Social Patterns for Designing Multiagent Systems

T. Tung Do Manuel Kolp Alain Pirotte

Information Systems Research Unit, Catholic University of Louvain,
1, Place de Doyens, 1348 Louvain-La-Neuve, Belgium
E-mail: {do,kolp,pirotte}@isys.ucl.ac.be

Abstract

Multi-Agent Systems (MAS) architectures now appear to be more appropriate than traditional ones for building latest generation software that is typically concurrent, distributed, and dynamic. 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 to conduct the design of a MAS architecture in terms of these newly proposed idioms. We define the Booking social pattern, to illustrate the modeling dimensions of* SKWYRL. *A framework for code generation is also presented.*

1. Introduction

The characteristics and expectations in new areas of information technology, like e-business, knowledge management, peer-to-peer computing, or web services, are deeply modifying software engineering. Most of the systems designed for those areas are now concurrent and distributed. They tend to be open and dynamic in the sense that they exist in a changing organizational and operational environment where new components can be added, modified, or removed at any time.

To address those new needs, we advocate the use of multiagent system (MAS) architectures that appear to be more flexible, modular, and robust than traditional ones. A MAS could be seen as a *social organization* of autonomous software entities (agents) that can flexibly achieve agreed-upon *intentions* by interacting with one another. MAS do allow dynamic and evolving structures which can change at runtime to benefit from the capabilities of new system entities or replace obsolete ones.

An important technique that helps in the construction and documentation of architectures is the reuse of design experience and knowledge. *Design patterns* have become an attractive approach to do that (See e.g.[6]). Patterns describe a problem commonly found in software design configurations and prescribe a flexible solution for the problem, so as to ease the reuse of that solution. This solution is repeatedly applied from one design to another, producing design structures that look quite similar across different applications.

A social organization-based MAS development can help matching the system architecture with its operational environment. Taking real-world social behaviors as a metaphor together with considering the importance of design patterns for building MAS architectures, this paper focuses on *social patterns*. We define social patterns as design idioms based on social and intentional behavior for constructing MAS. By that, we mean that MAS are composed of autonomous intentional agents that socially interact and coordinate to achieve their intentions as in an organizational society.

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 the project 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 for requirements analysis. In [10] we have overviewed a social ontology for Tropos to consider software as (built of) social and intentional structures all along the development life cycle. The ontology considers organizational styles for architectural design – where the global system architecture is defined in terms of subsystems, interconnected through data, control and other dependencies – and social patterns for detailed design – where behavior of each architectural component is defined in further detail. Organizational architectural styles for Tropos have been further detailed in [9].

The present paper details the notion of social patterns. It focuses on the conceptualization of a framework called SKwyRL[1] to model these newly proposed idioms according to five complementary dimensions: social, intentional, structural, communicational, and dynamic. We propose to apply the framework to help design the detail of MAS architectures. As an illustration, the paper defines and studies a social pattern called *Booking*. 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. Section 3 proposes the SKwyRL framework, illustrates it through the Booking pattern and overviews the code generation. Finally, Section 4 summarizes the results and points to further work.

2. Social Patterns

Considerable work has been done in software engineering for defining software patterns (see e.g., [6]). Unfortunately, little emphasis has been put on social and intentional aspects. Moreover, the proposals of agent patterns that address these aspects (see e.g., [1, 4]) are not intended to be used at a design level, but rather during implementation when low-level issues like agent communication, information gathering, or connection setup are addressed.

In the following, we present patterns focusing on social and intentional aspects that are recurrent in multi-agent and cooperative systems. In particular, the structures are inspired by the federated patterns introduced in [7, 9]. We have classified them in two categories. The *Peer* patterns describe direct interactions between negotiating agents. The *Mediation* patterns feature intermediate agents that help other agents to obtain some agreement about an exchange of services.

Some of the patterns are depicted by figures that reflect their projections on a particular aspect (called modeling dimension). The meaning of these modeling dimensions is detailed in Section 3.

2.1. Peer 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 service 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 **Call-For-Proposal** 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 who performs the contracted work and informs the initiator upon completion. Figure 1 shows the social and communicational dimensions of this pattern.

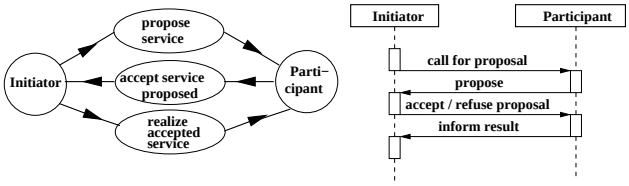

Figure 1. Call-For-Proposal Pattern

The **Bidding** pattern involves an initiator and a number of participants. The initiator organizes and leads the bidding process. He publishes the bid to the participants and receives various proposals. At every iteration, the initiator 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 a provider to satisfy the request of a consumer. Figure 2 models the social and communicational dimensions of this pattern.

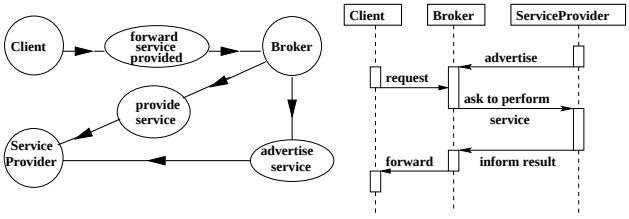

Figure 2. Broker

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

In the **Matchmaker** pattern, a matchmaker agent locates a provider for a given service requested by a consumer, and then lets the consumer interact directly with the provider, unlike brokers, who handle all interactions between consumers 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 consumers, 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 five complementary modeling dimensions, to introspect 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 the 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 the section details the dimensions. Each of them will be illustrated through the Booking pattern.

3.1. Social Dimension

The social dimension specifies a number of agents interacting with each other and their intentional interdependencies using the i* social model [13]. Agents are represented 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*).

To formalize the intentional interdependencies, we use Formal Tropos [5], a language that provides a textual notation for i* models and allows to describe dynamic constraints among the elements of the specification in a first-order temporal logic.

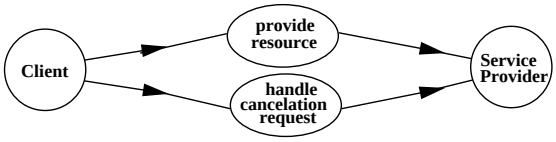

Figure 3. Social Diagram - Booking

Figure 3 shows the social-dimension diagram for the Booking pattern. The Client depends on the Service Provider to provide resources and cancel existing reservations.

A *provide resource* intentional dependency could be defined in Formal Tropos as follows.

Intention Provide Resource
Mode: Achieve
Depender: Client cl
Dependee: Service Provider
Fulfillment:

$$(\forall rr : ReservationRequest)$$
$$request(cl, rr) \rightarrow$$
$$(\exists sp : ServiceProvider, rt : ResourceType)$$
$$provide(sp, rt) \ \wedge \ ofType(rr, rt) \wedge$$
$$\Diamond \ assigned(sp, rt, rr.quantity, cl)$$

[The Client *cl* wishes that, for his reservation request *rr*, some time in the future there will exist a Service Provider *sp* able to provide *rt* the resource type of *rr* – represented by *ofType(rr, rt)* – and the number of *rt* units – represented by *rr.quantity* – will be assigned him. *ReservationRequest* and *ResourceType* are two entities that store information about the client reservation request and the type of resource that the Service Provider can provide.]

3.2. Intentional Dimension

While the social dimension focuses on interdependencies between agents, the intentional view aims to model the rationale of an agent. In other words, it is concerned with the identification of *services* possessed by agents and available to achieve the intentions identified by 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 Booking pattern with an informal definition.

Service Name	Informal Definition	Agent
FindPotentialSP	Find service providers that can provide the requested resource	Client
FindSP	Find service providers that can provide the requested resource by the client's own knowledge	Client
FindSPWithMM	Find service providers that can provide the requested resource with the help of a matchmaker	Client
SendReservation Request	Send a booking request to the potential service providers	Client
QueryResource Availability	Query the database for information about the availability of the requested resource	Service Provider
SendReservation Decision	Send answer to client	Service Provider
RecordSPRefusal	Record a negative answer from the service provider	Client
RecordWLProposal	Record a proposal for a waiting list option	Client
RecordOffer	Record an offer for a resource	Client
RecordSPRefusal	Record a negative answer	Service Provider

Table 1. Some Services of the Booking Pattern

The `FindPotentialSP` service allows a client to find service providers that can provide a requested resource. This service is fulfilled either by the `FindSP` or the `FindSPWithMM` services (the client finds potential service providers based on its own knowledge or via a matchmaker). The request is then sent to the potential service providers through the `SendReservationRequest` service. When receiving such a request, a service provider queries its database using the `QueryResourceAvailability` service and then answers the client through the `SendReservationDecision` service. Three alternative answers are possible: (1) the resource cannot be supplied; (2) the resource can be supplied but not for the moment: a waiting list option is proposed to the client; (3) the resource can be provided and an offer is made to the client. The client processes these answers with services `RecordSPRefusal`, `RecordWLProposal`, and `RecordOffer`, respectively. The service provider also records its negative answers, for later reminiscence, through its `RecordSPRefusal` service.

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

Service FindPotentialSP
Type : Intention
Mode: Achieve
Agent: Client cl
Fulfillment:
$(\forall rr : ReservationRequest)$
$request(cl, rr) \rightarrow$
$(\exists sp : ServiceProvider, rt : ResourceType)$
$(provide(sp, rt) \wedge ofType(rr, rt) \wedge$
$\Diamond known(cl, sp)$
[*FindPotentialSP* is fulfilled when the client has found (*known* predicate) some *ServiceProvider*(s) that is(are) able to perform (*provide* predicate) the reservation requested.]

3.3. Structural Dimension

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

As already said, a MAS is a social organization of intentional autonomous software entities called *agents*. The knowledge an agent has (about itself or about the environment to which it belongs) is stored in its *beliefs*. An agent can act in respond to the *events* it handles through its plans. A plan in turn, is used by the agent to read or modify its beliefs and send (or post) events to other agents (or 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 Booking pattern illustrates them.

3.3.1 Structural concepts

Figure 4 depicts the concepts and their relationships, needed to build the structural dimension. Each concept defines a common template for classes of concrete MAS (for example, `Agent` in Figure 4 is a template for agent class `ServiceProvider` of Figure 5).

A **Belief** describes the knowledge that an agent has about itself and its environment. A belief is a tuple 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 by Figure 4, 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 / Internal* event: event that the agent sends to other agents / event that the agent posts to itself. This property is captured by the *scope* attribute.
- *Normal / BDI* event: An agent has some alternative plans in response 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, in order to try to achieve successful plan execution. If all the plans in the set of selected plans have failed, the event is also said to have failed. The event type is captured by the *type* attribute.
- *Automatic / Not Automatic* 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* is monitored to determine when to automatically create an event.

A **Plan** describes a sequence of actions that an agent can take when an event occurs. As shown by Figure 4, 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, including the type of events it posts and sends, the plans it uses to respond to the events it handles, and the beliefs it has as its knowledge.

The agent structure is composed of five parts: declaration of its attribute, declaration of events it posts (or sends) explicitly (i.e., without using its plans), declaration of its plans, declaration of its beliefs and declaration of its methods.

The beliefs of an agent can be of type *private*, *agent*, or *global* depending on how the agent can access beliefs

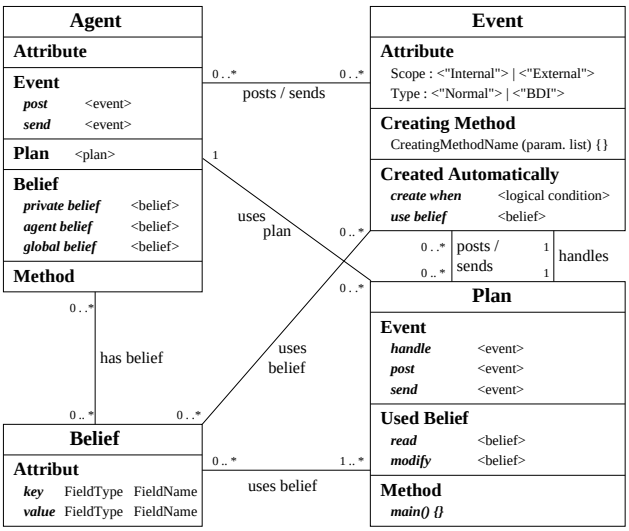

Figure 4. Structural Diagram Template

data. A *private* access means that the agent can read and modify independent of all other agents, even those of the same agent class; an *agent* access means that the agent has shared read-only access to the belief, but only with other agents of the same class; a *global* access means that the agent has shared read-only access to the belief with all other agents.

3.3.2 Booking Pattern Structural Model

As an example, Figure 5 depicts the Booking pattern components. Due to lack of space, each construct described earlier is illustrated only through one Booking pattern component. Each of these components can be considered an instantiation of the (corresponding) template defined in the previous section.

`ServiceProvider` is one of the two agents composing the Booking pattern. It uses plans such as `QueryResourceAvailability`, `SendReservationDecision`, etc. The plan name is also the name of the service that it operationalizes.

The global belief `ResourceType` is used to store the resource type and its descriptions (e.g., for an air ticket booking system, the resource type could be `economic / first class / business class`; for a hotel room booking system the resource type could be `single room / double room / ...`). This belief is declared as global since it will be used by both the client agent and the service provider agent. The other beliefs are declared as private since the service provider is the only agent that can manipulate them.

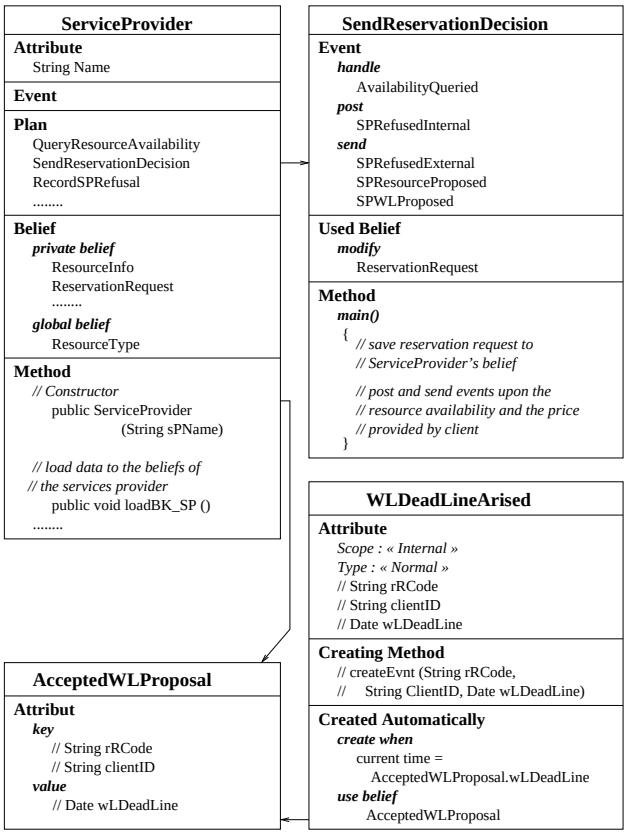

Figure 5. Structural Diagram - Booking

The constructor *method* allows to give a name to a service provider agent when created. This method may call other methods, for example `loadBK_SP()`, to initiate agent beliefs data.

`SendReservationDecision` is one of the Booking pattern plans the service provider uses to answer the client: the `SPRefusedExternal` event is sent when the answer is negative, `SPResourceProposal` when there is some resource available for the client, or `SPWLProposal` (Service Provider Waiting List Proposal) when the service provider is able to provide the type of resource asked by the client but not at the moment of the request since all these kind of resources are reserved. This plan is executed when the `AvailabilityQueried` event (containing the information about the availability of the resource type required by the client) occurs.

This plan also modifies `ReservationRequest`, the service provider's belief storing the client's reservation request before the service provider sends (or posts) his answer.

`AcceptedWLProposal` is one of the service provider's beliefs used to store the client's accepted waiting list proposal. The reservation request code `rRCode` and the `clientID` form the belief key. The `reservationInfoCode` attribute that contains the correspondent code of the resource type requested by the client, and the `wLDeadLine` that contains the time-out before which the service provider must send a *resource not available* message to the client if no resource is proposed, are declared as value fields.

`WLDeadlineArised` is an event that is posted automatically whenever the time-out `wLDeadLine` (of the `AcceptedWLProposal` belief) is reached. It will then invoke a plan to inform the client that the resource is not available.

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 (*pattern_name*) of the agent (*agent_name*) in the pattern; the arrows are labeled with the name of the exchanged events.

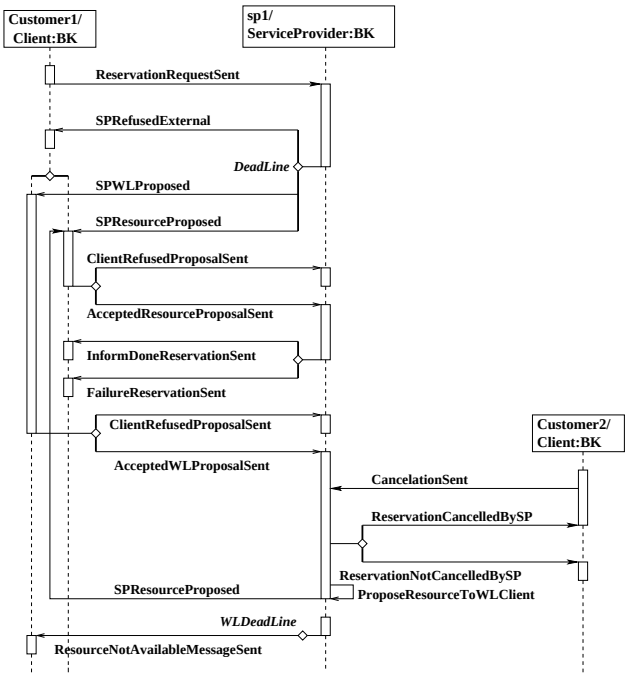

Figure 6. Communication Diagram - Booking

Figure 6 shows the sequence diagram for our Booking pattern. The client (`customer1`) sends a reservation request (`ReservationRequestSent`) containing the characteristics (place, room, etc.) of the resource it wishes to obtain from service providers. The service provider may alternatively answer with a denial (`SPRefusedExternal`), a waiting list proposal (`SPWLProposed`) or an approval, i.e, a resource proposal when there exists such a resource that satisfies the characteristics the client sent.

In the case of a waiting list proposal (`SPWLProposed`), when the client accepts it (`AcceptedWLProposalSent`), it sends a waiting list time-out (`wLDeadLine`) to the service provider. Before reaching the time-out, the service provider must send a refusal to the client, in the case it does not find an available slot in the waiting list (`ResourceNotAvailableMessageSent`), or propose a resource to the client. In the later case, the interaction continues as in the case that the resource proposal is sent to the client.

A resource that is not available becomes available when some client (`customer2` in Figure 6) cancels its reservation.

3.5. Dynamic Dimension

As described earlier, a plan can be invoked by an event that it handles and 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 an event is created, which plan handles which event, 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. A plan is represented by a lozenge shape. Synchronization and branching are represented as usual.

Four activity diagrams actually model the dynamic dimension of the Booking pattern; due to lack of space, we only present one, depicted by Figure 7. It models the flow of control from the emission of a reservation request to the reception by the client of the answer from the service provider (refusal, resource proposal, or waiting list proposal). Two swimlanes, one for each agent of the Booking pattern, compose the diagram.

`MaxPrice` stores the maximum value that the client can afford to obtain a resource unit; `quantity` stores the number of resource units the client wishes to book; `reservedQuantity` and `maxQuantity` respectively store the actual quantity of resource units that are reserved and the maximum number of resource units that the service provider can provide.

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

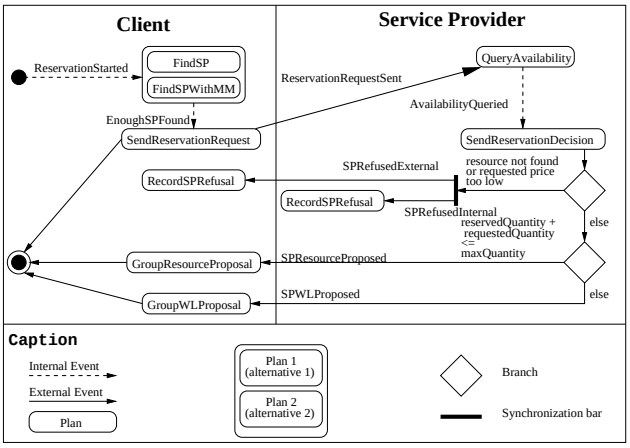

Figure 7. Dynamic Diagram - Booking

3.6. 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] have then included 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. The work of programmers is made easier, as they only have to add the application code too specific to be generated.

SKWYRL proposes a code generator for the social patterns introduced in Section 2. Figure 8 shows the main window of the tool. It is developed with 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.

With the code generator of SKWYRL, the programmer first chooses a social pattern to use and selects an application domain on which the pattern will be applied. For instance, for the Booking pattern, common domains such as `AirlineTicket` and `HotelRoom` have been already

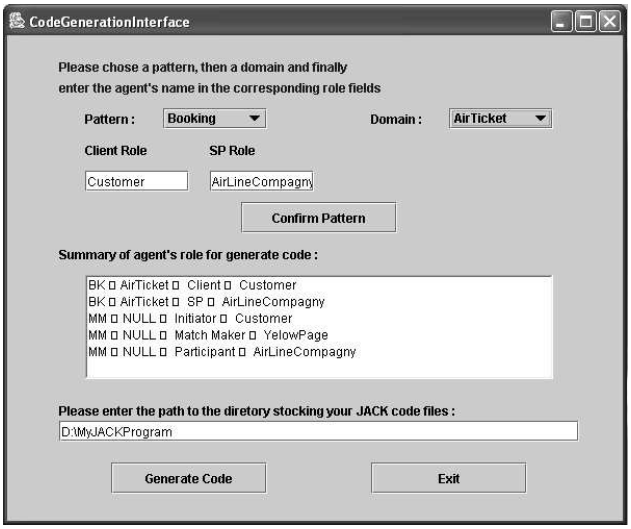

Figure 8. JACK Code Generation

pre-configured. The programmer has then to choose the role for each agent in the pattern (e.g. the `Traveler` agent will play the role of *client* in the `AirlineTicket` domain of the Booking pattern and he can also play the role of *initiator* of the Match Maker pattern in the same application).

As an experiment, on the 75-80 files (including the interfaces files) required to run a Booking program in JACK, SKWYRL's code generator produces the generic code for 70-72 of them (the .agent, .event, .plan, .bel JACK files).

4. Conclusion

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

- define social patterns and answer the above question according to five modeling dimensions: social, intentional, structural, communicational and dynamic.
- drive the design of the details of a MAS architecture in terms of these social patterns.

The paper has overviewed some social design patterns on which we are working with SKWYRL. The five dimensions of the framework are illustrated through the definition of a social pattern that we called *Booking*.

Future research directions include the complete and precise formalization, through SKWYRL, of a catalogue of social design patterns, 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. of 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, J. 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://http://www.rational.com/rose/.

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

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