From Agent Interaction Protocols to Executable Code: A Model-driven Approach

(Extended Abstract)

Christian Hahn, Ingo Zinnikus, Stefan Warwas, and Klaus Fischer
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
Saarbrücken, Germany
{christian.hahn,ingo.zinnikus,stefan.warwas,klaus.fischer}@dfki.de

ABSTRACT

In this paper, we demonstrate how to design protocols with the platform independent modeling language for multiagent systems (DSML4MAS) and discuss a model-driven approach to use protocol descriptions as a base for generating the corresponding agent behaviors which can finally be executed with Jack Intelligent Agents.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Language

Keywords

Agent Interaction Protocols, Model-driven Code Generation, Model Transformation

1. INTRODUCTION

In this paper, we demonstrate how to design protocol-based interactions using a platform independent modeling language for the domain of MASs called DSML4MAS. In [1] an overview of DSML4MAS and a direct model transformation to the execution platform of Jack Intelligent Agents is given. However, especially in a more business-oriented context or open MAS, partners often first define the manner in which they interact followed by defining the behaviors that actual implement the agreed interactions. Hence, in this paper, we discuss a methodology-like approach, where a model transformation takes a protocol description and generates a corresponding behavior description which is then in combination with manually added design transformed to Jack code.

Cite as: From Agent Interaction Protocols to Executable Code: A Model-driven Approach, (Extended Abstract), Christian Hahn, Ingo Zinnikus, Stefan Warwas, Klaus Fischer, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009),* Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 1199–1200

2. DOMAIN SPECIFIC MODELING LANGUAGE FOR MULTIAGENT SYSTEMS

The domain specific platform independent modeling language for MASs (DSML4MAS) defines a graphical language that allows define MASs independent of any existing agent-oriented programming language (AOPL). However, model transformations can be applied to automatically generate code in accordance to the AOPLs Jack and Jade. The abstract syntax of DSML4MAS is defined by a metamodel called PIM4Agents that consists of several aspects (e.g. agent, organization, environment, role, interactions, behaviors, etc.) each focusing on particular core building blocks of MASs. The PIM4Agents bases on Ecore the meta-metamodel of the Eclipse Modeling Framework[1]. A more complete overview on the remaining aspects can be found in [1, 2].

3. CONTRACT NET PROTOCOL

In the following, we demonstrate how to model AIPs in particular the Contract Net Protocol (CNP) using DSML4MAS's graphical editor (see [3] for detailed information on the graphical editor). For this purpose, we firstly introduce two actors called *Initiator* and *Participant*. Each of them have a certain numbers of message flows available that are responsible for sending and receiving messages. The protocols starts with the first message flow of the *Initiator* that is responsible for sending the *CallForProposal* message of the performative type *cfp*. The *CallForProposal* message specifies the task as well as the conditions that can be specified within the properties view of the graphical editor. When receiving the *CallForProposal*, each agent instance performing the *Participant* decides on the base of free resources whether to *Propose* or *Refuse*. However, how this selection function is defined cannot be expressed in the protocol description, as private information are later on manually added to the automatically generated behavior description. To distinguish between the alternatives, two additional actors (i.e. *Propose* and *Refuse*) are defined that are subactors of the *Participant* (i.e. any agent instance performing the *Participant* should either perform the *Propose* or *Refuse* actor). The transitions between the message flow within the *Participant* and the message flows of the *Refuse* or *Propose* actors through the *messageFlow* reference underlining the change of state for the agent instance performing the *Participant* actor. The message flows within the *Refuse* and *Propose* actors are then

[1] http://www.eclipse.org/modeling/emf/

responsible for sending the particular messages (i.e. *Refuse* and *Propose*).

After the deadline expired (defined by the *TimeOut*) or all answers sent by the agent instances performing the *Participant* actor are received, the *Initiator* evaluates the proposals in accordance to a certain selection function, chooses the best bid(s) and finally assigns the actors *BestBidder* and *RemainingBidder* accordingly. Again, the selection function is not part of the protocol, but can be defined later on in the corresponding plan. Both, the *BestBidder* actor as well as the *RemainingBiddder* actor—containing the agent instances that were not selected—are again subactors of the *Propose* actor. The *Initiator* sends an *AcceptProposal* message to the *BestBidder* and a *RejectProposal* message to the *RemainingBidder* in parallel.

After completing the work on the assigned task, the *BestBidder* reports its status to the *Initiator* either in the form of an *InformResult* or *InformResult* message to report the successful completion or in the form of a *Failure* message in case the *BestBidder* fails to complete the task. Therefore, we distinguish again between *InformResult*, *InformDone* and *Failure* actors which are subactors of *BestBidder*.

4. MODEL-DRIVEN METHODOLOGY TO GENERATE EXECUTABLE CODE

The process of generating executable code based on a protocol description is discussed in this section in more detail. This methodology consists of two phases. In a first step, a transformation from the protocol description to the internal behaviors of the participating agents is discussed. In a second step, the transformation from the behavioral perspective to the agent-based execution platform Jack is given. Both transformations were implemented utilizing the Atlas Transformation Language[2].

4.1 From Interaction Protocol to Behaviors

The first step of the interaction to behavior transformation is to generate a *Plan* for each *MessageFlow* an Actor is active. At this, however, we do not map *MessageFlows* that have a parent *MessageFlow* (through the *messageFlow* reference of *MessageFlow* where a super actor is active. In this case only one *MessageFlow* is finally generated. The different actions the subactors can potenially take is represented by a *Decision* in the corresponding *Plan*. The generated *Plans* corresponding to *MessageFlows* of the same active *Actor* can be included in a *Capability* which is provided by the particular *Actor* and used by the *AgentInstances* bound to that *Actor*. Within a *Plan*, the order of sending and receiving *Messages* is deduced from the order in which the corresponding *MessageScopes* are arranged and to which *Operations* (i.e. *None, Parallel, Loop* or *Sequence*) they refer. In particular this means that a *Parallel* operation is mapped to a *Parallel* activity, a *Loop* operation is mapped to a *Loop* activity, etc. As a *None* operation only refers to one *Message*, it is either mapped to a *Receive* or *Send* task, depending on whether the corresponding *MessageFlow* sends or receives a message. The *Send* or *Receive* activity than refers to that *Message*. As *Messages* in the *Protocol* can be sent and received to/from multiple *Actors* that again contain multiple agent types, both, *Send* and *Receive*, have to be included in a parallel statement (i.e. *ParallelLoop*) that iterates over all entities. This allows to keep a *Plan* as generic as possible, as the information how many agent instances are finally playing the particular *Actors* needs not be known at design time. Finally, a *TimeOut* is mapped in the manner that a *Wait* activity is introduced and integrated into a *Parallel* activity consisting of two paths. One path includes the *Wait* activity, the other one includes the *Activities* responsible for sending and receiving messages.

[2]http://www.eclipse.org/m2m/atl/

4.2 From Behaviors to Jack

The conceptual transformation from the behavioral metamodel of DSML4MAS to the Jack process metamodel is defines as follows. For each *Plan* generated by applying the interaction to behavior transformation, a *TeamPlan* is generated. The body of this *TeamPlan* is generated in an one-to-one manner at least for those activities in PIM4Agents that were directly supported by Jack. An example is the *Send* activity which is transformed to a *SendNode* in Jack. The *Message* a *Send* refers to is directly transformed to the corresponding *Event*. As a *TeamPlan* automatically handles an *Event*, we do not need to directly transfer *Receive* activities of DSML4MAS to related concepts in Jack. However, as *Plans* in DSML4MAS base on *MessageFlows* which clearly describe which *Messages* are received and sent, we can easily generated the *TeamPlan*-specific structure. Concepts like *Parallel*, *Decision* and *Wait* can also be mapped in an one-to-one fashion. In the contrast to concepts like *Loop*, *ParallelLoop* and *Sequence* which are not directly supported by Jack. In the case of *Sequences* this can easily be compensated by connecting the predecessor of a *Sequence* with the first activity within the *Sequence* and the last activity of a *Sequence* with its successor. For the concepts of *Loop* and *ParallelLoop*, we define templates consisting of several activities of the process view (e.g. *Decision* concept used for looping purpose) and filled with the activities contained by the source concepts (i.e. *Parallel* and *ParallelLoop*).

5. CONCLUSION & FUTURE WORK

This paper discusses an approach to describe agent interactions in a protocol-based manner. For this purpose, we illustrated how to use DSML4MAS to design the Contract Net Protocol. Furthermore, we discussed how to transform agent interaction protocols designed to executable behaviors by applying principles of model-driven development.

6. REFERENCES

[1] C. Hahn. A domain specific modeling language for multiagent systems. In L. Padgham, C. P. Parkes, J. Mueller, and S. Parsons, editors, *Proceedings of 7th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2008)*, pages 233–240, 2008.

[2] C. Hahn, C. Madrigal-Mora, and K. Fischer. A platform-independent metamodel for multiagent systems. *International Journal on Autonomous Agents and Multi-Agent Systems*, 18(2):239–266, 2008.

[3] S. Warwas and C. Hahn. The contrete syntax of the platform independent modeling language for multiagent systems. In *Proceedings of the Agent-based Technologies and applications for enterprise interOPerability (ATOP 2008) at AAMAS 2008*, 2008.