An Expressway from Agent-Oriented Models to Prototypes

Kuldar Taveter and Leon Sterling

Department of Computer Science and Software Engineering
the University of Melbourne
Vic 3010, Australia
{kuldar,leon}@csse.unimelb.edu.au
http://www.csse.unimelb.edu.au

Abstract. Agent-oriented software engineering can be viewed as applying software engineering principles to agent-oriented development or applying agent-oriented principles to software engineering. In this paper, we are more concerned with the second view. We describe how prototype systems can be efficiently created from agent-oriented domain and design models. We propose a conceptual space that accommodates model transformations described by the Model-Driven Architecture. We explain agent-oriented domain models and platform-independent design models and show how the first can be mapped to the latter. We demonstrate how design models can be turned into the implementation of an agent-based prototype on a specific platform. The approach has potential for accelerating the process of rapid prototyping.

1 Introduction

Agent-oriented software engineering can be viewed as applying software engineering techniques and principles to the development of agent-oriented systems, but also as applying agent-oriented principles to developing software. In the latter spirit, we believe that agent-oriented modelling techniques are not just useful for designing systems consisting of software agents, i.e. multi-agent systems. Agent-oriented modelling can, and should, be more generally utilized for designing distributed open socio-technical systems. It can accommodate Web services and component-based systems. What makes agent-oriented modelling suitable is distinguishing between active entities — agents — and passive ones — objects.

Model-Driven Architecture (MDA) [1] by Object Management Group (OMG) is an approach to using models in software development that separates the domain model of a socio-technical system from its design and implementation models. The MDA proposes three types of models: Computation-Independent Models (CIM), Platform-Independent Models (PIM), and Platform Specific Models (PSM). In MDA, a *platform* denotes a set of subsystems and technologies that provide a coherent set of functionalities through interfaces and specified

M. Luck and L. Padgham (Eds.): AOSE 2007, LNCS 4951, pp. 147–163, 2008.

usage patterns. Some examples of platforms are CORBA, Java 2 Enterprise Edition, Microsoft.NET and JADE.

In addition to defining model types at different abstraction layers, the MDA also introduces the term "Model transformation" which is the process of converting one model to another model of the same system. It defines mapping between models as a "specification of a mechanism for transforming the elements of a model conforming to a particular metamodel into elements of another model that conforms to another (possibly the same) metamodel" [1]. To that end, different techniques like model marking as described by MDA, and using templates and mapping languages have been proposed. The MDA focuses on transformation between PIM and PSM, because executable PSM models can be easily generated from PIM models. This is not the case for mapping from CIM to PIM, which are conceptually more separated. To support mapping from CIM to PIM, we propose an appropriate set of CIM and PIM concepts that can be mapped from one another.

As represented in Figure 1, the modelling abstractions we advocate in CIM include *goals* and *roles*, which appear in most agent-oriented methodologies with a similar — though often not identical — meaning. In addition, *social policies* are constraints on interaction and behaviour of agents playing the roles. *Domain entities* define the basic concepts of the problem domain at hand.

For PIM, we have chosen *activities* that are triggered by *rules* as key notions. Both activities and rules are rooted in activity theory [16]. We prefer them to capabilities and plans because activities and rules represent more naturally the nature of activities by human and man-made agents and are free from the bias towards any specific agent architecture like BDI [6]. According to Figure 1, goals and roles can be mapped to *activity types* and *agent types*, respectively. *Social policies* can be mapped to *rules* and *domain entities* to *knowledge* items. Activity types, in turn, consist of *action types*.

The mappings explained do not imply the losing of knowledge of higher abstraction levels at lower abstraction levels. For example, the knowledge of roles can still be retained and utilized at the PIM level and goals after they have been assigned to activities for achieving them can still be explicitly represented in PIM.

The platform-independent notions *action types, rules*, and *agent types*, along with *perception types* and *knowledge items* can be mapped into the corresponding *concrete action types, behavioural construct types*, and *concrete agent types* as well as *event types* and *concrete object types* of some specific platform like JADE [13].

The mappings outlined in Figure 1 can be used for rapid obtaining of prototypes. In some cases, also final implementations can be obtained, but usually design decisions are restricted by commercially available and *preferred* technology.

In addition to the horizontal dimension of modelling, which is represented by Figure 1, there is also a vertical dimension. In [9], the first author has performed a thorough study of various software engineering methodologies and modelling approaches and has concluded that agent-oriented models should address a problem domain from six perspectives: informational, organisational, interactional,

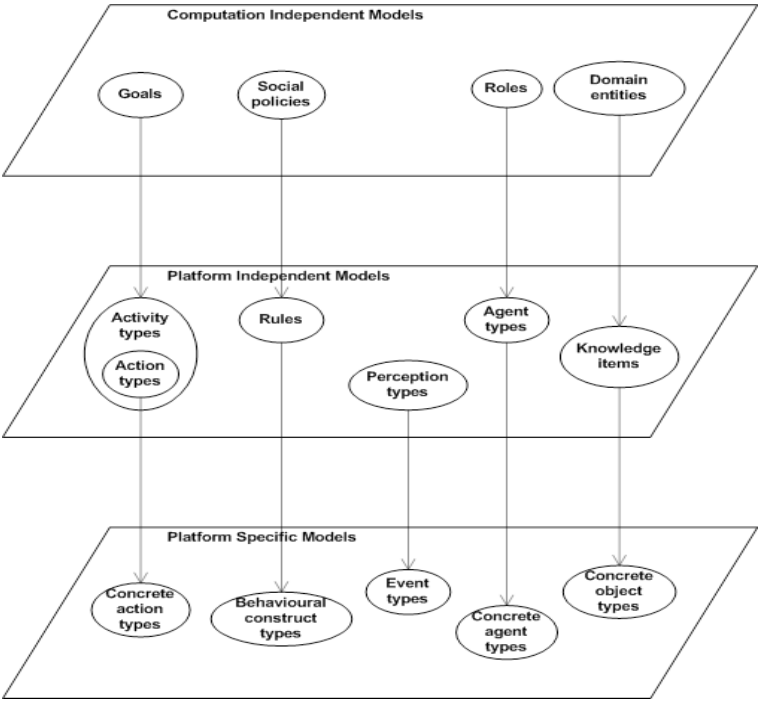

Fig. 1. The Conceptual Space of transformations between different layers of MDA

Table 1. The Viewpoint Modelling Framework

Viewpoint models	***Viewpoint aspect***		
Abstraction layer	*Organisation/ Interaction*	*Information*	*Motivation/ Behaviour*
Computation independent domain analysis (CIM)	Role Models (ROADMAP)	Domain Model (ROADMAP)	Goal Models (ROADMAP)
Platform independent computational design (PIM)	Interaction Models (RAP/AOR)	Information Model (RAP/AOR)	Behaviour Models (RAP/AOR)
Platform specific design and implementation (PSM)	Class and Sequence Diagrams (UML)	Class Diagrams (UML)	Class and Sequence Diagrams (UML)

functional, motivational, and behavioural. In [11], we have identified informational, interactional, and behavioural perspectives as the most crucial ones for agent-oriented design. On the other hand, it can be concluded from [3], [4], and [17] that organisational, informational, and motivational perspectives are the most relevant ones for agent-oriented domain analysis. In Table 1, we have accordingly grouped the perspectives explained above as three viewpoint aspects. This table can be populated in many ways. For example, at the CIM level, motivation models are featured in MaSE [18] as Goal Hierarchy Diagrams, domain models have been proposed as Environment Models in GAIA [19], and organisation models appear as Organisation Diagrams in MESSAGE [20]. Similarly, at the PIM level, behaviour models are represented as Multi-Agent Behaviour Descriptions in PASSI [21], information models appear in MAS-CommonKADS [22] as Expertise Models, and interaction models are featured in Prometheus [14] as Interaction Diagrams and Interaction Protocols.

The structure of Table 1 is thus not associated with any specific software engineering methodology but provides a universal framework for classifying the kinds of models appearing in various methodologies and approaches. However, we have populated Table 1 in a specific way to cater for the needs of rapid prototyping addressed by this article. In other words, we have selected the types of models appearing in Table 1 because it has been shown earlier [23] that this combination of models facilitates rapid prototyping. The model types chosen by us originate in the ROADMAP [3,4] and RAP/AOR [11] methodologies and in the Unified Modelling Language (UML) [12]. Please note that UML models as such are not platform-specific but can be used for modelling platform-specific issues.

In the next section we present types of models at the three abstraction layers — computation independent modelling, platform independent computational design, and platform specific design and implementation — by using an example of creating a system for ordering take-away food, which has been borrowed from [2].

2 Computation Independent Modelling

According to MDA [1], the models created at the computation independent modelling stage should be capable of bridging the gap between experts about the domain and its requirements on one hand, and experts about the design and construction of the socio-technical system on the other. The models should address *motivation* for the system to be designed, *organisation* of the system, and the *environment* in which the system is to be situated. Our experience with industry reported in [23,27], as well as with students in our graduate Agents class at the University of Melbourne, has proven that motivation for the system can be effectively communicated by Goal Models, organisation of the system — by Role Models — and the environment — by Domain Models.

Our goal and role models have been described in [3] and [4], and we review here for completeness. The Goal Model provides a high-level overview of system requirements. Its main objective is to enable both domain experts and developers

to pinpoint the *goals* of the system and the *roles* the system needs to fulfil in order to meet those goals. Design and implementation details are not described at all, as they are not addressed during requirements analysis. The Goal Model contains three components: goals, quality goals, and roles. A *goal* represents a functional requirement of the system. A *quality goal*, as its name implies, represents a non-functional or quality requirement of the system. A *role* is some capacity or position that the system requires in order to achieve its goals. As Figure 2 shows, goals and quality goals can be decomposed into smaller related sub-goals and sub-quality goals, allowing hierarchical structure between a goal and its sub-goals. The resulting hierarchy is by no means an "is-a" or generalisation relationship as is common in object-oriented methodologies. Rather, the hierarchical structure is just to show that the sub-component is an *aspect* of the top-level component.

Figure 2 represents the Goal Model of a socio-technical system to be designed for ordering take-away food. In the diagram, the root goal is to 'provide meal'. This goal is associated with the roles Customer, Ordering Centre, and Restaurant. The role Customer represents the stakeholders whose needs the socio-technical system is to satisfy. The system itself consists of actors playing the roles Ordering Centre and Restaurant. The goal to 'provide meal' can be decomposed into the following four sub-goals: to 'take order', 'provide waiting estimate', 'confirm order', and 'deliver meal'. The goal to 'provide meal' is characterized by the quality goal 'customer happy'. There are also the quality goals 'fast reply' and 'fast delivery' pertaining to the sub-goals to 'provide waiting estimate', 'confirm order', and 'deliver meal'. Quality goals represent social policies, which can be anything from access rights, to social norms, to obligations [17]. Please note that the order in which the sub-goals are presented in Figure 2 does not per se imply any chronological order in which they are to be achieved.

The Role Model describes the properties of a role. The Role Model consists of the role name, textual description, and the specifications of its responsibilities

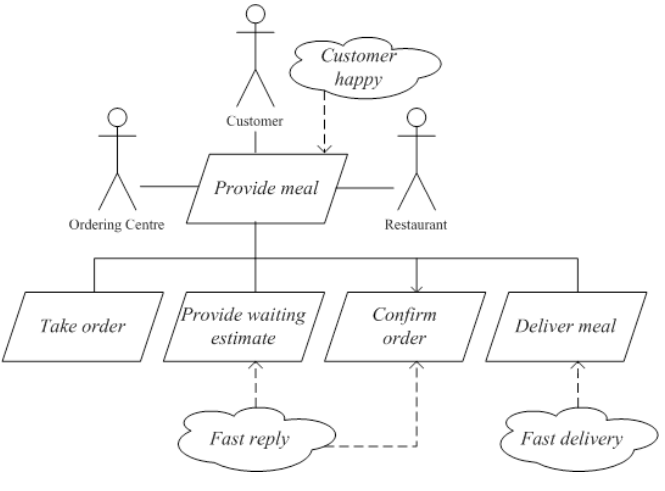

Fig. 2. The Goal Model for the take-away food ordering system

and constraints. Clearly, this is analogous to the delegation of work through the creation of positions in a human organisation. Every employee in the organisation holds a particular position in order to realise business functions. Different positions entail different degrees of autonomy, decision-making, and responsibilities. Taking this analogy, the Role Model is the "position description" for a particular role. Table 2 shows the Role Model created for the role Restaurant shown in the Goal Model in Figure 2.

Table 2. The Role Model for the Restaurant

Role Name	Restaurant
Description	Provides the time estimate for delivery and delivers the meal
Responsibilities	Receive the order Estimate the time required for cooking Inform the ordering centre about the time required Accept the confirmation by the ordering centre Deliver the meal to the customer
Constraints	The deliverer must use an electronic signature device to register the delivery

The Domain Model represents agents' knowledge about their physical and conceptual environments. It can be viewed as an *ontology* providing a common framework of knowledge for agents playing the roles of the problem domain. For example, a take-away food ordering system requires the domain entities Cook, Dish, and Order. The first describes the kinds of agents in the system's physical environment, the second — a particular kind of food and the third — a particular order. The Domain Model can be initially expressed as a list of domain entities showing for each of them with which role(s) it is associated. For example, the domain entities Dish and Order are associated with all three roles — Customer, Ordering Centre, and Restaurant — while the domain entity Cook is associated with just the role Restaurant. Relationships between domain entities, such as generalisation and aggregation, can be represented by using a UML-like notation.

3 Platform Independent Design

According to MDA [1], platform independent modelling focuses on the operation of a system while hiding the details necessary for a particular platform. The resulting models are suitable for use with a number of different platforms of a similar type. The models should address *interactions* between agents of the system to be designed, *information* that those agents require for operating, and *behaviours* of the agents.

Since our models can be used for designing Web services as well as agent-based systems, we are interested in goal-oriented rather than goal-governed agents [5]. *Goal-governed agents* refer to the strong notion of agency, that is, they are agents with some forms of cognitive capabilities, making possible explicit representation

of their goals that drive the selection of agent actions. An example class of goal-governed agents are BDI-agents [6]. *Goal-oriented agents* refer to the weak notion of agency, that is, they are agents whose behaviour is directly designed and programmed to achieve some goal, which may not be explicitly represented. Goal-oriented agents generalize over a wide range of software components rather than just over software agents. An example goal-oriented agent architecture is AGENT-0 by Yoav Shoham [7]. Agents of both kinds can be derived from the Goal Models, Role Models, and Domain Models.

We view goal-oriented agents as being engaged in various *activities*. Based on activity theory [16], we consider activities as fundamental units of human and man-made agent behaviour. Activity is started by a *rule* when the activity's triggering conditions are true. Activity is triggered by some event perceived by an agent and/or by some value associated with an object in the agent's knowledge base.

We have chosen as the goal-oriented agent architecture of PIM *Knowledge-Perception-Memory-Commitment* (KPMC) agents, proposed in [8] and extended by [9]. KPMC-agents can be graphically modelled by using diagrams included by the Radical Agent-Oriented Process / Agent-Object-Relationship (RAP/AOR) methodology of software engineering and rapid prototyping, which was introduced in [11]. Before introducing PIM models of the case study of ordering take-away food, we briefly explain the notation that will be used.

An *external* (that is, modelled from the perspective of an external observer) *Agent-Object-Relationship (AOR) diagram* specified by Figure 3 enables the representation in a single diagram of the types of human and man-made (for example, software) agents of a socio-technical system, together with their beliefs about instances of "private" and external ("shared" with other agents) object types. There may be attributes and/or predicates defined for an object type and relationships (associations) among agent and/or object types. A predicate, which is visualized as depicted in Figure 3, may take parameters.

Figure 3 reflects that our graphical notation distinguishes between an *action event* (an event perceived by one agent that is created through the action of another agent, such as a physical reception/delivery of a meal) *type* and a *non-action event type* (for example, types of temporal events or events created by natural forces). We further distinguish between a *communicative* action event (or *message*) type and a *non-communicative* (physical) action event type like providing the customer with a meal.

The first thing to be done at the design stage is mapping the abstract constructs from the analysis stage — *roles* — to concrete constructs — *agent types*. Each agent type may be assigned one or more roles and the other way round. In our simple example, assigning the roles to agent types is straightforward. All three roles — Customer, Centre, and Restaurant — are mapped to the respective man-made agent types `CustomerAgent`, `CentreAgent`, and `RestaurantAgent`. There may be several instances of `CustomerAgent` and `RestaurantAgent`, and there is exactly one `CentreAgent`.

In [11], three complementary modelling perspectives are identified for agent-oriented design. The resulting models can be represented as just one diagram of the kind shown in Fig. 3. We will now treat platform independent design from each of the three perspectives — *interaction design, information design*, and *behaviour design*. As stated above, interaction design models capture interactions between the agents of the system, information design models represent information that those agents require for operating, and behaviour design models specify behaviours of the agents.

In our view, the mapping between CIM and PIM cannot be fully formalized because of the intangible nature of CIM models. What is important is that the mapping is traceable in the sense that it can be seen how CIM modelling constructs relate to the PIM models. The mapping should be supported by tools no matter what degree of automation can be achieved. In the next three sections, we also explain the rationale of deriving a design model of each kind.

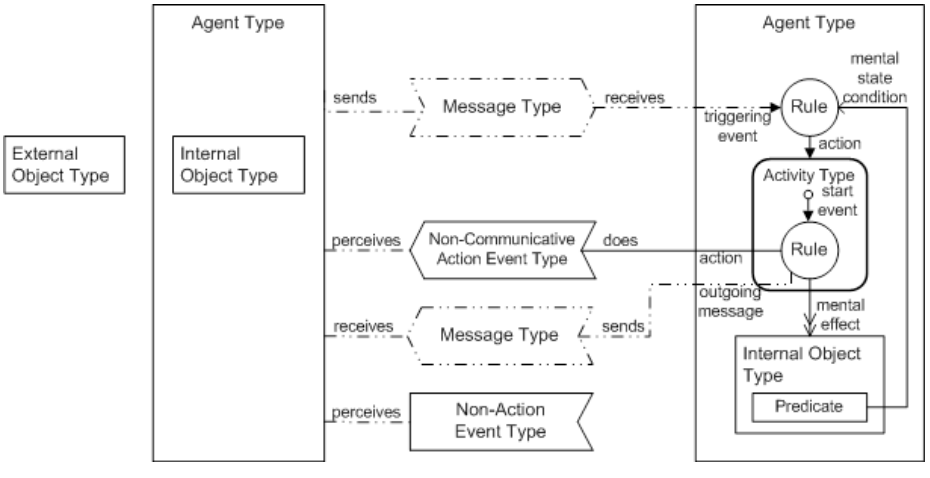

Fig. 3. The belief structure and behaviour modelling elements of external AOR diagrams

3.1 Interaction Design

After determining agent types, we can capture interactions between agents of those types with the Interaction Model represented as an interaction-frame diagram. Interactions can be derived from responsibilities included by Role Models. The interaction frame diagram depicted in Figure 4 consists of two *interaction frames* that have been derived from the Role Model shown in Table 2: one between the agents of a customer and the ordering centre, and the other one between the agents of the ordering centre and a restaurant. Messages in interaction frames have four modalities: "request", "inform", "confirm", and "reject". With a message of the "request" modality, an agent requests another agent to perform a certain action, which can be a communicative action — sending a message — or

a physical action. A message of the "inform" modality serves to inform another agent on something. The last two modalities explain themselves. Messages of different modalities can be combined. For example, with a message of the type `request inform time-estimate(Dish(?DishName))`, an agent requests another agent to inform it about the expected time required to prepare and deliver the meal described by a serialized object of the type `Dish`. An argument preceded by a question mark appearing in message content, such as `?DishName`, denotes a string. The interaction represented at the bottom of Figure 4 models a physical action of the type `provideDish(Order(?OrderID))` that occurs between agents of the types `RestaurantAgent` and `CustomerAgent`. This action is naturally only registered rather than performed by the corresponding software agents. This can be accomplished by an electronic device incorporating both an actuator and a sensor where the action is pushing a button by the deliverer and the event is signing by the customer.

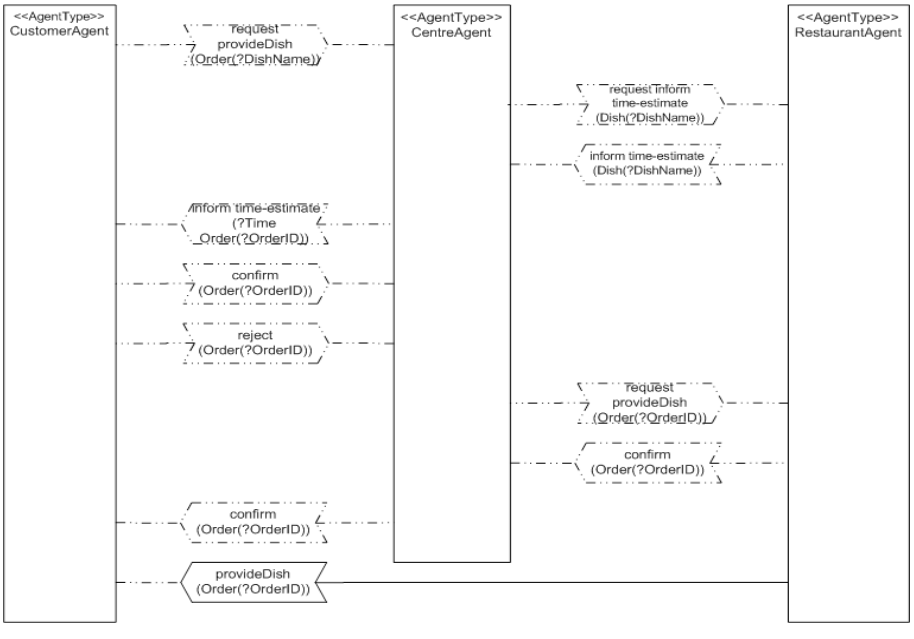

Fig. 4. The Interaction Model for the take-away food ordering system

3.2 Information Design

In information modelling, we further extend and formalize the *ontology* providing a common framework of knowledge for the agents of the problem domain. Recall that the initial version of this ontology — the Domain Model — was created at the stage of domain analysis. Each agent can see only a part of the ontology; that is, each agent views the ontology from a specific perspective. We represent the resulting Information Model as the *AOR agent diagram* shown in Figure 5.

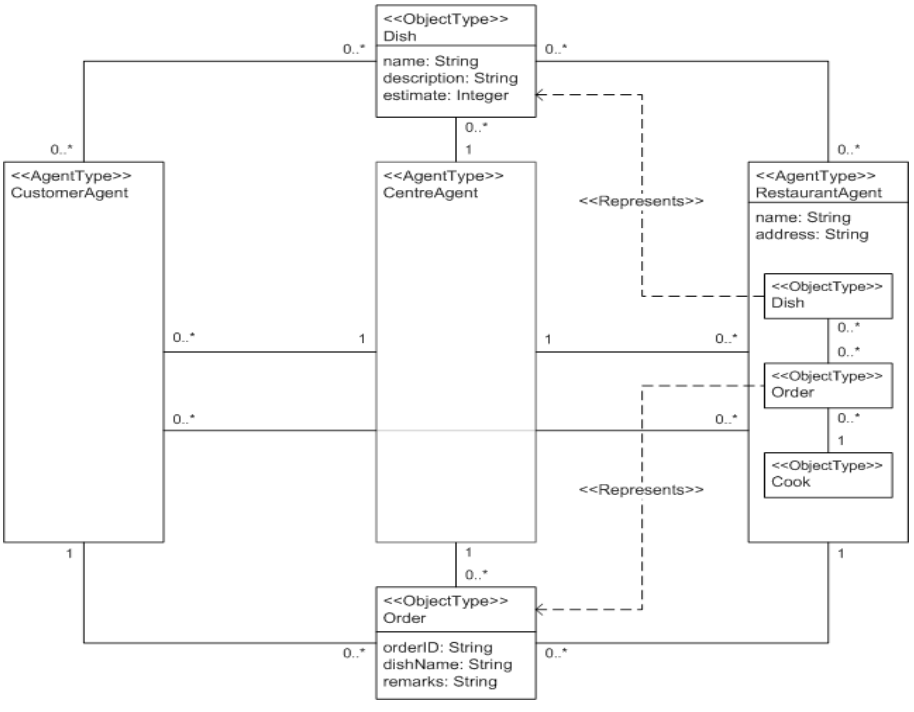

Fig. 5. The Information Model for the take-away food ordering system

In the figure, an agent of the type `CustomerAgent`, representing a customer, has knowledge about one agent of the type `CentreAgent`, which represents the ordering centre, and about several agents of the type `RestaurantAgent` representing restaurants. The `CentreAgent`, in turn, is aware of agents of both other types. Each restaurant agent is aware of the `CentreAgent` and of agents of its customers served by the restaurant.

Additionally, the Information Model depicted in Figure 5 represents that agents of all three types may have a shared knowledge about one or more instances of the object types `Dish` and `Order`. The model also shows that a restaurant agent has private knowledge about inter-related instances of the object types `Dish` and `Order`. Atomic information elements are described as *attributes* rather than objects. As is reflected by Figure 5, an agent of the type `RestaurantAgent` has the attributes `name` and `address` that characterize the restaurant represented by it. Objects of the types `Dish` and `Order` are also described by their respective attributes.

3.3 Behaviour Design

Under behaviour design, goals of CIM are mapped to activity types of PIM. An activity of a given type accomplishes a goal from the Goal Model. For example,

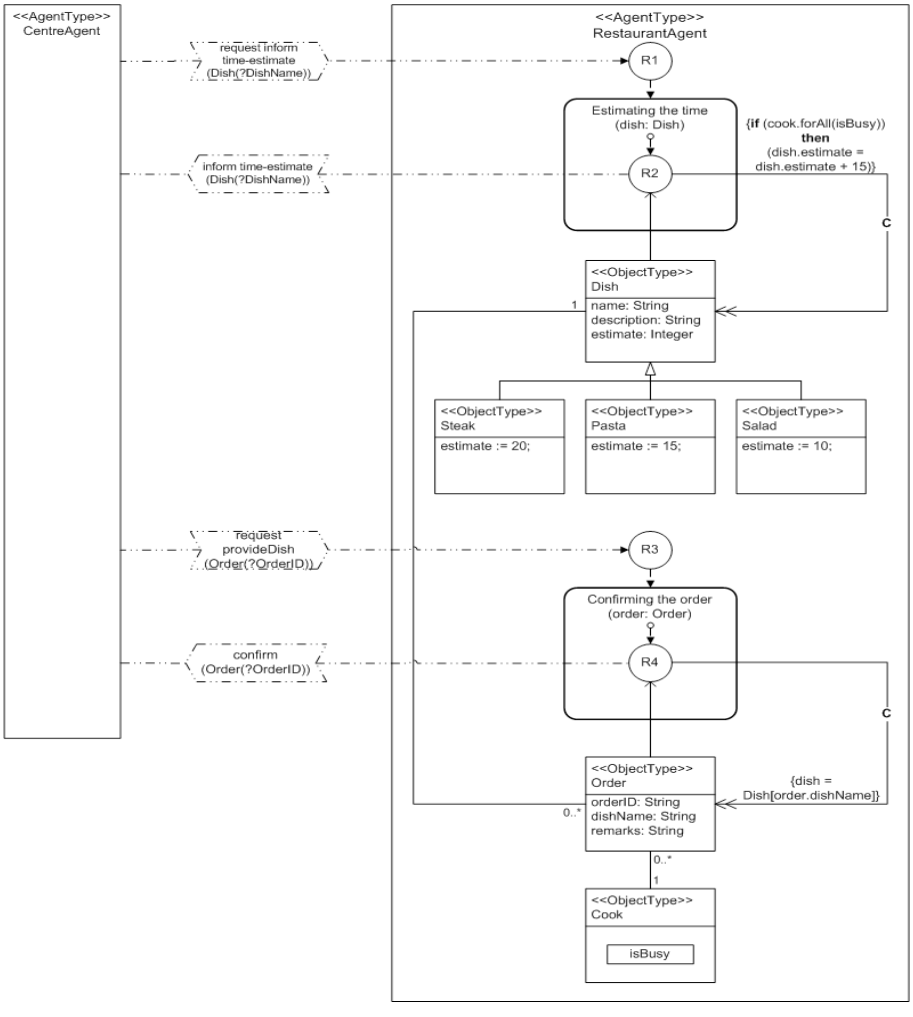

Fig. 6. The Behaviour Model for an agent representing a restaurant

an activity of the type "Estimating the time" represented in Figure 6 achieves a goal to 'provide waiting estimate' modelled in Figure 2. *Rules* determine when, by whom, and under which conditions an activity is invoked. For example, rule `R1` specifies that an activity of the type "Estimating the time" is started by the `RestaurantAgent` upon receiving from the `CentreAgent` a request to provide the waiting estimate. Rules also carry out social policies. For example, rules `R1, R2, R3`, and `R4` shown in Figure 6 realize the social policy "Fast reply".

Figure 6 represents the Behaviour Model of a `RestaurantAgent` type in the scenario of ordering take-away food. The behaviour involves the activity types "Estimating the time" and "Confirming the order". An activity of the type

"Estimating the time" is started by rule R1, which is triggered by a communicative action event (message) of the type request inform time-estimate (Dish(?DishName)). As has been pointed out in Section 3.1, with this message, the CentreAgent requests the RestaurantAgent to inform it about the estimated waiting time required to prepare and deliver the meal that is identified by a serialized object of the type Dish. Rule R2 prescribes an instance of the object type Dish to be created from the serialized object. As there can be three different types of dishes in our example, an instance of Dish created by rule R2 always belongs to one of the subtypes Steak, Pasta, or Salad. It can be seen in Figure 6 that each of them is modelled with the respective value of the attribute estimate. Additionally, there is an Object Constraint Language (OCL) [12] clause specifying that if all the cooks are busy at the time of creating an instance of Dish, represented by the predicate isBusy of the RestaurantAgent's private object type Cook, the value of the attribute estimate should be increased by 15. Rule R2 further specifies that a modified instance of the object type Dish should be serialized and sent to the CentreAgent.

An activity of the type "Confirming the order" is started by rule R3. This rule processes a serialized instance of the object type Order, which is included by a message of the type request provideDish(Order(?OrderID)). The message means that the CentreAgent requests the RestaurantAgent to perform a physical action of the type provideDish(Order(?OrderID)) according to the enclosed order. Rule R4 prescribes an instance of the internal object type Order to be created from the serialized object. At the creation of an Order instance, the value of its identifying attribute orderID will be automatically generated. The OCL clause dish = Dish[order.dishName] specifies the creation of the association link between the order and the corresponding instance of Dish. Rule R4 further expresses through its connection to the message type confirm(Order(?OrderID)) that a modified instance of the object type Order should be serialized and sent to the CentreAgent. In a later stage of the business process of ordering take-away food, an association between the order and the object representing the cook to which the order is allocated will be created.

4 Platform Specific Design and Rapid Prototyping

Finally, the modelling constructs of PIM are mapped to the corresponding constructs of PSM. It has been shown in [9] that external AOR diagrams can be straightforwardly mapped into the programming constructs of the Java Agent Development Environment (JADE, http://jade.cselt.it/) agent platform. The JADE agent platform [13] is a software framework to build agent-based systems in the Java programming language in compliance with the standard proposals for multi-agent systems by the Foundation for Intelligent Physical Agents (FIPA, http://www.fipa.org/). The mapping principles are more particularly addressed in [9].

Table 3. Mapping of notions of KPMC agents to the object classes and methods of JADE

Notion of KPMC agent	Object class in JADE	Object method of JADE
Object type	java.lang.Object	-
Agent type	jade.core.Agent	-
Elementary activity type	jade.core.behaviours. OneShotBehaviour	-
Sequential activity type	jade.core.behaviours. SequentialBehaviour	-
Parallel activity type	jade.core.behaviours. ParallelBehaviour	-
Execution cycle of a KPMC agent	jade.core.behaviours. CyclicBehaviour	-
Waiting for a message to be received	jade.core.behaviours. ReceiverBehaviour	-
Starting the first-level activity	jade.core.Agent	public void addBehaviour (Behaviour b)
Starting a sub-activity	jade.core.behaviours. SequentialBehaviour	public void addSubBehaviour (Behaviour b)
Starting a parallel sub-activity	jade.core.behaviours. ParallelBehaviour	public void addSubBehaviour (Behaviour b)
Start-of-activity event type	jade.core.behaviours. OneShotBehaviour	public abstract void action()
Start-of-activity event type	jade.core.behaviours. SequentialBehaviour, jade.core.behaviours. ParallelBehaviour	public abstract void onStart()
End-of-activity event type	jade.core.behaviours. Behaviour	public int onEnd()
Agent message	jade.lang.acl.ACLMessage	-

Table 3 shows how various modelling notions of KPMC agents can be mapped to the corresponding object classes and methods of the JADE platform. In particular, activity types and the execution cycle of a KPMC agent map to JADE *behaviours*. Rules are not included in Table 2 because they are mapped to various constructs represented in Java. The programs resulting from the mappings are complemented by simple graphical user interfaces and thereafter executed, as is exemplified by a snapshot shown in Figure 7.

Table 3 does not include the mapping of OCL clauses. We used OCL clauses for representing pre- and post-conditions, which specify the state of the world before and after triggering a rule without considering *how* the desired state of the world will be achieved. This feature of being "side-effect free" is one of the basic features of OCL. The particular way of changing the world state is specified

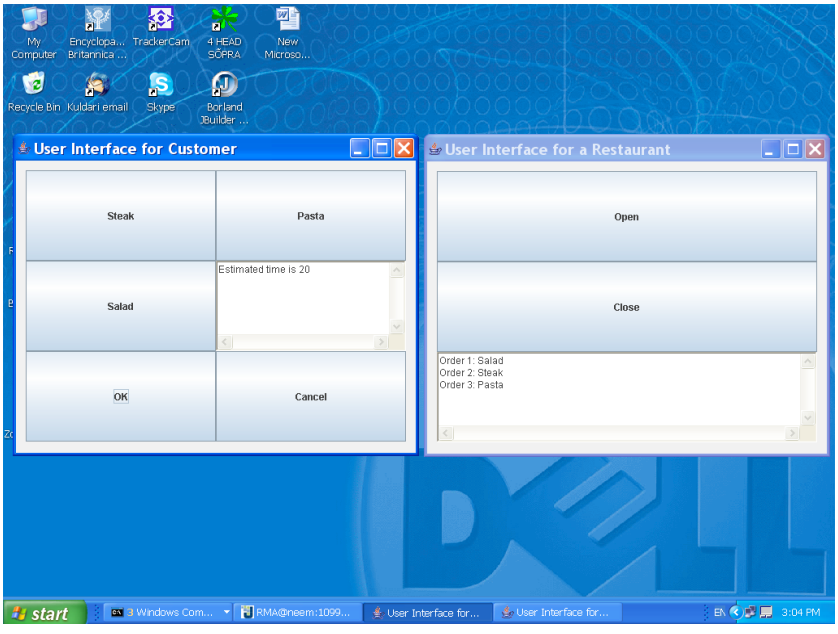

Fig. 7. A snapshot of the prototype created from the CIM and PIM models

only at the PSM level in terms of the constructs of a particular platform, which in our case study was JADE.

The first author has shown in earlier work [11,27] how to represent external AOR diagrams by a graphical tool, enabling mappings into equivalent XML-based representations that are then interpreted and executed by software agents. Since the authors of this paper no longer have access to that tool, we have mapped manually the models for the case study of the take-away food ordering system. However, this was not hard because of the intuitiveness and straightforwardness of the mappings under discussion.

5 Related Work and Conclusions

We have described a technique that maps models of a problem domain into the platform-independent design models of a socio-technical system created for that domain, and from the design models to the a system implementation on a specific platform. The mappings are straightforward, which has been achieved by making use of agent-oriented analysis and design models, as well as of an agent-based implementation platform. Representing the design models in a single diagram increases the transparency of the mappings.

This paper was triggered by the approach to prototyping described in [2]. While the message sequence charts used in [2] are claimed to represent requirements, we believe they are essentially design models. Our technique, on the

contrary, starts with modelling requirements at a high level of abstraction that is understandable to both domain experts and software engineers. We acknowledge that we fall short of [2] in fully automated generation of models from design models. However, as has been shown in [11,27], this is not hard to accomplish with our approach, which we plan to do in the near future.

We emphasise that the contribution is that we can generate prototypes rapidly from high-level requirements prior to commitments to detailed design decisions. Other agent-oriented methodologies tend to concentrate on an ultimate agent implementation, and have not focussed on early rapid prototyping. While in principle this may be possible, for example, generating prototypes from Prometheus system overview diagrams [14], not all information such as agent beliefs have been identified or are available during requirements analysis and high level design.

Because of limited space, we confine specific comparisons with related work to other MDA-related model mapping techniques. CIM models employed in [15] represent agent component types, such as belief, trigger, plan, and step. Jayatilleke et al.'s approach assumes from the very beginning that a system will be implemented as a software agent system. However, in our view this is a *design decision*, which should be postponed until the design phase. Considering this, the starting point for our approach entails technology-independent notions of goals, roles, social policies, and domain entities. Differently from us, the approaches described in [28] and [29] address only mapping from PIM to PSM in the context of software agent systems, while our approach has a more generic software engineering stance.

In [25], agents in domain modelling are described in terms of their capabilities, which are then mapped into plans consisting of activities. Differently from [25], we view activities as fundamental concepts. This enables to distinguish between *contextual, goal-oriented, and routine activities.* The notion of norms used in [26] is roughly equivalent to what we mean by rules. However, we think that the work reported in [26] could benefit from the precise modelling of actions and events adopted by us.

In summary, our technique can be used for rapid production of prototypes from agent-oriented models. The technique has been used in industry-related projects of business-to-business electronic commerce [11,27], manufacturing simulation [24], and future home management [23]. We are currently applying the technique in a research project with industry dealing with airport simulation and optimisation.

References

1. MDA Guide Version 1.0.1. Retrieved February 3, 2007, from `http://www.omg.org/cgi-bin/doc?omg/03-06-01`
2. Barak, D., Harel, D., Marelly, R.: InterPlay: Horizontal scale-up and transition to design in scenario-based programming. IEEE Trans. Soft. Eng. 32(7), 467–485 (2006)
3. Juan, T., Sterling, L.: The ROADMAP meta-model for intelligent adaptive multi-agent systems in open environments (Revised Papers). In: Giorgini, P., Müller, J.P., Odell, J.J. (eds.) AOSE 2003. LNCS, pp. 826–837. Springer, Heidelberg (2004)

4. Kuan, P.P., Karunasakera, S., Sterling, L.: Improving goal and role oriented analysis for agent based systems. In: Proceedings of the 16th Australian Software Engineering Conference (ASWEC 2005), Brisbane, Australia, 31 March – 1 April 2005, pp. 40–47. IEEE Computer Society Press, Los Alamitos (2005)
5. Castelfranchi, C., Falcone, R.: From automaticity to autonomy: The frontier of artificial agents. In: Hexmoor, H., Castelfranchi, C., Falcone, R. (eds.) Agent Autonomy, pp. 103–136. Kluwer Academic Publishers, Dordrecht (2003)
6. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI architecture. In: Allen, J., Fikes, R., Sandewall, E. (eds.) Proceedings of Knowledge Representation 91 (KR-91), pp. 473–484. Morgan Kaufmann, San Francisco (1991)
7. Shoham, Y.: Agent-Oriented Programming. Artificial Intelligence 60(1), 51–92 (1993)
8. Wagner, G., Schroeder, M.: Vivid agents: Theory, architecture, and applications. Journal of Applied Artificial Intelligence 14(7), 645–675 (2000)
9. Taveter, K.: A multi-perspective methodology for agent-oriented business modelling and simulation. PhD thesis, Tallinn University of Technology, Estonia (ISBN 9985-59-439-8) (2004)
10. Henderson-Sellers, B., Giorgini, P. (eds.): Agent-oriented methodologies. Idea Group (2005)
11. Taveter, K., Wagner, G.: Towards radical agent-oriented software engineering processes based on AOR modelling. In: [10], pp. 277–316
12. Unified Modeling Language: Superstructure. Version 2.0 (August, 2003), Retrieved February 5, 2007 from `http://www.omg.org/cgi-bin/doc?ptc/2003-08-02`
13. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi-agent systems with a FIPA-compliant agent framework. Software - Practice and Experience 31, 103–128 (2001)
14. Padgham, L., Winikoff, M.: Developing intelligent agent systems. John Wiley & Sons, Chichester (2004)
15. Jayatilleke, G.B., Padgham, L., Winikoff, M.: A model driven component-based development framework for agents. Comput. Syst. Sci. & Eng. 20(4) (2005)
16. Kuutti, K.: Activity Theory as a potential framework for human-computer interaction research. In: Nardi, B. (ed.) Activity Theory and Human Computer Interaction, pp. 17–44. MIT Press, Cambridge (1995)
17. Rahwan, I., Juan, T., Sterling, L.: Integrating social modelling and agent interaction through goal-oriented analysis. Comput. Syst. Sci. & Eng. 21(2), 87–98 (2006)
18. DeLoach, S.A., Kumar, M.: Multi-agent systems engineering: An overview and case study. In: [10], pp. 317–340
19. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Multi-agent systems as computational organizations: The Gaia methodology. In: [10], pp. 136–171
20. Caire, G., Coulier, W., Garijo, F., Gomez-Sanz, J., Pavon, J., Kearney, P., Massonet, P.: The MESSAGE methodology. In: Bergenti, F., Gleizes, M.-P., Zambonelli, F. (eds.) Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook, pp. 177–194. Kluwer Academic Publishers, Dordrecht (2004)
21. Cossentino, M.: From requirements to code with the PASSI methodology. In: [10], pp. 79–106
22. Iglesias, C. A., Garijo, M. The agent-oriented methodology MAS-CommonKADS. In: [10], pp. 46–78.

23. Sterling, L., Taveter, K.: The Daedalus Team. Building agent-based appliances with complementary methodologies. In: Tyugu, E., Yamaguchi, T. (eds.) Knowledge-Based Software Engineering: Proceedings of the Joint Conference on Knowledge-Based Software Engineering, Tallinn, Estonia, August 28-31, 2006, pp. 223–232. IOS Press, Amsterdam (2006)
24. Taveter, K., Wagner, G.: Agent-oriented modelling and simulation of distributed manufacturing. In: Rennard, J.-P. (ed.) Handbook of Research on Nature Inspired Computing for Economy and Management, pp. 541–556. Idea Group (2006)
25. Penserini, L., Perini, A., Susi, A., Mylopoulos, J.: From stakeholder intentions to software agent implementations. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 465–479. Springer, Heidelberg (2006)
26. Kasinger, H., Bauer, B.: Towards a model-driven software engineering methodology for organic computing systems. In: Hamza, M.H. (ed.) Computational Intelligence: IASTED International Conference on Computational Intelligence, Calgary, Alberta, Canada, July 4–6, 2005, pp. 141–146. IASTED/ACTA Press (2005)
27. Taveter, K.: A Technique and Markup Language for Business Process Automation. In: Proceedings of the Workshop on Vocabularies, Ontologies, and Rules for The Enterprise (VORTE 2006), held in conjunction with the Tenth IEEE International EDOC (The Enterprise Computing) Conference, Hong Kong, 16–20 October 2006, IEEE Computer Society Press, Los Alamitos (2006)
28. Perini, A., Susi, A.: Automating model transformations in agent-oriented modeling. In: Müller, J.P., Zambonelli, F. (eds.) AOSE 2005. LNCS, vol. 3950, pp. 167–178. Springer, Heidelberg (2006)
29. Hahn, C., Madrigal-Mora, C., Fischer, K., Elvester, B., Berre, A.-J., Zinnikus, I.: Metamodels, models, and model transformations: Towards interoperable agents. In: Fischer, K., Timm, I.J., André, E., Zhong, N. (eds.) MATES 2006. LNCS (LNAI), vol. 4196, pp. 123–134. Springer, Heidelberg (2006)