Modeling and Executing Service Interactions using an Agent-oriented Modeling Language

Christian Hahn[1] and Ingo Zinnikus[1]

DFKI GmbH
Stuhlsatzenhausweg 3
66123 Saarbrücken
{Christian.Hahn,Ingo.Zinnikus}@dfki.de

Abstract. Modern information systems are considered as collection of independent units called services that interact with each other through the exchange of messages. This paper focuses on interactions from a more centralized or global perspective (i.e. choreography), validates the underlying approach to model interactions, and discusses how choreographies can be executed with an established agent-oriented programming language basing on the principles of model-driven development.

1 Introduction

Service-oriented architectures (SOAs) as an approach to design and implement modern information systems (ISs) aim to support business process management within an organization and across organizational borders. At this services are employed to perform tasks within these processes and processes themselves can be exposed as services. In these kinds of settings, service interactions are at the center of attention where two complementary perspectives can be distinguished.

Recently, several approaches have been proposed to describe the interaction between entities either from a local (e.g. Business Process Execution Language) or global perspective (e.g. Web Service Choreography Description Language). In this paper, we propose an agent-based approach as agent systems provide several built-in features and concepts that allow to execute SOAs in a nice manner (see [1] for more details).

A lot of effort has been undertaken to identify the most common interaction scenarios from a business perspective, which have been published as *service interaction patterns* by Barros et al. [2]. We take these patterns as a base and demonstrate how a platform independent domain specific modeling language for multiagent systems called DSML4MAS fulfills the proposed requirements. Furthermore, we also aim at providing an agent-based model-driven methodology that allows executing choreographies.

In the remainder of this paper we discuss a selected interaction pattern and demonstrate how choreographies can be transformed to executable code by applying principles of model-driven development.

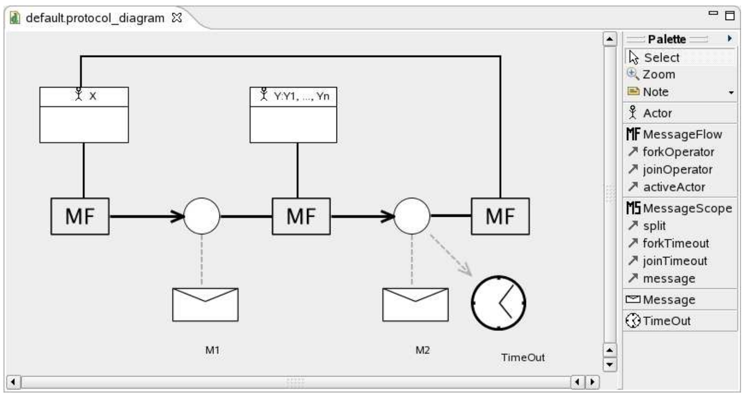

Fig. 1. Pattern 7: One-to-many send/receive modeled with the graphical editor of DSML4MAS.

2 A Platform Independent Modeling Language for Multiagent Systems

DSML4MAS defines a graphical language that could be used to define agent systems independent of any existing agent-oriented programming language (AOPL). However, model transformations can be applied to generate code with respect to selected AOPLs.

DSML4MAS is divided into several viewpoints (e.g. agent, organization, behavior, etc,), however, in this paper we mainly focus on the interaction aspect and demonstrate how to model the proposed service interaction patterns.

In general, an *Interaction* refers to a set of *Messages* and *Actors* that make use of these *Message* for the purpose of interaction. The *Actor* can again refer to a set of *Actors* as subactors, meaning that the set of instances performing the superactor is split into the several subactors. In general, the subactors are determined at design time, but filled with the particular instances that perform this kind of role at run-time. Furthermore, a *Protocol* that should be considered as a specialization of an *Interaction* refers to a set of *MessageFlows* that specify how the exchange of *Messages* is proceed.

The *MessageFlows* again refer to a set of *Actors* that are active in the current state, i.e. those instances that send the particular *Messages*. Furthermore, it specifies a join and fork operator which are both of the type *MessageScope* that defines the *Messages* and their order how these arrive. In particular, this means that *Messages* are connected via a *None, Parallel, Loop, Sequence, XOR*, or *OR* operator. Furthermore, the *MessageFlow* refers to a *TimeOut* that specifies the latest point in time a *Message* should arrive. Beside *Messages* that are sent, the *MessageFlow* may also refer to *Protocols* that are initiated at some specific point in time in the parent *Protocol* in order to execute nested protocols.

3 Modeling Service Interaction Patterns using DSML4MAS

Barros et al. [2] consolidate recurrent scenarios and abstract them in a way that provides reusable knowledge. They distinguish between four groups of patterns, however, we focus on the single-transmission patterns in which a party involved may send or receive multiple messages but as part of different interaction threads dedicated to different parties. A specific case is pattern 7: One-to-many send/receive. Here, a party X sends a request message to several other parties Y1,...,Yn, which may all be identical or logically related. Responses are expected within a given timeframe. However, some responses may not arrive within the timeframe. The interaction may complete successfully or not depending on the set of responses gathered. Fig. 1 depicts the one-to-many send/receive pattern using DSML4MAS. The parties are again modeled as *Actors*, where the atomic entities Y1,...,Yn are bound to *Actor* Y. Sending a *Message* to an *Actor* means that the particular *Message* is sent to each instance that is bound to the target *Actor*. This means that *Message* M1 is sent to each of the Y1,...,Yn in parallel. When receiving M1, each of these entities sends the corresponding answer *Message* M2 to *Actor* X. A *TimeOut* ensures that the interaction does not end up in a deadlock.

4 Model-driven Methodology to Generate Executable Code

In this section, the transformation from the local perspective to the agent-based execution platform JACK is given. JACK is a process-centric agent-based programming language that bases on principle of the belief-desire-intention theory [3]. We firstly introduce the core concepts of JACK. Due to space restrictions this is a very rough summary, however, a detailed overview regarding the JACK metamodel can be found in [4]. The most relevant concept in JACK is the concept of a *Team*, which can be either an atomic *Agent*, or a set of required *Roles* (i.e. subteams) that all together form the *Team*. A *Role* specifies which *Events* the role fillers are able to react to and send. How a *Team* actually reacts to an incoming request is specified by a set of *TeamPlans*.

The transformation to the JACK metamodel uses pattern 7 as an input model. We generate a *Team* for each *Actor* that performs a particular role (e.g. Role_X for team X) and requires *Roles* (e.g. Role_Y) to which the *Messages* in the DSML4MAS behavior model are sent (cf. Fig. 2). The *Messages* of the DSML4MAS model are mapped to *Events* in JACK (e.g. event M1 and M2). For each *Plan* in the behavior model, we instantiate a *TeamPlan* that is used by the particular *Team* (e.g. XSendM1 and XReceiveM2). The body of the *TeamPlans* is mainly generated in an one-to-one manner from the *Plans* in DSML4MAS. For instance, the XSendM1 *TeamPlan* (Fig. 2 right-hand side) also includes a parallel statement that iterates over the various role fillers and sends the event instance m1 of M1 to the role fillers y. The parallel statement ends if the *Event* has been sent to all role fillers. We refer to [4] for more detailed information regarding the model transformation from DSML4MAS to JACK.

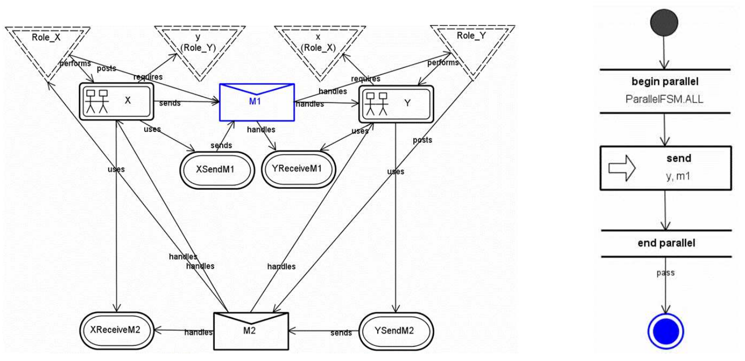

Fig. 2. The generated JACK models based on the DSML4MAS behavior for Pattern 7.

5 Conclusion

This paper discusses an agent-based approach to describe choreography-based interactions. Therefore, we proposed a modeling language for multiagent systems called DSML4MAS and demonstrated that DSML4MAS supports modeling the proposed service interaction patterns. The main result of this evaluation is that each pattern—in contrast to other proposed standards—can nicely be described.

Based on DSML4MAS, we discussed a model-driven methodology to derive code based on the choreography description. The DSML4MAS model that includes the particular generated behavior model is mapped to an agent-based programming language JACK that finally executes the choreography description.

References

1. Zinnikus, I., Hahn, C., Klein, M., Fischer, K.: An agent-based, model-driven approach for enabling interoperability in the area of multi-brand vehicle configuration. In: Proceedings of the 5th Conference on Service-Oriented Computing. Volume 4749 of Lecture Notes in Computer Science., Springer (2007) 330–341
2. Barros, A.P., Dumas, M., ter Hofstede, A.H.M.: Service interaction patterns. In van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F., eds.: Business Process Management. Volume 3649. (2005) 302–318
3. Rao, A.S., Georgeff, M.P.: Modeling agents within a BDI-architecture. In Fikes, R., Sandewall, E., eds.: Proceedings of the 2rd International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann (1991) 473–484
4. Hahn, C.: A domain specific modeling language for multiagent systems. In: Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS). (2008) (accepted).