A Framework for Design Patterns in TROPOS

T. Tung Do, Manuel Kolp, T. T. Hang Hoang, and Alain Pirotte
Information Systems Research Unit, University of Louvain
Place des Doyens, 1, 1348, Louvain-la-Neuve, Belgium
{do,kolp,hoang}@isys.ucl.ac.be,pirotte@info.ucl.ac.be

Abstract

Multi-Agent Systems (MAS) architectures are gaining popularity over traditional ones for building open, distributed, and evolving software. Since the fundamental concepts of multi-agent systems are social and intentional rather than object, functional, or implementation-oriented, the design of MAS architectures should be eased by using what we call *social patterns* rather than object-oriented design patterns. Social patterns are idioms inspired by social and intentional characteristics used to design the details of a system architecture. The paper presents a framework called SKWYRL used to gain insight into social patterns and help design a MAS architecture in terms of these new idioms. The framework is integrated in the TROPOS agent methodology. It is developed according to the five modeling dimensions provided by TROPOS: social, intentional, structural, communicational, and dynamic. We consider the *Broker* social pattern as a combination of patterns and use it to illustrate the modeling dimensions of SKWYRL. A framework for code generation is also presented as well as an e-business broker module.

Key-words: Design Patterns, Multi Agent Systems, Tropos Methodology, Social Structures

1. Introduction

The explosive growth of application areas such as electronic commerce, knowledge management, peer-to-peer and mobile computing has profoundly changed our views on information systems engineering. Systems must now be based on open architectures that continuously evolve to accommodate new components and meet new requirements. These new requirements call, in turn, for new concepts and techniques for engineering and managing information systems. Therefore, Multi-Agent System (MAS) architectures are gaining popularity over traditional systems, including object-oriented ones.

Developing organizational information systems with a MAS architecture permits a better match between system architectures and their operational environment. A MAS can be seen as a *social organization* of autonomous software entities (agents) that can flexibly achieve agreed-upon *intentions* through their interactions. MASs support dynamic and evolving structures which can change at run-time to benefit from the capabilities of new system entities or replace obsolete ones.

Design patterns (see, e.g., [6]) have significantly contributed to the reuse of design experience and knowledge. Each pattern identifies a type of problem commonly encountered in software design, and it describes a reusable and flexible solution for it.

This paper focuses on *social patterns*. Taking real-world social behaviors as a metaphor, social patterns describe MAS as composed of autonomous agents that interact and coordinate to achieve their intentions, like actors as in human organizations.

This work continues the research in progress in the TROPOS project, whose aim is to construct and validate a software-development methodology for agent-based software systems. The TROPOS methodology [3] adopts ideas from MAS technologies and concepts

from requirements engineering, where agents and goals have been used heavily for organizational modeling. The key premise of TROPOS is that agents and goals can be used as fundamental concepts for analysis and design during *all the phases of the software development life cycle*, and not just requirements analysis.

TROPOS spans four phases of software development:

- Early requirements, concerned with the understanding of a problem by studying an organizational setting; the output is an organizational model which includes relevant actors, their goals and their interdependencies.
- Late requirements, where the system-to-be is described within its operational environment, along with relevant functions and qualities.
- Architectural design, where the system architecture is defined in terms of subsystems, interconnected through data, control, and dependencies.
- Detailed design, where the behavior of each architectural component is defined in further detail.

TROPOS also includes techniques for generating an agent implementation from a detailed design. Using an agent-oriented programming platform for the implementation is intuitive, given that the detailed design is defined in terms of (system) actors, goals and interdependencies among them.

We have overviewed in [10] a social ontology for TROPOS that considers software as (built of) social and intentional structures all along the development life cycle. The ontology considers organizational styles for architectural design and social patterns for detailed design. Organizational architectural styles for TROPOS have been further detailed in [9].

The present paper details the notion of social patterns for TROPOS. It focuses on the conceptualization of a framework called SKWYRL[1] that we have integrated in TROPOS. To do so, SKWYRL models the social patterns according to the five complementary dimensions proposed by TROPOS: social, intentional, structural, communicational, and dynamic. The framework facilitates the design of MAS architectures during the detailed design phase of TROPOS as well as the generation of code for agent implementation.

As an illustration, the paper studies a social pattern called *Broker*. We also introduce the generation of code from given social patterns into JACK [8], a JAVA agent-oriented development environment.

The paper is organized as follows. Section 2 introduces some major social patterns used in TROPOS. Section 3 proposes the SKWYRL framework, illustrates it through the Broker pattern, and overviews the code generation as well as an e-business broker example. Finally, Section 4 summarizes the results and points to further work.

2. Social Patterns

Considerable work has been done in software engineering on defining software patterns (see e.g., [6]). Still, little emphasis has been put on social and intentional aspects. Moreover, the proposals of agent patterns that address those aspects (see e.g., [1,4]) are not aimed at the design level, but rather at the implementation of lower-level issues like agent communication, information gathering, or connection setup.

In the following, we present patterns focusing on social and intentional aspects that are

[1]Socio-Intentional ArChitecture for Knowledge Systems and Requirements ELicitation (http://www.isys.ucl.ac.be/skwyrl/)

recurrent in multi-agent and cooperative systems. In particular, the structures are inspired by the federated patterns introduced in [7,9] and used in TROPOS . We have classified them in two categories. The *Pair* patterns describe direct interactions between negotiating agents. The *Mediation* patterns feature intermediate agents that help other agents reach agreement about an exchange of services.

2.1. Pair Patterns

The **Booking** pattern involves a client and a number of service providers. The client issues a request to book some resource from a service provider. The provider can accept the request, deny it, or propose to place the client on a waiting list, until the requested resource becomes available when some other client cancels a reservation.

The **Subscription** pattern involves a yellow-page agent and a number of service providers. The providers advertise their services by subscribing to the yellow pages. A provider that no longer wishes to be advertised can request to be unsubscribed.

The **Call-For-Proposals** pattern involves an initiator and a number of participants. The initiator issues a call for proposals for a service to all participants and then accepts proposals that offer the service for a specified cost. The initiator selects one participant to supply the service.

The **Bidding** pattern involves an initiator and a number of participants. The initiator organizes and leads the bidding process, and receives proposals. At every iteration, the initiator publishes the current bid; it can accept an offer, raise the bid, or cancel the process.

2.2. Mediation Patterns

In the **Monitor** pattern, subscribers register for receiving, from a monitor agent, notifications of changes of state in some subjects of their interest. The monitor accepts subscriptions, requests information from the subjects of interest, and alerts subscribers accordingly.

In the **Broker** pattern, the broker agent is an arbiter and intermediary that requests services from providers to satisfy the request of clients. The rest of the paper details latter the pattern to illustrate SKWYRL.

In the **Matchmaker** pattern, a matchmaker agent locates a provider for a given service requested by a client, and then lets the client interact directly with the provider, unlike brokers, who handle all interactions between clients and providers.

In the **Mediator** pattern, a mediator agent coordinates the cooperation of performer agents to satisfy the request of an initiator agent. While a matchmaker simply matches providers with clients, a mediator encapsulates interactions and maintains models of the capabilities of initiators and performers over time.

In the **Embassy** pattern, an embassy agent routes a service requested by an external agent to a local agent. If the request is granted, the external agent can submit messages to the embassy for translation in accordance with a standard ontology. Translated messages are forwarded to the requested local agent and the result of the query is passed back out through the embassy to the external agent.

The **Wrapper** pattern incorporates a legacy system into a multi-agent system. A wrapper agent interfaces system agents with the legacy system by acting as a translator. This ensures that communication protocols are respected and the legacy system remains decoupled from the rest of the agent system.

3. SKwyRL: A Social Patterns Framework

This section describes SKwyRL, a conceptual framework based on the five complementary modeling dimensions of Tropos, to investigate social patterns. Each dimension reflects a particular aspect of a MAS architecture, as follows.

- The *social dimension* identifies the relevant agents in the system and their intentional interdependencies.
- The *intentional dimension* identifies and formalizes services provided by agents to realize the intentions identified by the social dimension, independently of the plans that implement those services. This dimension answers the question: "What does each service do?"
- The *structural dimension* operationalizes the services identified by the intentional dimension in terms of agent-oriented concepts like beliefs, events, plans, and their relationships. This dimension answers the question: "How is each service operationalized?"
- The *communicational dimension* models the temporal exchange of events between agents.
- The *dynamic dimension* models the synchronization mechanisms between events and plans.

The social and the intentional dimensions are specific to MAS. The last three dimensions (structural, communicational, and dynamic) of the architecture are also relevant for traditional (non-agent) systems, but we have adapted and extended them with agent-oriented concepts.

The rest of this section details the dimensions and illustrates them with the Broker pattern.

3.1. Social Dimension

The social dimension specifies a number of agents and their intentional interdependencies using the i* model [13]. Figure 1 shows a social-dimension diagram for the Broker pattern. Agents are drawn as circles and their intentional dependencies as ovals. An agent (the *depender*) depends upon another agent (the *dependee*) for an intention to be fulfilled (the *dependum*).

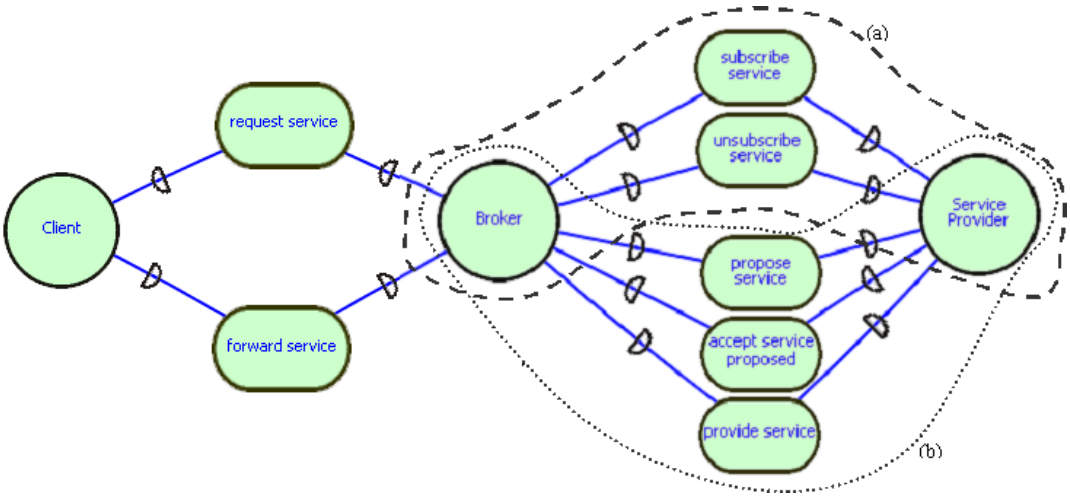

Figure 1. Social diagram for the Broker pattern

The Broker pattern can be considered as a combination of (1) a Subscription pattern (shown enclosed within dashed boundary (a)), that allows service providers to subscribe their services to the Broker agent and where the Broker agent plays the role of yellow-page agent, (2) one of the other pair patterns - Booking, Call-for-Proposals, or Bidding - whereby the Broker agent requests and receives services from service providers (in Figure 1, it is a Call-for-Proposals pattern, shown enclosed within dotted boundary (b)), and (3) interaction between broker and the client: the Broker agent depends on the client for sending a service request and the client depends on the Broker agent to forward the service.

To formalize intentional interdependencies, we use Formal Tropos [5], a first-order temporal-logic language that provides a textual notation for i* models and allows to describe dynamic constraints. A *forward service* dependency can be defined in Formal Tropos as follows.

Dependum Forward Service
Mode: Achieve
Depender: Client *cl*
Dependee: Broker *br*
Fulfillment:
($\forall$ *sr*: ServiceRequest, *st*: ServiceType)
request(*cl, br, sr*) $\wedge$ provide(*br, st*) $\wedge$ ofType(*sr, st*)
$\rightarrow \Diamond$ received(*cl, br, st*)

[Broker *br* successfully provides its service to client *cl* if all requests *sr* from *cl* to *br*, that are of a type *st* that *br* can handle, are eventually satisfied]

3.2. Intentional Dimension

While the social dimension focuses on interdependencies between agents, the intentional view aims at modeling agent rationale. It is concerned with the identification of *services* provided by agents and made available to achieve the intentions identified in the social dimension. Each service belongs to one agent. Service definitions can be formalized as intentions that describe the fulfillment condition of the service. The collection of services of an agent defines its behavior.

Table 1 lists several services of the Broker pattern with an informal definition. With the `FindBroker` service, a client finds a broker that can handle a given service request. The request is then sent to the broker through the `SendServiceRequest` service. The broker

can query its belief knowledge with the `QuerySPAvailability` service and answer the client through the `SendServiceRequestDecision` service. If the answer is negative, the client records it with its `RecordBRRefusal` service. If the answer is positive, the broker records the request (`RecordClientServiceRequest` service) and then broadcasts a call (`CallForProposals` service) to potential service providers. The client records acceptance by the broker with the `RecordBRAcceptance` service.

The Call-For-Proposals pattern could be used here, but this presentation omits it for brevity.

The broker then selects one of the service providers among those that offer the requested service. If the selected provider successfully returns the requested service, it informs the broker, that records the information and forwards it to the client (`RecordAndSendSPInformDone` service).

Service Name	Informal Definition	Agent
FindBroker	Find a broker that can provide a service	Client
SendServiceRequest	Send a service request to a broker	Client
QuerySPAvailability	Query the knowledge for information about the availability of the requested service	Broker
SendService RequestDecision	Send an answer to the client	Broker
RecordBRRefusal	Record a negative answer from a broker	Client
RecordBRAcceptance	Record a positive answer from a broker	Client
RecordClient ServiceRequest	Record a service request received from a client	Broker
CallForProposals	Send a call for proposals to service providers	Broker
RecordAndSend SPInformDone	Record a service received from a service provider	Broker

Table 1. Some services of the Broker pattern

Services can be formalized in Formal Tropos as illustrated below for the FindBroker service.

Service FindBroker (*sr*: ServiceRequest)
Mode: Achieve
Agent: Client *cl*
Fulfillment:
($\exists$ *br* : Broker, *st*: ServiceType)
provide(*br, st*) $\wedge$ ofType (*sr, st*)
$\rightarrow \Diamond$ known(*cl, br*)

[*FindBroker* is fulfilled when client *cl* has found (*known* predicate) *Broker br* that is able to perform (*provide* predicate) the service requested.]

3.3. Structural Dimension

While the intentional dimension answers the question "What does each service do?", the structural dimension answers the question "How is each service operationalized?". Services are operationalized as *plans*, that is, sequences of actions.

The knowledge that an agent has (about itself or its environment) is stored in its

beliefs. An agent can act in response to the *events* that it handles through its plans. A plan, in turn, is used by the agent to read or modify its beliefs, and send events to other agents or post events to itself.

The structural dimension is modeled using a UML style class diagram extended for MAS engineering.

The required agent concepts extending the class diagram model are defined below. The structural dimension of the Broker pattern illustrates them.

3.3.1. Structural concepts

Figure 2 depicts concepts and their relationships to build the structural dimension. Each concept defines a common template for classes of concrete MAS (for example, `Agent` in Figure 2 is a template for the agent class `Broker` of Figure 3).

A **Belief** describes a piece of the knowledge that an agent has about itself and its environment. Beliefs are represented as tuples composed of a key and value fields.

Events describe stimuli, emitted by agents or automatically generated, in response to which the agents must take action. As shown in Figure 2, the structure of an event is composed of three parts: declaration of the attributes of the event, declaration of the methods to create the event, declaration of the beliefs and the condition used for an automatic event. The third part only appears for automatic events. Events can be described along three dimensions:

- *External or internal* event: external events are sent to other agents while internal events are posted by an agent to itself. This property is captured by the *scope* attribute.
- *Normal or BDI* event: an agent has a number of alternative plans to respond to a BDI event and only one plan in response to a normal event. Whenever an event occurs, the agent initiates a plan to handle it. If the plan execution fails and if the event is a normal event, then the event is said to have failed. If the event is a BDI event, a set of plans can be selected for execution and these are attempted in turn. If all selected plans fail, the event is also said to have failed. The event type is captured by the *type* attribute.
- *Automatic or nonautomatic* event: an automatic event is automatically created when certain belief states arise. The *create when* statement specifies the logical condition which must arise for the event to be automatically created. The states of the beliefs that are defined by *use belief* are monitored to determine when to automatically create events.

A **Plan** describes a sequence of actions that an agent can take when an event occurs. As shown by Figure 2, plans are structured in three parts: the Event part, the Belief part, and the Method part. The Event part declares events that the plan `handles` (i.e., events that trigger the execution of the plan) and events that the plan produces. The latter can be either `posted` (i.e., sent by an agent only to itself) or `sent` (i.e., sent to other agents). The Belief part declares beliefs that the plan `reads` and those that it `modifies`. The Method part describes the plan itself, that is, the actions performed when the plan is executed.

The **Agent** concept defines the behavior of an agent, as composed of five parts: the

declaration of its attributes, of the events that it can post or send explicitly (i.e., without using its plans), of the plans that it uses to respond to events, of the beliefs that make up its knowledge, and of its methods.

The beliefs of an agent can be of type *private*, *agent*, or *global*. A *private* access is restricted to the agent to which the belief belongs. *Agent* access is shared with other agents of the same class, while *global* access is unrestricted.

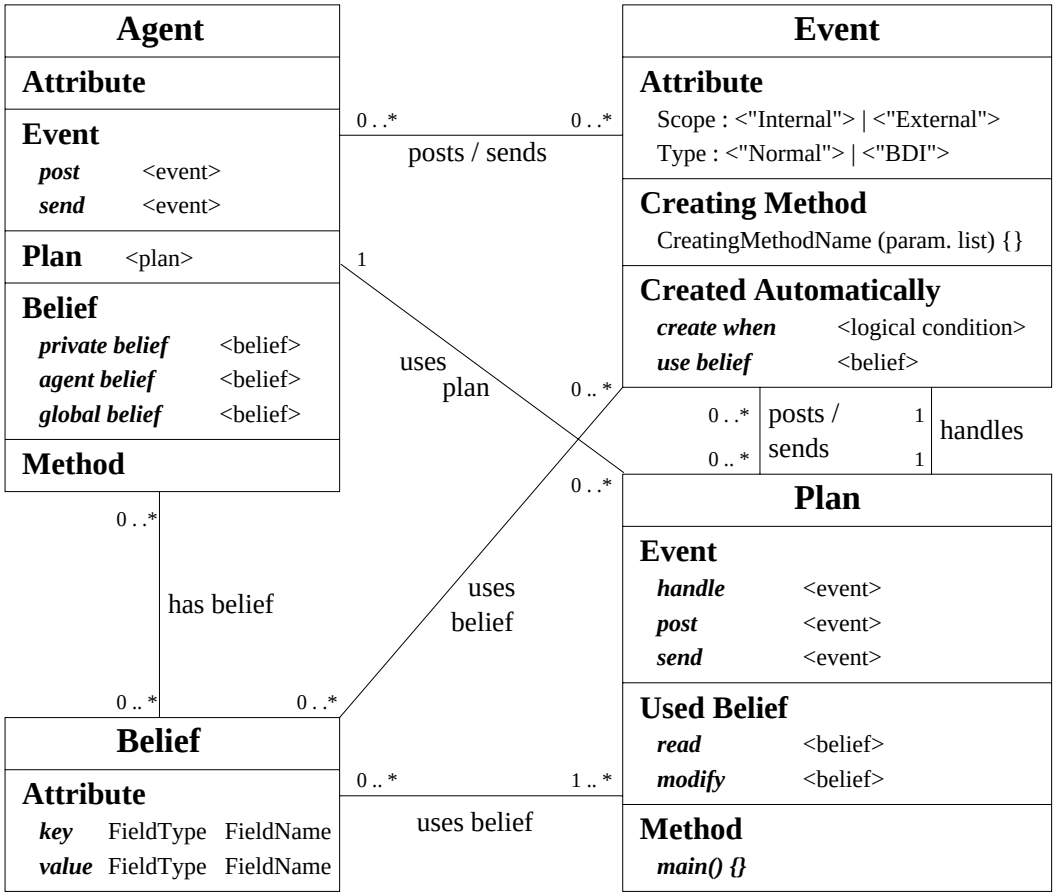

Figure 2. Structural Diagram Template

3.3.2. Structural Model for the Broker Pattern

As an example, Figure 3 depicts the Broker pattern components. For brevity, each construct described earlier is illustrated only through one component. Each component can be considered as an instantiation of the (corresponding) template in Figure 2.

`Broker` is one of the three agents composing the Broker pattern. It has plans such as `QuerySPAvailability`, `SendServiceRequestDecision`, etc. When there is no ambiguity, by convention, the plan name is the same as the as the name of the service that it operationalizes. The private belief `SPProvidedService` stores the service type that each service provider can provide. This belief is declared as private since the broker is the only agent that can manipulate it. The `ServiceType` belief stores the information about types of service provided by service providers and is declared as global since its must be known both by the service provider and the broker agent.

The constructor *method* allows to give a name to a broker agent when created. This

method may call other methods, for example `loadBR()`, to initialize agent beliefs.

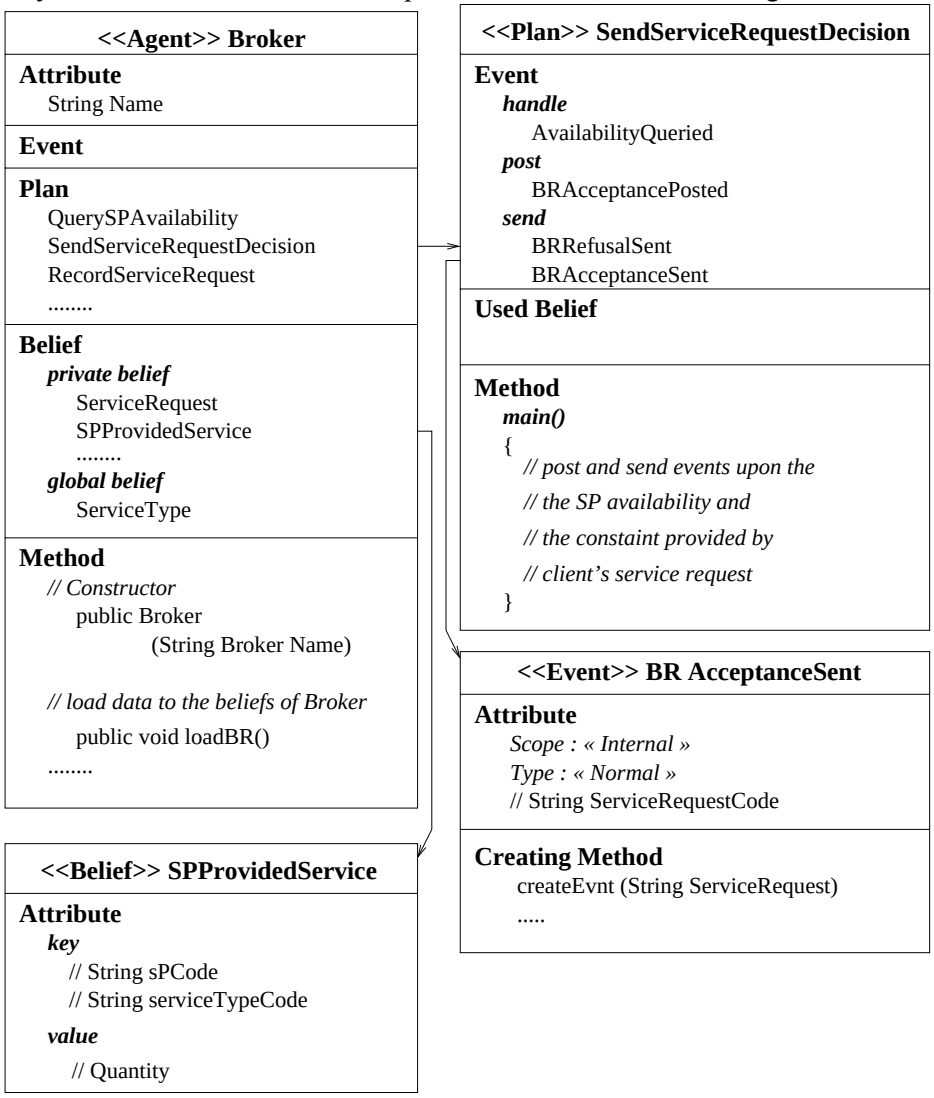

Figure 3. Structural Diagram - Some components of the Broker pattern

`SendServiceRequestDecision` is one of the plans that the broker uses to answer the client: the `BRRefusalSent` event is sent when the answer is negative, `BRAcceptanceSent` when the broker has found service provider(s) that may provide the requested service. In the latter case, the plan also posts the `BRAcceptancePosted` event to invoke the process of recording the service request and the 'call for proposals' process between the broker and services providers. The `SendServiceRequestDecision` plan is executed when the `AvailabilityQueried` event (containing the information about the availability of the service provider to realize the client's request) occurs.

`SPProvidedService` is one of the broker's beliefs used to store the services provided by the service providers. The service provider code `sPCode` and the service type code `serviceTypeCode` form the belief key. The corresponding `quantity` attribute is declared as value field.

`BRAcceptanceSent` is an event that is sent to inform the client that its request is accepted.

3.4. Communication Dimension

Agents interact with each other by exchanging events. The communicational dimension models, in a temporal manner, events exchanged in the system. We adopt the sequence diagram model proposed in AUML [2] and extend it: *agent_name/role:pattern_name* expresses the role (*role*) of the agent (*agent_name*) in the pattern; the arrows are labeled with the name of the exchanged events.

Figure 4 shows a sequence diagram for our Broker pattern. The client (`customer1`) sends a service request (`ServiceRequestSent`) containing the characteristics of the service it wishes to obtain from the broker. The broker may alternatively answer with a denial (`BRRefusalSent`) or a acceptance (`BRAcceptanceSent`).

In the case of an acceptance, the broker sends a call for proposal to the registered service providers (`CallForProposalSent`). The call for proposal (CFP) pattern is then applied to model the interaction between the broker and the service providers. The service provider either fails or achieves the requested service. The broker then informs the client about this result by sending a `InformFailureServiceRequestSent` or a `ServiceForwarded`, respectively.

The communication dimension of the subscription pattern (SB) is given at the top-right and the communication dimension of the call-for- proposals pattern (CFP) is given at the bottom-right part of Figure 4. The communication specific for the broker pattern is given in the left part of the figure.

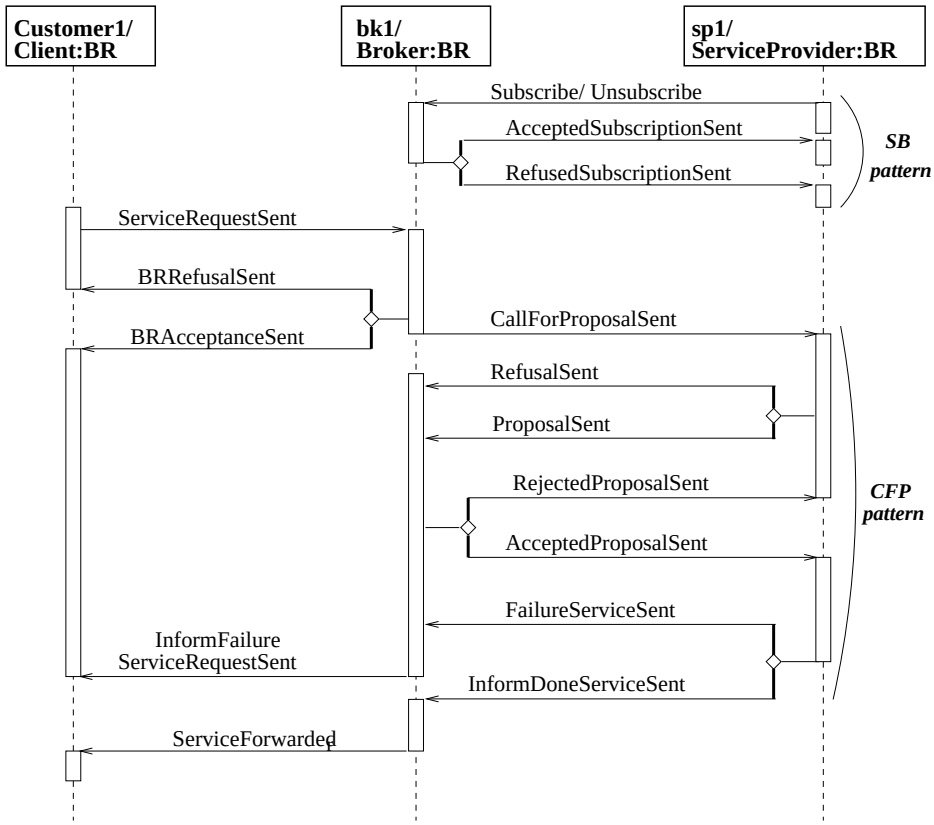

Figure 4. Communication Diagram - Broker

3.5. Dynamic Dimension

As described earlier, a plan can be invoked by an event that it handles and it can create new events. Relationships between plans and events can rapidly become complex. To cope with this problem, we propose to model the synchronization and the relationships between plans and events with activity diagrams extended for agent-oriented systems. These diagrams specify the events that are created in parallel, the conditions under which events are created, which plans handle which events, and so on.

An internal event is represented by a dashed arrow and an external event by a solid arrow. As mentioned earlier, a BDI event may be handled by alternative plans. They are enclosed in a round-corner box. Synchronization and branching are represented as usual.

We omit the dynamic dimension of the Subscription and the CFP patterns, and only present in Figure 5 the activity diagram specific to the Broker pattern. It models the flow of control from the emission of a service request sent by the client to the reception by the same client of the realized service result sent by the broker. Three swimlanes, one for each agent of the Broker pattern, compose the diagram. In this pattern, the `FindBroker` service described in Section 3.2, is either operationalized by the `FindBR` or the `FindBRWithMM` plans (the client finds a broker based on its own knowledge or via a matchmaker).

At a lower level, each plan could also be modeled by an activity diagram for further detail if necessary.

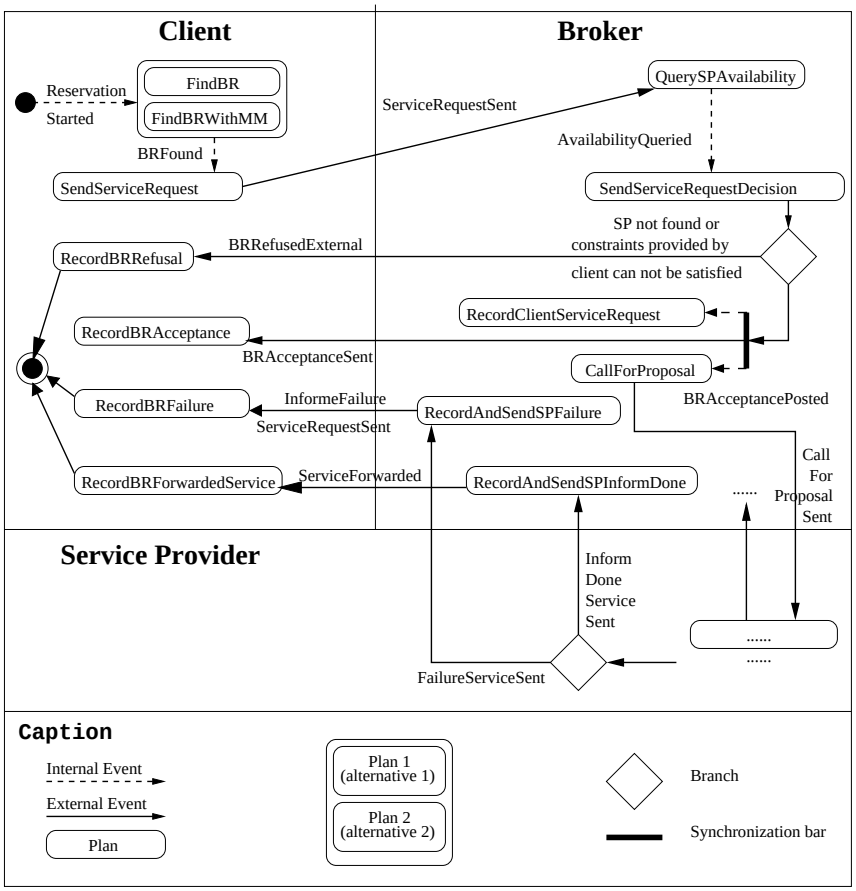

Figure 5. Dynamic Diagram - Broker

4. Code Generation

The main motivation behind design patterns is the possibility of reusing them during system detailed design and implementation. Numerous CASE tools such as Rational Rose [11] and Together [12] include code generators for object-oriented design patterns. Programmers identify and parameterize, during system detailed design, the patterns that they use in their applications. The code skeleton for the patterns is then automatically generated and programming is thus made easier.

SKWYRL proposes a code generator for the social patterns introduced in Section 2. Figure 6 shows the main window of the tool. It was developed in Java and produces code for JACK [8], an agent-oriented development environment built on top of Java. JACK extends Java with specific capabilities to implement agent behaviors. On a conceptual point of view, the relationship of JACK to Java is analogous to that between C++ and C. On a technical point of view, JACK source code is first compiled into regular Java code before being executed.

In SKWYRL's code generator, the programmer first chooses which social pattern to use, then the roles for each agent in the selected pattern (e.g. the `E_Broker` agent plays the *broker* role for the Broker pattern but can also play the *initiator* role for the CallForProposals pattern and the *yellow page* role for the Subscription pattern in the same application). The process is repeated until all relevant patterns have been identified. The code generator then produces the generic code for the patterns (.agent, .event, .plan, .bel JACK files).

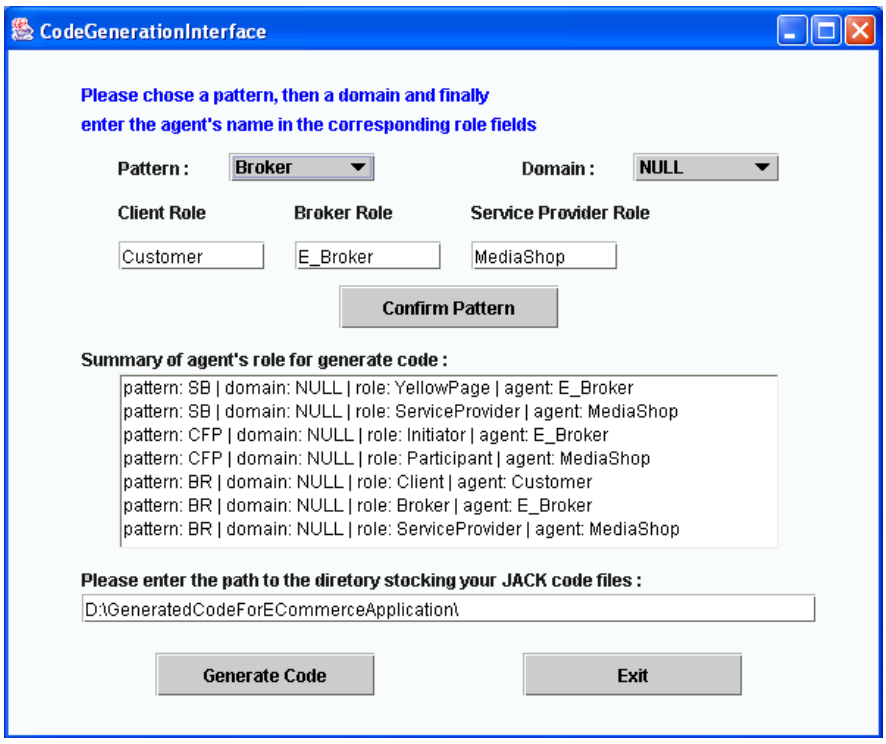

Figure 6. JACK Code Generation

The programmer has to add the particular JACK code for each generated files and implement the graphical interface if necessary.

Figure 7 shows an e-business broker developed with JACK. The code skeleton was generated with our code generator using the Broker pattern explained in the paper. The

bottom half of the figure shows the interface between the customer and the broker. The customer sends a service request to the broker asking for buying or sending DVDs. He chooses which DVDs to sell or buy, selects the corresponding DVD titles, the quantity and the deadline (the time-out before which the broker has to realizes the requested service). When receiving the customer's request, the broker interacts with the media shops to obtain the DVDs. The interactions between the broker and the media shops are shown on the bottom-right corner of this figure. The top half of the figure shows the items that are provided by each media shop. This part is hidden for the customer. However, for the ease of visualization reason of theses items, we show this interface in the figure 7.

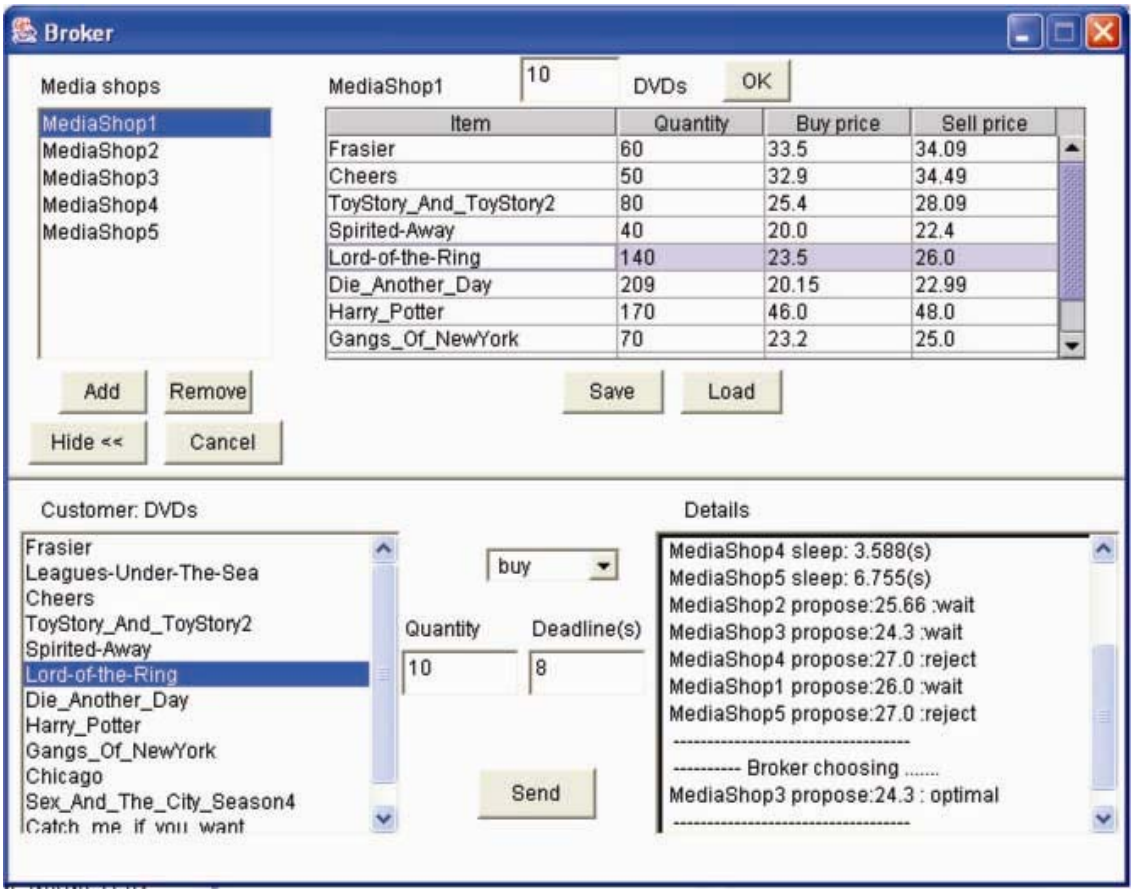

Figure 7. An E-Business Broker

5. Conclusion

Patterns ease the task of developers describing system architectures. This paper has introduced SKWYRL, a design framework, designed for the TROPOS agent methodology, to formalize the *code of ethics* for social patterns – MAS design patterns inspired by social and intentional characteristics –, answering questions like : "what can one expect from a broker, mediator, or embassy?". The framework is used to:

- define social patterns and answer the above question according to the five modeling dimensions of TROPOS: social, intentional, structural, communicational, and dynamic.

- drive the design of the details of a MAS architecture in terms of those social patterns during the detailed design phase.

The paper has overviewed some social design patterns on which we are working. The five dimensions of the framework have been illustrated through the Broker pattern.

Future research directions include the precise formalization of a catalog of social design patterns for TROPOS, including the characterization of the sense in which a particular MAS architecture is an instance of a configuration of patterns. We will also compare and contrast social patterns with classical design patterns proposed in the literature, and relate them to lower-level architectural components involving (software) components, ports, connectors, interfaces, libraries and configurations.

References

[1] Y. Aridor and D. B. Lange. "Agent Design Patterns: Elements of Agent Application Design", in *Proc. of the 2nd Int. Conf. on Autonomous Agents* (Agents'98), St Paul, Minneapolis, USA, 1998.

[2] B. Bauer, J. P. Muller and J. Odell "Agent UML: A Formalism for Specifying Multiagent Interaction". in *Proc. of the 1st Int. Workshop on Agent-Oriented Software Engineering* (AOSE'00), Limerick, Ireland, 2001.

[3] J. Castro, M. Kolp and J. Mylopoulos. "Towards Requirements-Driven Information Systems Engineering: The Tropos Project", in *Information Systems* (27), Elsevier, Amsterdam, The Netherlands, 2002.

[4] D. Deugo, F. Oppacher, J. Kuester and I. V. Otte. "Patterns as a Means for Intelligent Software Engineering", in *Proc. of the Int. Conf. on Artificial Intelligence* (IC-AI'99), Vol. II, CSRA, 1999.

[5] A. Fuxman, M. Pistore, J. Mylopoulos and P. Traverso. "Model Checking Early Requirements Specifications in Tropos", in *Proc. of the 5th IEEE Int. Symposium on Requirements Engineering* (RE'01), Toronto, Canada, 2001.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] S. Hayden, C. Carrick and Q. Yang. "Architectural Design Patterns for Multiagent Coordination", in *Proc. of the 3rd Int. Conf. on Agent Systems* (Agents'99), Seattle, USA, 1999.

[8] JACK Intelligent Agents. http://www.agent-software.com/.

[9] M. Kolp, P. Giorgini and J. Mylopoulos. "A Goal-Based Organizational Perspective on Multi-Agents Architectures", in *Proc. of the 8th Int. Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages* (ATAL'01), Seattle, USA, 2001.

[10] M. Kolp, P. Giorgini and J. Mylopoulos. "Information Systems Development through Social Structures", in *Proc. of the 14th Int. Conf. on Software Engineering and Knowledge Engineering* (SEKE'02), Ishia, Italy, 2002.

[11] Rational Rose. http://www.rational.com/rose/.

[12] Together. http://www.togethersoft.com/.

[13] E. Yu. *Modeling Strategic Relationships for Process Reengineering*, PhD thesis, University of Toronto, Department of Computer Science, Canada, 1995.