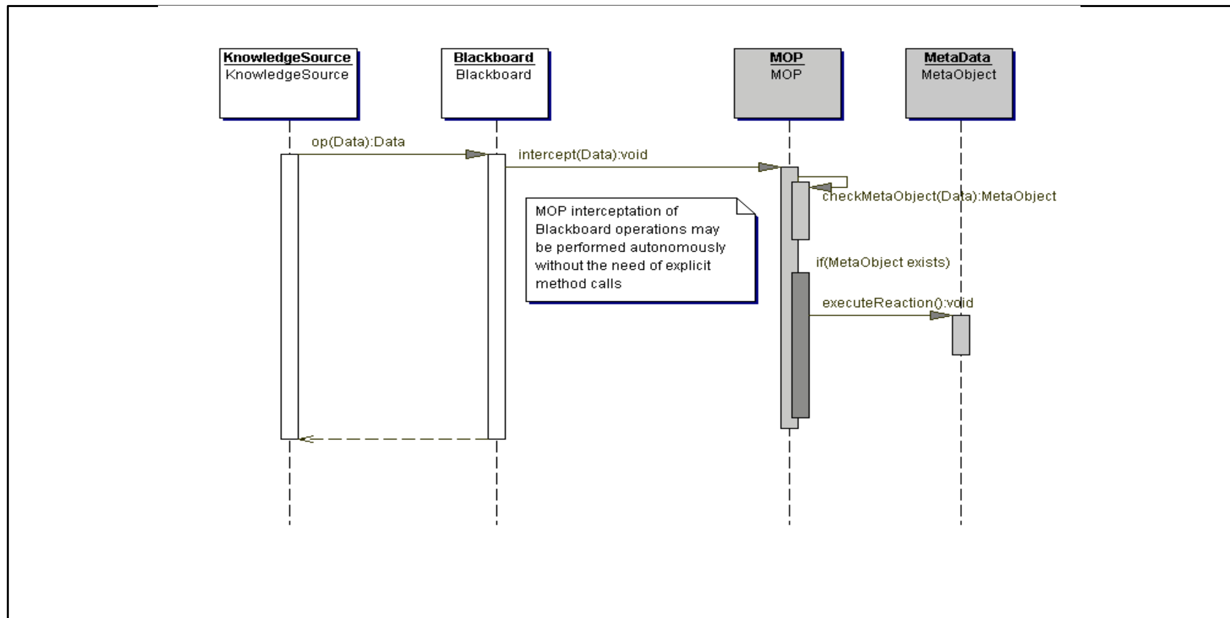


**Fig 4** Reflective Blackboard dynamics

Whenever an agent performs any operation over a specific piece of data stored at blackboard, the MOP component verifies if there is any meta-object associated to it. If positive it executes the reaction associated to the meta-object, i.e. its behavior. The meta-object execution can access the blackboard writing and deleting data. In this way, in a reflective blackboard architecture the semantics of a blackboard operation, in fact, is the result of the execution of the meta-objects associated to it. Meta-objects also may exist in the control subsystem without correspondent data in the blackboard. In this way the multi-agent system can associate reactions to data that is part of the multi-agent system vocabulary and probably will be written in the blackboard at runtime.

## 2.5 Dynamics

Reflection is used to intercept and modify the effects of operations of the blackboard. From the point of view of application agents, computational reflection is transparent: an agent writes a piece of data on the blackboard, and has no knowledge this write operation has been intercepted and redirected to the meta-level. The following scenario illustrates the general behavior of the Reflective Blackboard architecture:

1. A knowledge source (or agent) performs an operation on the blackboard (write for example), supplying a piece of data and expecting some other piece of retrieved data;
2. This operation is intercepted by the meta-level's MOP, which will perform, if specified, control activities over the performed operation;
3. The MOP checks for the existence of meta-objects associated to the blackboard data and related to the performed operation. If the knowledge source has performed a write operation on the blackboard, the searched meta-objects will be those related to the written piece of data. On the other hand, if the knowledge source has performed read or delete operations, the searched meta-object will be related to the piece of data read from the blackboard;

4. If the searched meta-object exists, its behavior (i.e., its reaction) is executed. The possible effects of the Reaction must be specified by the implementation of the Reflective Blackboard pattern (section 3.3). Depending on the implementation, the reaction can modify blackboard data, activating other knowledge sources among other types of control activities.

Fig 4 uses a UML [2] sequence diagram to visually illustrate this scenario.

*Concerning the motivation example presented in Section 2.1, this scenario can be specialized into two different ones. The first refers to the update of the message forwarding strategy while the second refers to the message forwarding strategy itself. The message forwarding strategy update scenario starts when Agent1 decides to move to another host and notifies its home environment, represented by* BlackboardA*, that it is going to leave. This notification is represented by the* MobileAgentData *that is written in the blackboard. The write operation is intercepted by the* MOP, which *checks the existence of any meta-object associated with the* MobileAgentData*. If such meta-object exists, in fact it will be responsible for updating the message forwarding strategy as specified in Section 2.3. In this way, the reaction (i.e. behavior) associated to this meta-object is responsible for creating a new message forwarding strategy and consequently notifying the* MOP *that a new meta-object exists. At this point, the meta-level operation ends and* Agent1 *can actually move to its destination environment. This scenario is represented in Fig 5 using an UML sequence diagram.*

*After the message pointer is updated, every message addressed to* Agent1 *will be forwarded to its new environment. In the motivation example scenario* Agent2 *sends a message, addressed to* Agent1*, to* BlackboardA*. This process is represented by the operation* write *performed by* Agent2 *over* BlackboardA*. This*

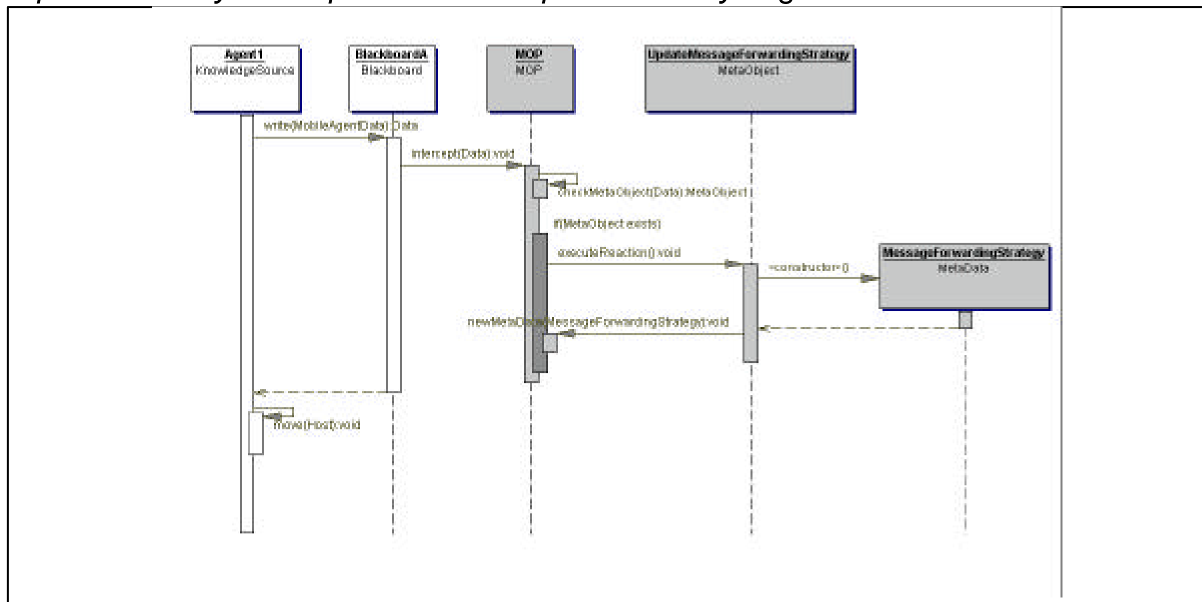


**Fig 5** Dynamics for updating the message forwarding strategy

*operation is intercepted by the MOP, which will check the existence of any meta-object associated to the written message. Such meta-object is in fact the message forwarding strategy that was created in the scenario presented above. This meta-data is responsible to associate a reaction, responsible to the message forwarding process, to messages addressed to* Agent1. *If the searched meta-object exists on the meta-level its reaction will be executed. The reaction execution will remove (take) the message from* BlackboardA *and write it on* BlackboardB. *After the reaction execution, the meta-level operation ends and Agent1 can read the message from* BlackboardB. *This scenario is represented in Fig 6 using an UML sequence diagram.*

## 2.6 Consequences

The Reflective Blackboard architectural pattern promotes the following **benefits**:

*Separate handling of control concerns.* The use of reflective blackboard architectures to develop MASs promotes the separation of their control policies from their basic functionality. In addition, it separates application data from control data. These kinds of separation enable the smooth handling of different control aspects of the system. Moreover, the different control strategies are composed independently from the application at the meta-level. The application developers focus their attention on the intra-agent concerns at the base level. This is particularly important when a large MAS is involved since it is often composed of organized societies of agents, with each particular society having different, very complex control policies. These policies are difficult to handle if they are tangled with system basic data and functionality.

*Improved reusability and maintainability.* Agents' code is not intermingled with explicit invocations of control strategies. The MOP does these invocations in a way that is transparent to the application functionality. As a consequence, it improves readability, which in turn promotes reusability and maintainability. Reuse and maintenance also are improved due to the separate incorporation of control strategies. Different applications demand different implementations of control strategies. So reuse of the application logic (i.e. the agents) can be gathered, since such control strategies are implemented at the meta-level. In this way, the separation of concerns achieves

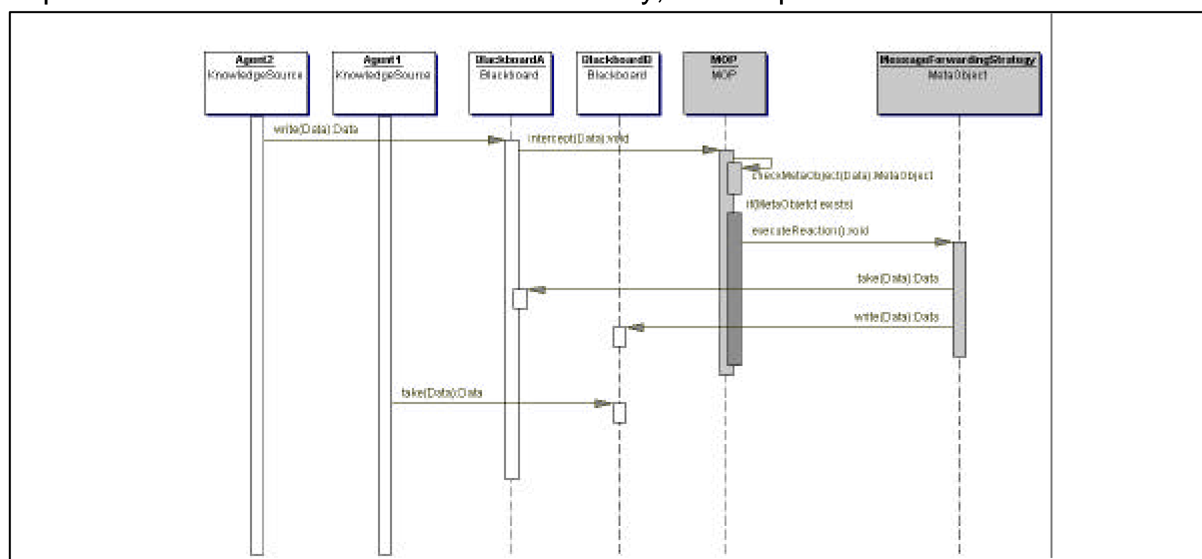


**Fig 6** Dynamics for the message forwarding strategy

reuse at different levels: the agent level, the control-strategy level and the systemic-property level.

*Improved writeability.* Architectures of large MASs often comprise isolated agents and organizations of independently designed agents. The presence of the MOP and meta-objects allows writing and associating code of control strategies with various levels of an MAS, e.g. the agent-level, the organization-level and the system-level. In this way, the complexity of MAS can be controlled in a flexible and systematic manner, and control strategies can be added at the levels where they are needed. However, care should be taken while improving the power of the meta-level and meta-information. Unnecessary expressive power may complicate both using the proposed architecture and understanding of the MAS code, increasing the probability of error introductions and making the testing phase more difficult.

*Acceleration of the MAS development process.* In complex systems, the process is likely to involve several software engineers, and a good separation of concerns contributes decisively to acceleration of the development process by paralleling the development of different architectural components and the handling of different system aspects. The proposed pattern enables engineers of multi-agent software to work separately on the abstraction levels of different systems . *Meta-level software engineers* decide how to refine the meta-level components to incorporate and compose the system's control policies, and *base-level software engineers* are concerned only with the internal architecture of agents and its basic functionality.

*Dynamic Reconfiguration*. Distributed multi-agent applications typically have dynamic systemic requirements that need more complex algorithms. The pattern defines an approach that supplements standard blackboard architectures with a general reflective mechanism for injecting control activities dynamically into the communications between software agents. So dynamic reconfigurability is achieved through the extensive use of reflection since the meta-level comprises reflective facilities to expose the structure and behavior of MAS components to the meta-level engineers, enabling dynamic inspection and adaptation. Algorithms that support systemic requirements are separated from functional components but may be invoked whenever agents communicate using the blackboard. Since the MOP component provides an interface to change the application behavior dynamically, meta-level engineers can reconfigure the meta-level to inject new control policies, remove existing ones, and decide which policy should be enforced in a given system's execution point at run-time.

On the other hand, using a Reflective Blackboard architecture has some **liabilities**:

*Performance overhead.* A possible disadvantage of this pattern is that reflective architectures are usually slower than non-reflective architectures. This problem occurs because of the additional computation that is needed to change dynamically control flow from the base level to the meta-level and to activate meta-objects responsible for implementing control activities.

## 2.7 Known Uses

Tuplespace architectures are a classic implementation of blackboard architectures. **TSpaces** [11] is a well-known tuplespace architecture that implements the Reflective

Blackboard pattern. TSpaces is a Linda-like blackboard architecture for network communication with database capabilities. It provides group communication services, database services, and event notification services. The TSpaces Event notification engine plays the role of the MOP component of the Reflective Blackboard pattern. TSpaces reactions are called callback objects and TSpaces meta-data contains information about the operation and the data monitored by the event engine. When implementing an MAS, the TSpaces event monitoring services are used to establish control strategies. MASs implement this by registering events that notify agents that relevant data has been written in the blackboard. Since agents are notified of a specific event, the associated control strategy is performed.

**MARS** [4] is another implementation of the proposed pattern. It defines Linda-like blackboards, which can be programmed to react with specific actions to the accesses made by agents. MARS is implemented using the JavaSpaces [5] technology. MARS was created to help in the task of defining coordination strategies in mobile agents applications. MARS meta-data are called meta-tuples and contain information about the agent that performs a specific operation over specific pieces of data. The MOP protocol is implemented using template-matching searches on a meta-level blackboard where meta-data is stored.

**TuCSoN** [18] is a coordination model that can be thought of as an implementation of the Reflective Blackboard pattern. This model is based on the notion of tuple centers, which are in fact programmable blackboards. Tuple centers are programmed by associating reactions to specific data and operations. Reactions are created using a proprietary specification language and are handled separately from application basic logic and data.

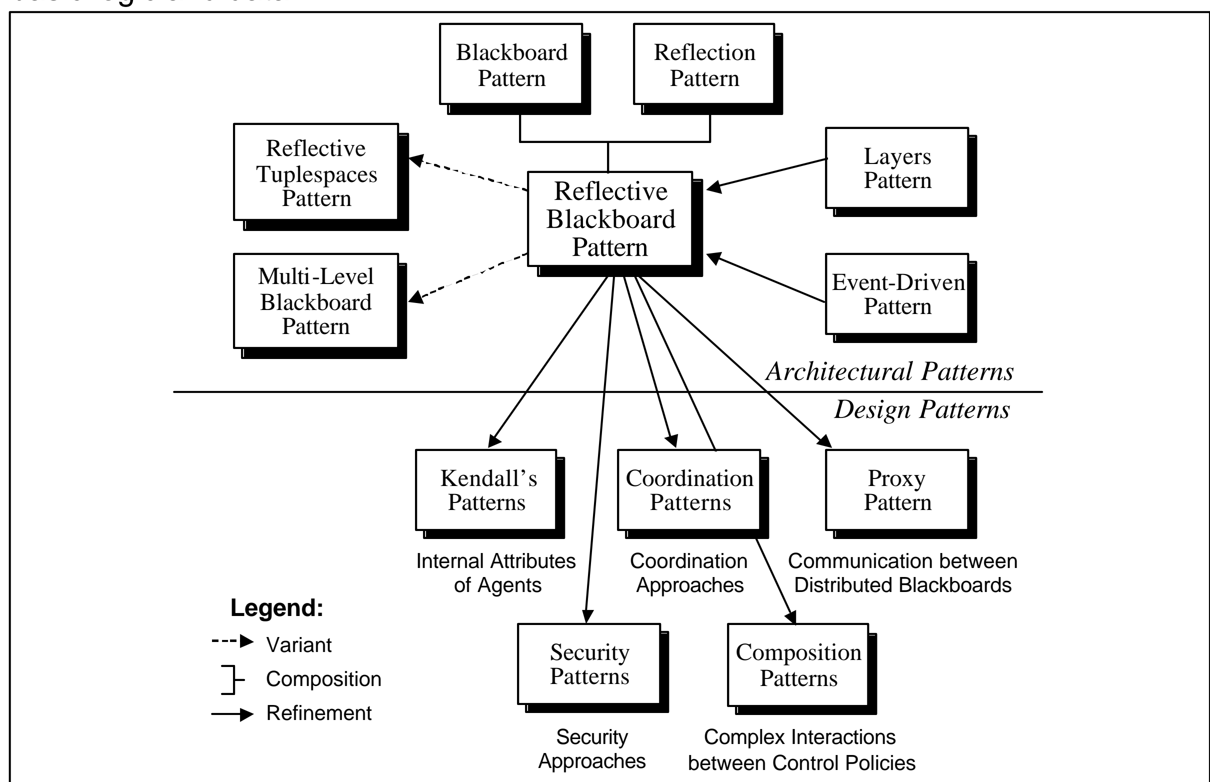


**Fig 7** The Reflective Blackboard Pattern and its Related Patterns

## 3 Reflective Blackboards and the Development of Large MASs

The Reflective Blackboard pattern provides, during the architectural design stage, the context in which more detailed design decisions related to system-level properties are made in later MAS development stages. Thus, this section builds up the overall picture; it discusses how meta-level and base-level engineers proceed from the architectural phase to the design and implementation phases of MAS construction. Since the proposed architecture has been chosen, MAS engineers must describe how multiple control policies are introduced into the system (Section 3.1), how the reflective blackboard pattern is connected with other related patterns that cover additional aspects of MAS development (Section 3.2) and how the components of the pattern can be implemented (Section 3.3).

### 3.1 Achieving Multiple Control Strategies

Large-scale MASs are driven by multiple, complex control strategies that encompass system-level properties and are not part of an application's basic functionality. This section illustrates how introducing some particular system-wide properties into MASs based on the reflective blackboard solution, which is the structural foundation upon which more detailed pattern languages for systemic properties can be based. We illustrate the benefits of the proposed pattern to inject in the marketplace application (Section 2.1) of typical systemic properties, such as coordination activities, security policies and management strategies.

*Coordination.* Coordination, which is defined as the management of dependencies between agents in order to foster harmonious interaction between them [34], is indispensable for effective cooperation between autonomous agents, as well as for safe competition between them [23]. With regard to the marketplace example, coordination strategies are needed in several contexts, e.g. in the case that multiple buyers eventually join up with each other to buy products together and minimize costs. Coordination activities include accessing the bids of a marketplace and communicating and synchronizing with cooperating mobile agents that are visiting other distributed marketplaces in order to find the best price proposal. In a complex open system, coordination activities encompass application-dependent strategies – related to the specific roles of the application agents – and application-independent ones – related to the interaction of the agents with the other agents of the same application and with the visited execution environments – which should be separated [39, 26]. The Reflective Blackboard pattern clearly supports separating the application-dependent coordination activities (base level) and the application-independent ones (meta-level). Hayden et al propose a system of patterns for multi-agent coordination [28].

*Security.* Security involves confidentiality and integrity factors and is primarily a combination of policies for access control, intrusion detection, authentication and encryption [1, 9]. Each of these policies traditionally are implemented by controlling the communication process that involves the application components. The use of a reflective blackboard architecture allows us to incorporate easily such security policies since the agent communication is centralized on the blackboard. Meta-level engineers use meta-objects to implement each security policy and the MOP to intercept operations issued on the blackboard in order to activate these meta-objects.

In the marketplace example, security is a fundamental requirement since the marketplace is open. So meta-objects are implemented to control agents joining or leaving the marketplace and to encrypt communications, reliably sending user authentication from marketplace to marketplace (and pass if along to dependent requests), and to check the access rights of mobile agent requests. All this is independent of the actual application code. Yoshioka et al. [42] propose a system of patterns to implement security policies that can be combined with the Reflective Blackboard pattern.

*Manageability*. Manageability includes administrative activities such as accounting, logging, configuration management, performance measurement, report generation and so forth. In the marketplace case, administrators usually need to obtain information about transactions performed in their marketplaces, as well as information about visiting agents that join and leave them. The reflective blackboard architecture supports means of analyzing the activities of the marketplace since all transactions are conducted upon the blackboard. The MOP is used to intercept operations of transactions and meta-objects are used to process the information associated with such transactions and generate logging files and reports.

*Composition of Multiple Systemic Properties.* The proposed pattern allows system-level properties and strategies be entirely implemented separately as meta-objects. However, some system-level properties are naturally interactive. In practice, because they occur concurrently in distributed systems, multiple policies can interfere with each other. For example, many replication strategies require logging and distributed updates on every agent and blackboard modification and security policies often constrain coordination activities. When composition conflicts are not managed properly, it is likely to cause deadlocks, livelocks, dangling resources, inconsistencies, and incorrect execution semantics. One approach to dealing with interference during strategy composition in an MAS is using composition patterns. Composition patterns, such as the Mediator pattern [41] and the Chain of Responsibility pattern [41] provide a means of allowing safe integration of interactive properties at the meta-level. The Mediator pattern, for instance, defines an object that encapsulates how a set of objects interact; this solution promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

### 3.2 Architectural Refinements and Design Decisions

Our pattern is the basis for the composition of multiple known patterns during the refinement of complex multi-agent software architectures. The previous section discussed how meta-level engineers incorporate specific properties using the proposed pattern. This section discusses related architectural and design patterns, methods and guidelines that help with taking design decisions and refining the basic architecture of reflective blackboards. Most important, this section shows how the Reflective Blackboard pattern is connected with other patterns, with which other patterns it can be refined and combined, which variants it exposes and which other patterns solve the same problem in a different way. Fig 7 illustrates the interconnections of the Reflective Blackboard pattern with other architectural and design patterns.

*Internal Architecture of Agents (Base Level).* The architecture of a single agent is very complex since it encapsulates a mental state and a number of behavioral features, such as autonomy, adaptation, collaboration and learning. Kendall et al. [10] examine design patterns for agents with a layered architecture. They illustrate patterns applicable to each layer constructing the agents. Garcia et al. propose an aspect-oriented method to structure the internal design of software agents [7] and compare it with a pattern-oriented method [22].

*Reflective Tuplespaces (Base Level).* In this variant, the blackboard component of the proposed pattern is structured as tuplespaces, which are shared, associatively addressed memory spaces that are composed of a bag of tuples. Tuplespace architectures originate from the Linda project at Yale University [40]. Being a global memory, tuplespace architectures are often characterized as special kinds of blackboard architectures. The meta-level components are structured as tuples, stored in meta-level tuplespaces. TSpaces, MARS, TuCSoN, and T-Rex, the known uses presented in Section 2.7, implement this variant of the proposed pattern. The next section shows how to implement this variant of the proposed pattern.

*Event-Driven Blackboard (Base Level).* The Reflective Blackboard pattern can be combined with the Event-Driven architectural pattern [14]. An event model is used to signal when changes are made to the blackboard and to notify the agents that something changed. An event could trigger the activation of a set of agents or the controller could dynamically determine which agent to start. In addition, the meta-level could activate a control strategy based on a specific event.

*Meta-Level Organization (Meta-Level).* The general structure of a reflective architecture is very much like the Layers architectural pattern [3, 14]. The metal-level and base level are two layers, each of which provides its own components. However, in contrast to a layered architecture, there are mutual dependencies between both layers. The base level builds on the meta-level, and vice-versa. An example of the latter occurs when meta-objects implement behavior that is executed in case of an exception. The kind of exception handler that must be executed often depends on the current state of computation. In a pure layered architecture, these bi-directional dependencies between layers are not allowed. Every layer only builds upon the layers below. Another issue is that the meta-level of the proposed pattern can use the structure of the Layers pattern to refine its meta-level in multiple meta-levels, leading to a variant termed *Multi-level Blackboards* (Fig 7). This variant is composed of a tower of meta-levels, where each level incorporates different control levels.

*Distributed Blackboards (Meta-Level).* The meta-levels of different blackboards may have to communicate with each other in order to implement a given systemic property. The Proxy design pattern [41] is a solution for remote communication. The proxy pattern provides a surrogate or placeholder for another object to control access to it. Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. A remote proxy provides a local representative for an object in a different address space, and hides the fact that an object resides in a different address space. A protection proxy controls access to the original object, which is useful when objects should have different access rights.

### 3.3 Implementation Issues

The software architecture proposed in this paper has been identified and developed based on our extensive work implementing the T-Rex framework [15,16] and other reflective architectures [15, 20, 21], and on our study of a number of related implementation architectures [4,11,18]. Since the proposed pattern is independent of programming languages or implementation architectures, a wide range of MAS developers can employ it. This Section points out issues that software engineers should consider to implement the proposed architectural pattern, and the procedure by which such issues are realized in the T-Rex framework:

**Step 1: How will meta-objects be structured?** A fundamental issue is deciding which meta-information will be available in the meta-objects components. In fact this decision depends on the specificity of the control implemented by the multi-agent system. Common meta-objects contain references to the base level data, agent identification and blackboard operation [4,16]. However it can also contain more application-specific information such as the hypothesis level of abstraction and degree of certainty. In the T-Rex framework's implementation meta-objects are implemented through meta-tuples [15,16], which associate a specific **reaction** to a given **operation** performed by an **agent** over a piece of **data** stored on the blackboard. In this way, the meta-objects' meta-information are 4-tuples that have the following strucuture: (reaction, operation, agent, data).

**Step 2: How will the MOP be implemented?** Another important issue is deciding how the meta-object protocol will act over meta-objects. A possible implementation is to use another blackboard to store meta-objects, and thus the MOP will use standard blackboard operations to write and search for meta-objects. Using Linda-like tuple spaces [6] in this "meta-level blackboard" implementation can help this task since it is useful to use template match searches while looking for meta-data. By using another approach, one could also use native reflective architectures such as Guaraná [25] to implement meta-level activities. The MOP in T-Rex is implemented through reflective tuple spaces where any operation executed over the base-level blackboard is intercepted. After this interception the control is deviated to the meta-level and meta-objects associated to the performed operations are searched. If there exists any associated meta-object, its associated reaction is executed.

**Step 3: Which components will be able to access the meta-level?** It is important to establish the access policies to the multi-agent system's meta-level, where its control strategy will be implemented. It can be defined that the meta-data is written only in the implementation phase and remains unchanged at runtime. On the other hand, the system administrator and even agents can insert meta-data at runtime. The adoption of this access policy may require special attention to the meta-level control and implementing a meta-meta-level could be useful to deny harmful changes to the control component. The T-Rex framework does not provide access restrictions to the meta-level. In this way meta-objects can be created or deleted at runtime, by the MAS administrator or even by the agents that belong to the system. Specific implementations of the framework are able to use meta-meta-level tuple spaces to enable access control policies on the meta-level, following the same idea proposed by the Reflective Blackboard pattern.

**Step 4: Which components will the reactions be able to modify?** Reactions are components that specify changes in the normal system's behavior. It is an important decision establishing which components they can access. In general, by accessing the multi-agent systems' blackboard it is possible to change the systems' overall behavior, since it will change the agents' environment and thus be acting directly upon the agents' sensors. On the other hand, to implement some control strategies it might be necessary to allow reactions to access the other system specific components, such as the operational system functions file system or the network. In the T-Rex framework there are not access restrictions to reactions. These restrictions should be implemented on the applications that are developed using the reflective infrastructure provided by T-Rex.

## 4 Conclusion and Ongoing Work

Agent technology has been revisited as a complementary approach to the object paradigm, and has been applied in a wide range of realistic application domains. The inclusion and composition of system-level properties and their control strategies into an MAS is one of the major sources of software complexity. For complex MASs, most code is not devoted to implementing the desired functional behavior, in terms of its agents, but rather to providing system-wide properties (and their control strategies) like coordination, security, reliability and manageability. Hence, many ongoing investigations are concerned with mastering this software complexity by means of effective software engineering techniques in order to enhance system reusability, maintainability and stability. Patterns are a useful technique for MAS engineers since they capture existing, well-proven experience in software development and help to promote good design practice.

This article describes an architectural pattern that enables a more complete separation of systemic properties implementation from agent functionality, allowing these properties to be developed, maintained, and modified with minimal impact on agent implementations. The Reflective Blackboard pattern is quite useful when developing multi-agent systems with huge numbers of agents and whose control strategies are very complex. Its use can minimize development efforts since it promotes a better separation of concerns. The base level specifies the interface for exploiting application functionality. The meta-level layer defines the MOP to modify the control strategies, implemented by meta-objects. The basic idea of this pattern is that since communication and coordination, the basic properties in MAS, are centralized on the blackboard, we may easily control it and insert new systemic properties in the desired points to the agents' functionality in a largely transparent way. This pattern provides a loosely coupled meta-level controller. This component is handled separately, keeping control aspects independent from the functional aspects of an MAS and, consequently, improves its maintainability and reusability. Our pattern is the basis for the composition of multiple known patterns during the construction of complex MASs. In this sense, this paper also discussed how the proposed pattern is integrated with other known patterns, enabling the effective use of reflective blackboard in MAS development.

As future work, we are planing to create domain specific patterns for each internal feature of an agent and for specific system-wide properties, investigate their

ability and scalability and accumulate more practical know-how to construct a pattern language. Section 3 described some guidelines to deal with different systemic properties based on a reflective blackboard. However, we need to conduct some case studies and experiments to understand better how traditional software concerns (like privacy and dependability) and MAS-specific concerns (like coordination and emergent behavior) manifest and interact with each other during different MAS development stages. Up to now, we developed a first empirical study [22] to understand how the internal concerns of agents interact with each other and can be explicitly separated during software lifecycle phases, using a pattern-oriented method [22] and an aspect-oriented method [43].

**Acknowledgements.** This work has been partially supported by CAPES for Otavio and CNPq under grant No. 141457/2000-7 for Alessandro, and by FAPERJ under grant No. E-26/150.699/2002 for Alessandro. Otavio, Alessandro and Carlos are also supported by the PRONEX Project under grant 7697102900. We would also like to thank the anonymous referees for the good suggestions during this work.

## References

1. M. Barbacci. “Quality Attributes”. Technical Report, CMU/SEI-95-TR-021, December, 1995.
2. G. Booch, J. Rumbaugh. “Unified Modeling Language – User Guide”. Addison-Wesley, 1999.
3. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. “Pattern-Oriented Software Architecture: A System of Patterns”, John Wiley & Sons, 1996.
4. G. Cabri, L. Leonardi, F. Zambornelli, “MARS: A Programmable Coordination Architecture for Mobile Agents”, IEEE Internet Computing, Vol. 4, No. 4, pp. 26-35, July-August 2000.
5. E. Freeman, S. Hupfer, K. Arnold, “JavaSpaces(TM) Principles, Patterns and Practice”, Addison-Wesley Pub Co, 1999
6. D. Galernter, “Generative Communication in Linda” ACM Transactions on Programming Languages and Systems, vol. 7 - No.1, pp 80-112, 1985
7. A. Garcia, C. Chavez, O. Silva, V. Silva, C. Lucena. “Promoting Advanced Separation of Concerns in Intra-Agent and Inter-Agent Software Engineering”. Workshop on Advanced Separation of Concerns in Object-oriented Systems (ASoC) at OOPSLA'2001, Tampa Bay, USA, October 2001
8. M. Huhns, L. Stephens. “Multiagent Systems and Societies of Agents”, in *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*, ed. G. Weiss, MIT Press, 2000
9. Institute of Electrical and Electronics Engineers. “IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries”. New York, 1990.
10. E. Kendall, P. Krishna, C. Pathak, C. Suresh, “A Framework for Agent Systems”, in *Implementing Applications Frameworks: Object Oriented Frameworks at Work*, ed. M. Fayad, D. Schmidt, R. Johnson, John Wiley & Sons, 1999.
11. T. Lehman, S. McLaughry, P. Wyckoff. “TSpaces: The Next Wave”. Hawaii International Conference on System Sciences (HICSS-32), January, 1999.
12. P. Maes. “Concepts and Experiments in Computational Reflection”. ACM SIGPLAN Notices, 22(12), pp 147-155, 1987

13. Object Management Group – Agent Platform Special Interest Group. “Agent Technology – Green Paper – Version 1.0”, OMG, September 2000.
14. M. Shaw, D. Garlan. “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, 1996.
15. O. Silva, A. Garcia, C. Lucena, “A Unified Software Architecture for System-Level and Agent-Level Dependability in Multi-Agent Object-Oriented Systems”, 7th ECOOP Workshop on Mobile Objects Systems, Budapest, Hungary, June 2001
16. O. Silva, A. Garcia, C. Lucena, “T-Rex: A Reflective Tuple Space Environment for Dependable Mobile Agent Systems”. III WCSF at IEEE MWCN 2001, Recife, Brasil, August 2001
17. C. E. Perkins, "Mobile IP," IEEE Communications Magazine, vol. 35, no. 5, pp. 84-99, May 1997
18. A. Omicini, F. Zambonelli, “TuCSoN: a Coordination Model for Mobile Information Agents”. 1st International Workshop on Innovative Internet Information Systems (IIIS'98), Pisa (I), June 1998
19. L. Ferreira, C. Rubira, “The Reflective State Pattern”. Proceedings of the 5th Patterns Languages of Programs Conference (PLoP’98), August 98, Monticello, EUA.
20. A. Garcia, C. Rubira. “A Architectural-based Reflective Approach to Incorporating Exception Handling into Dependable Software”. In: A. Romanovsky et al (Eds). "Advances in Exception Handling Techniques". Springer-Verlag, LNCS-2022, April 2001, pp. 189-206.
21. A. Garcia, C. Rubira. “A Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling”. The Computer Journal, Special Issue on High Assurance Systems Engineering, January 2002.
22. A. Garcia, V. Silva, C. Chavez, C. Lucena. “Engineering Multi-Agent Systems with Aspects and Patterns”. Journal of the Brazilian Computer Society, Special Issue on Software Engineering and Databases, August 2002.
23. N. Minsky, V. Ungureanu. “Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems”. ACM Transactions on Software Engineering and Methodology, Vol. 9, No. 3, July 2000, pp. 273-305.
24. N. Karnik, A. Triphathi, “Security in the Ajanta Mobile Agent System”, Software - Practice and Experience, January 2001.
25. A. Oliva, I.Garcia, L.Buzato, “The reflexive architecture of Guaraná”. Technical Report IC-98-14, Institute of Computing, State University of Campinas, April 1998
26. F. Zambonelli, N. Jennings, M. Wooldridge. “Organizational Abstractions for the Analysis and Design of Multi-Agent Systems”. In Proc. of the 1st International Workshop on Agent-Oriented Software Engineering at ICSE 2000, Limerick (IR), June 2000.
27. L. Erman, F. Hayes-Roth, V. Lesser, D. Reddy. “The HEARSAY-II speech-understanding system: Integrating knowledge to resolve uncertainty”. Computing Surveys 12(2): 213-253.
28. S. Hayden, C. Carrick, Q. Yang. “Architectural Design Patterns for Multiagent Coordination”. In Proceedings of the International Conference on Agent Systems '99 (Agents'99), Seattle, WA, May 1999.
29. A. Porto, G. Roman (Eds.). Coordination Languages and Models. Proc. of the 4th International Conference COORDINATION 2000, Limassol, September 2000. LNCS 1906, Springer.
30. M. Huget, F. Dignum, J. Koning (Eds.). Proc. of the Workshop on Agent Communication Languages and Conversation Policies. AAMAS 2002, Bologna, Italy, July 2002
32. N. Karnik, A. Tripathi. "Design Issues in Mobile-Agent Programming Systems". IEEE Concurrency, vol. 6, n. 3, 1998, pp.52-61.
33. E. Freeman, S. Hupfer, K. Arnold, “JavaSpaces(TM) Principles, Patterns and Practice”, Addison-Wesley Pub Co, June 1999
34. T. Malone, K. Crowston. “The Interdisciplinary Study of Coordination”. *ACM Computing Surveys 26*, 1 (March), 87-119.

35. J. Bailey, Y. Bakos. “An Exploratory Study of the Emerging Role of Electronic Intermediaries”. International Journal of Electronic Commerce 1(3), Spring 1997.
36. Y. Bakos. “The Emerging Role of Electronic Marketplaces on the Internet”. CACM, August 1998.
37. R. Guttman, A. Moukas, P. Maes. “Agent Mediated Electronic Commerce: A Survey”. Knowledge Engineering Review, June, 1998.
38. M. Tsvetovatyy, M. Gini, B. Mobasher, Z. Wieckowski. MAGMA: An Agent Based Virtual Market for Electronic Commerce. International Journal of Applied Artificial Intelligence, September 1997.
39. F. Zambonelli, G. Cabri, L. Leonardi. “Developing Mobile Agent Organizations: A Case Study in Digital Tourism”. Proceedings of the 3rd International Symposium on Distributed Objects & Applications (DOA) 2001, Rome (I), September 2001.
40. D. Gelernter. Generative Communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1), 1985.
42. H. Yoshioka, Y. Tahara, A. Ohsuga, S. Honiden. “Security for Mobile Agents”. In Proc. of the 1st International Workshop on Agent-Oriented Software Engineering at ICSE 2000, Limerick (IR), June 2000.
43. A. Garcia, V. Silva, C. Lucena, R. Milidiú. “An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems”. XXI Brazilian Symp. on Software Engineering, Rio de Janeiro, Brazil, October 2001, pp. 177-192.
44. E. Gamma et al. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley, Reading, 1995.