Modelling Interactions for Automatic Execution Using UML Activity Diagrams

Werner Putschögl[1] and Bernhard Dorninger[2]

Abstract: Software applications involving interactions of distributed systems are very common nowadays. Frequently, interactions are modelled during the analysis phase of a project and subsequently coded manually. Very often this results in a divergence of analysis model and the implemented code. Moreover, the border between interaction handling code and domain code may be blurred. In addition, hard-coding collaborations and interactions may impede maintainability of an application. In this paper, we propose a procedure for modelling interactions—and also collaborations—with the means of slightly extended UML activity diagrams. The resulting interaction model is then transformed to a machine interpretable format and may subsequently be processed and controlled by an interaction infrastructure, which we developed for this purpose. In addition, our procedure encourages a clear separation of interaction processing and domain code.

1 Introduction

Designing distributed, collaborative applications poses a demanding challenge. Major concerns include coordinating work among the participating nodes and the distribution of data needed and produced by the nodes. These concerns do apply for autonomous multi node systems as well as common client-server applications involving human-computer interaction. One means to cope with these challenges can be model-driven software development (MDSD), which nowadays is well established due to its various benefits [Se03]. Especially the Unified Modelling Language (UML) has emerged as the lingua franca for modelling the various aspects of software. Nearly every UML tool is capable of generating code at least from static models (class diagrams). There are also numerous tools that support the generation of code from behavioural models. However, UML [OMG09a] is often deemed insufficient for use in the context of MDSD [SV06]. Also, modelling interactions between autonomous systems and/or UI based applications with UML is not always considered adequate.

Frequently, interactions—especially in the field of business processes and workflows—are initially modelled via use cases and later detailed by activity diagrams (e.g. during requirements engineering). Use cases and activity diagrams have been criticized to lack formal semantics to generate code—although there are several proposals for enriching activities to solve this [SH05] [BS09]. A modelling method intended solely for business process modelling is Business Process Modelling Notation (BPMN) [GDW09]. Interactions can also be modelled with UML collaboration/communication diagrams [CBJ02], but are used to visualize object level collaborations rather than high level

[1] Software Competence Center Hagenberg, Softwarepark 21, 4232 Hagenberg, Werner.Putschoegl@scch.at
[2] Software Competence Center Hagenberg, Softwarepark 21, 4232 Hagenberg, Bernhard.Dorninger@scch.at

interactions. The Human Computer Interaction (HCI) community also has offered several suggestions to extend UML with mechanisms supporting the modelling of interactions [SP03] [PBL03]. These methods focus on the modelling of user interface related aspects of interactions and/or were developed for communicative purposes rather than for generating code or executable information.

Often, the analysed model of interactions is lost as the design and implementation process progresses, especially when interactions are coded manually. Modelled interactions are split and refined to various domain or infrastructure objects and functions in the codebase [CBJ02]. This results in a lack of traceability from the modelled interaction to the code. This problem is reinforced by the fact that implementations often mix up interaction related code with functional code and infrastructure related code. It may lead to highly complex method/function implementations in the collaborative peers, which in turn decreases maintainability as well as comprehensibility. In addition, it makes interactions and domain functions difficult to reuse in other scenarios. The principles of Domain Driven Design (DDD) [Ev04] may help here, which suggest a clear separation of domain code from infrastructure code.

In this paper we outline an approach of how to avoid the aforementioned downsides by proposing a procedure of interaction modelling with regard to a strict decoupling of interaction and domain functions and preserving the modelled information in the code.

2 Goals and Challenges

The primary goal of our work described in this paper is to provide a procedure for modelling and implementing interactions between software systems as well as a reusable software infrastructure for processing the modelled interactions.

It is desired that efforts for implementing and maintaining collaborations/interactions is kept low. Thus, implementation of interactions shall be automated as far as possible. Manual coding shall be reduced to the need of implementing the content of domain operations, whereas interaction related code or interaction descriptions shall be generated from predefined models. The procedure shall prescribe a clear separation of interaction and domain/business functions and preserve this separation in the resulting implementation. On the other hand, the key here is to provide traceable and comprehensible information concerning the flow of interaction and the relationship of actions to domain operations. Finally, modelling shall be based on a standardized technique. Possible enhancements shall be as simple as possible and shall not depend on a specialized modelling tool.

The interaction infrastructure shall not only suggest the compliance to an architectural pattern [CBJ02] or concentrate on the protocol or application level communication layer [ASQP05] but rather provide a reusable mechanism for processing any interaction or collaboration scenarios.

To satisfy these requirements, we have to cope with two key challenges.

- Distribution of control: Processing interactions in collaborative systems certainly involves the changeover of control. Each participant has its tasks to solve and actions to perform, which may depend on the actions and decisions of the fellow participants. We have to find a way to control the flow of interactions independent of a concrete interaction, i.e. the infrastructure shall be able to handle arbitrary interaction scenarios.
- Distribution of interaction data: Each action or decision within an interaction needs and/or produces data. This data may be needed by subsequent actions and/or decisions in the interaction flow. Since the interaction is distributed over potentially more than two nodes, we have to secure the availability of the needed data in the respective participant. Of course data distribution and availability shall be addressed already in the model and shall not lie in the responsibility of the programmer.

The remainder of this paper outlines the developed procedure and explains the necessary steps from analysis of the interaction to deployment of the solution.

3 Creating an Executable Interaction Model

In the software development process, the analysis phase usually results in a more or less detailed model of the respective domain, consisting at least of a data model and the usage scenarios of the planned software. Scenarios are documented in form of use cases, which may be represented by textual descriptions, use case diagrams and even behavioural diagrams like activity diagrams (AD). These use cases implicate necessary domain operations (aka business logic) as well as interactions between systems or users and a system.

In this section, we will outline a procedure, which builds on the results from the analysis phase by augmenting AD with additional information and subsequently generating an interpretable representation of interactions. For executing the interactions, an infrastructure was developed managing the distributed execution of an interaction at runtime. Figure 1 illustrates the fundamental steps of our procedure, structuring the procedure into three basic steps:

1. Modelling: The use cases from analysis phase have to be modelled as AD according to specifications we introduce in our procedure. At the same time, domain operations have to be modelled matching the actions from the AD. Next, data used by these operations during the interaction has to be specified. The procedure of modelling is described in detail in Section 3.1.
2. Transformation and Implementation: The goal of this step is the generation of an interpretable representation of the interaction model. Another task is the generation of code stubs for the modelled domain operations from step 1.

Subsequently, the functionality of the generated stubs has to be implemented. Section 3.2 will elaborate on this step.

3. Application: To verify the results of our efforts, we developed an interaction infrastructure, which is responsible for distributed invocation and execution of modelled interactions. This infrastructure ensures that each participating system processes the assigned actions, executes the respective domain operation and is supplied with data required from other participants. The concept of the infrastructure and its processing of interactions is explained in Section 3.3.

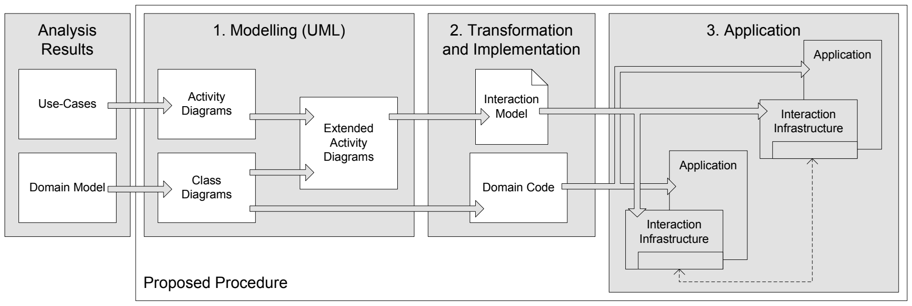

Fig. 1: Procedure Overview

The following sections describe each step in detail by using a simple example. We will demonstrate the way from use case descriptions to executable representations of the implicated interactions, and how interactions are executed and controlled by the infrastructure.

3.1 Modelling

The modelling step builds on the artefacts of the analysis phase and can be divided into three sub-steps, illustrated in Fig. 2. The figure depicts the artefacts required and generated in each sub-step.

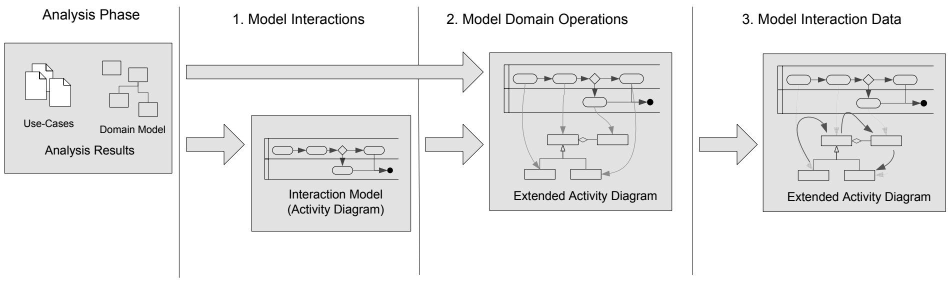
Analysis Phase
Use-Cases
Domain Model
Analysis Results
1. Model Interactions
Interaction Model (Activity Diagram)
2. Model Domain Operations
Extended Activity Diagram
3. Model Interaction Data
Extended Activity Diagram

Fig. 2: Modelling Overview

The analysis phase provides us with use cases, which implicate domain operations and optional domain rules their execution is subjected to. Domain rules are conditions which have to be fulfilled before an operation can be executed. In addition we get a data model of the domain. These artefacts form the basis of our four modelling sub-steps.

1. Model Interactions: If not already created during analysis, the interactions described in use cases have to be transformed into activity diagrams following certain rules (see Section 3.1.1).
2. Model Domain Operations: Since we want to automatically execute and control the modelled interactions, our procedure prescribes the presence of domain operations for each modelled activity or domain rule from an activity diagram. Therefore, in this step, we explicitly model the domain operations in a static view, in our case an UML Class diagram. In doing so, we consider the recommendations of the Domain Driven Design approach [Evans]. Finally each activity and decision node is assigned to the appropriate domain operation. This connects the dynamic view of the activity diagram with the static view of the class diagram. The step "Model Domain Operations is described in detail in Section 3.1.2.
3. Model Interaction Data: Finally, we have to model the data, which is distributed between the individual operations. Furthermore, we have to concretize our domain rules in a machine-recognizable way. Section 3.1.3 describes this step in detail.

After these steps, the model is ready for transformation.

3.1.1 Model Interactions

We use a simple example of a flight booking interaction to demonstrate our proposed procedure. Listing 1 shows the use case gained from the analysis phase. Fig. 3 shows the activity diagram of the interaction based on the use case shown in Listing 1.

Flight booking:

Main Course:

1. The user chooses a flight to book.
2. The system asks the user for the number of passengers.
3. The user enters the number of passengers.
4. The system ensures that there are enough places vacant, reserves the places and books the flight.
5. The use-case ends successfully.

Alternative Courses

4.a.: The check fails as there are not enough seats available.

4.a.1. The system informs the user.

4.a.2. The user chooses to pick another flight

List. 1: Flight Booking Use Case

The example includes actions that are performed by a user on a client system, actions performed by the systems as well as a simple domain rule. This domain rule indicates that there must be sufficient seats available to continue with the reservation. In addition, it serves as precondition for the reserving and booking actions. The “Check Availability” decision (Fig. 3) evaluates this domain rule.

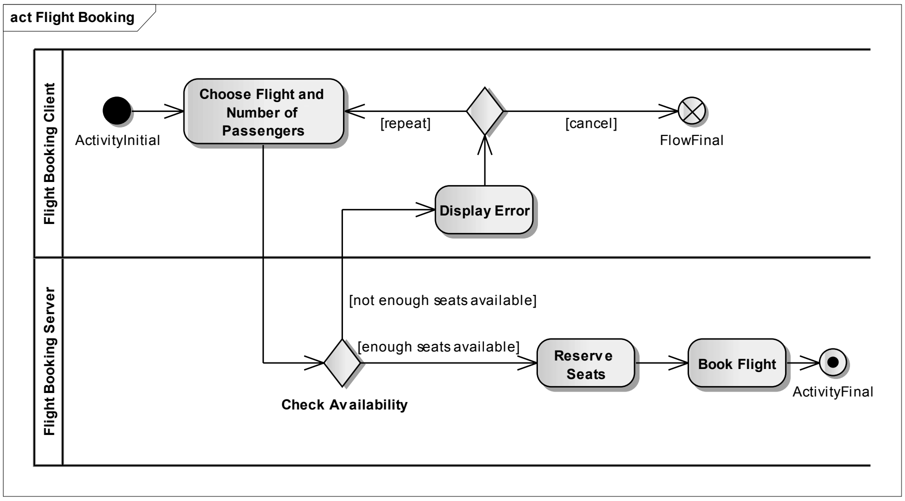

Fig. 3: A Basic Flight Booking Example

Distributed execution requires a mechanism which specifies where an operation is going to be processed. Our procedure introduces system roles to identify different systems in an interaction. A system role is a classifier such as “Flight Booking Client” or “Flight Booking Server”, not a specific physical system, as for instance there can be multiple client systems connected to one server in the above example. In the AD we use partitions to model these system roles, each partition representing one role. Each diagram element has to be put in one partition. In the example shown in Fig. 3 each element is put in one of the two partitions specifying that the “Flight Booking” action is to be executed on a “Flight Booking Server”-type system.

Domain rules are represented by a decision node. To enable an automated evaluation of the decision each outgoing control flow has to be modelled using mutually exclusive conditions. At this stage of modelling, guards can only be expressed by abstract, textual conditions. These guard expressions will be refined into machine-recognizable expressions after specifying the interaction data (see Section 3.1.3).

3.1.2 Model Domain Operations

Having modelled the dynamic view with activity diagrams is not sufficient for our purpose. We also need to model the operations, which will be executed when processing an interaction. When modelling applications, it is common to have static views depicting the domain's data model as well as operations on these data. Concerning operations, our procedure suggests applying the principles of Domain-Driven Design (DDD) [Ev04]. Evans introduces the so called "Service" pattern, which proposes the modelling of domain operations, which manipulate other domain objects (such as data or resources), as stateless services. We follow this view and expect the domain operations to be modelled according to this pattern. In addition, we model each operation as an interface, since our procedure requires the presence of an appropriate domain operation for each action in the activity diagram. To be able to process domain rules within an interaction we apply the "Specification" pattern proposed by Fowler and Evans [EF97]. It describes the modelling of domain rules as interfaces just like operations. The pattern allows integrating domain rules into our interaction model with their execution being similar to the execution of domain operations.

After having modelled the domain operations and rules we create an explicit relation between the dynamic view from the activity diagrams and the static view of the operations in the class diagrams. This is done by mapping each activity element and its respective domain operation (or the shortcut thereof).

Fig. 4 shows how the "Reserve Seats" action from the activity diagram is mapped to the "`ReserveSeatsOperation`" interface via an explicit link. The result of the link is that upon processing the interaction, the operation implemented in the interface is executed when the action is processed.

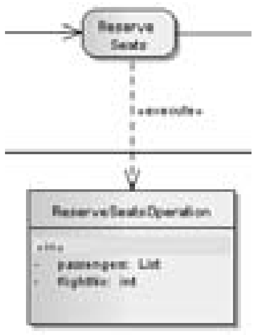

Fig. 4: Example Link

3.1.3 Model Interaction Data

The next task is to specify the data used in the context of the interaction. As already mentioned the distribution of data to systems involved in an interaction is a key challenge. We have to model data that will be exchanged by the operations and rules during the execution of interaction. The entirety of data exchanged between operations or between operations and rules within one interaction—which may span several system roles—is called *interaction context*. Data is specified as properties of the modelled domain operations or rules. Data properties of operations must be stereotyped as input or output (or as both if this applies). The result of an evaluation of a domain rule is regarded an output data property. Once defined, each data property in the interaction context is available for all other operations and rules within the regarded interaction. The mapping of data properties between the operations and rules is achieved via lexical identity of the property names.

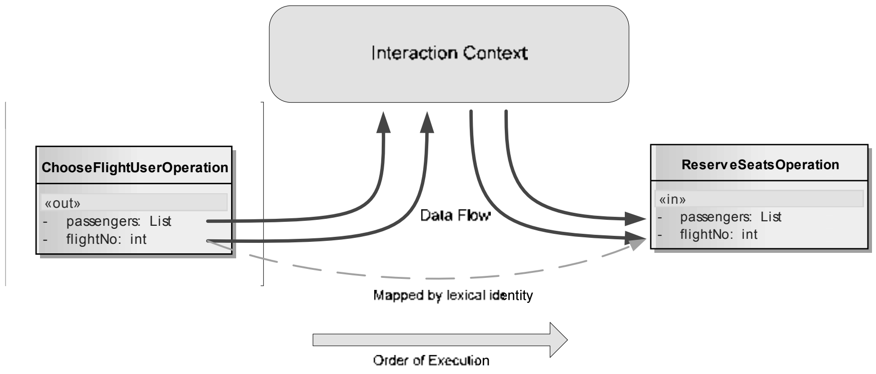

Fig. 5: Data Supply Example

In Fig. 5 the solid line arrows depict the data flow between two operations (via the interaction context) during the execution of an interaction. The action `ChooseFlightUserOperation` produces the data *flightNo*, which is required by the operation `ReserveSeatsOperation`. The mapping of domain operation data is achieved by lexical identity of the property names. This is indicated by the dashed line arrow. Any domain operation in the same interaction seeking to access *flightNo,* would have to define it as a `<<in>>` or `<<in,out>>` property.

Only data exchanged between operations and rules has to be specified here. Additional data required privately by a domain operation should not be part of the model.

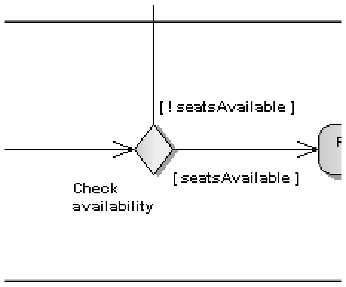

Fig. 6: Modelling Guard Conditions

After the interaction context has been defined, we now have to adapt the guard conditions of our domain rules to match the appropriate data. In addition, we mandate the guards being expressed in a machine-interpretable way for allowing their automatic evaluation. Fig. 6 displays an example.

The Boolean data property "*seatsAvailable*" is the result generated by evaluating the preceding domain rule. Knowing the data property name, the guard condition must be transformed manually from a textual definition to a machine processable condition using the correct property name and values.

3.2 Transformation and Implementation

The interaction model and the domain operations created during modelling form the base for the step *Transformation and Implementation*. This step includes two sub-steps:

1. Model Transformation: The UML model is transformed into a XML representation of the interaction model used as input for the infrastructure.
2. Generation and Implementation: Based on the interfaces modelled to represent operations and rules, interfaces and class stubs are generated. The generated stubs have to be completed by manually coding the required functionality of the domain operations and domain rules.

This section describes the sub-steps to generate the artefacts required to be used by interaction infrastructure.

3.2.1 Model Transformation

For the model transformation, we define our own metamodel representing interactions. The next step is to export the AD modelling the interaction from the modelling tool and transform it into our metamodel. The XML Metadata Interchange (XMI) standard defined by the Object Management Group is widely supported by UML Modelling tools for exchanging and exporting model data. Since XMI is very verbose, we extract the relevant information concerning the interaction and transform it to an XML representation of our own metamodel (Fig. 7).

Our metamodel is loosely based on the UML metamodel [OMG09b], but of course takes a simplified view suitable for our needs. All elements of the metamodel representing actions or decisions implement the *InteractionElement* interface. This implies that all elements have an identifier unique for the interaction as well as a target role, defined by the partitions in the UML interaction model. The current metamodel supports two types of interaction elements: *ActionElements* and *DecisionElements*. *SpecificationElements* differ from *ActionElements* only in that *ActionElements* execute domain operations whereas *SpecificationElements* evaluate domain rules. The information required for both comprises the domain operation or rule to process including the data properties modelled and the succeeding element. For decisions associated with a domain rule a *SpecificationElement* is generated followed by a *DecisionElement.* The *DecisionElement* contains the guard conditions determining the possible successors.

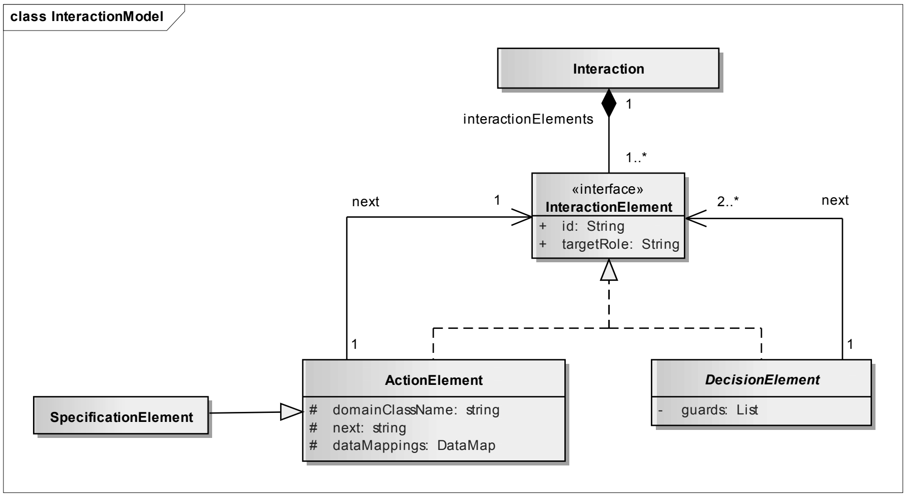

Fig. 7: The Interaction Metamodel

List. 1 is a short excerpt of the transformed XML file showing the "Reserve Seats" operation from the example.

```
…
<actionelement id='Reserve Seats' targetrole='Flight Booking Server'
operation='ReserveSeatsOperation' next='Book Flight'>
    <property name='flightNo' kind='in'>
    <property name='passengers' kind='in'>
</actionelement>
…
```

List. 1: Excerpt of generated XML.

3.2.2 Generation and Implementation

Based on the modelled domain operations and rules, interfaces and class stubs are generated. The domain operations and rules represented by the class stubs have to be implemented for the system they are intended for. This is done by manually adding the code of the functionality to the generated stubs. For data properties modelled in the interaction, get and set methods are generated automatically. The interaction infrastructure uses these to supply or retrieve data required by other operations or rules within the same interaction. Access to additional data necessary for an operation has to be coded as needed.

3.3 Application and Execution of Interactions

After having generated interpretable interaction definitions and the implementation of the relevant domain code (operations and rules), it is desired to execute these interactions. We developed an infrastructure to invoke and control the modelled interactions using Java. The infrastructure may be embedded into any Java based application, where it handles control flow, data supply and synchronization for the participating systems. Control flow management involves a mechanism keeping track of the progress of an invoked interaction and controlling which element has to be executed by which system. Execution encompasses calling the associated domain operation or evaluating a rule as well as supplying the required data.

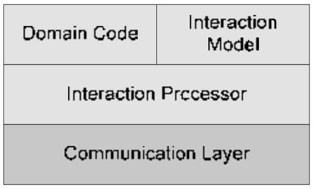

Fig. 8: Infrastructure Layers

The interaction infrastructure can be separated into two layers: the Interaction Processor and the Communication Layer (Fig. 8). The Domain Code and Interaction Model have to be created individually for each application.

The Interaction Processor is responsible for building an object representation from the generated interaction definition.

It furthermore has to handle the processing of the interactions, which includes data supply as well as the execution of domain operations or rules. The Interaction Processor also manages the execution focus, i.e. it has to take care that each domain operation or rule is executed by the system defined in the model. To achieve this, the processor uses the Communication Layer. The Communication Layer defines an interface that has to be implemented to connect the systems participating in an interaction. All technology dependent functionality is encapsulated in the Communication Layer. Our prototypical implementation relies on Java RMI, but other implementations using different technologies (such as CORBA) are feasible without changing the Interaction Processor.

Deployment and Configuration

To make use of the interaction infrastructure each of the systems requires deployment information in addition to the interaction model and the domain code. Deployment information has to be present for each participating system in an interaction. It includes the respective participant's system ID, its system role and information concerning the other potentially participating systems. The system roles are determined by the different partitions used in the interaction model. The system ID is an abstract, technology independent name for uniquely identifying a specific system within the set of participants. Deployment information also has to contain a mapping from logical system IDs to physical system addresses. This information depends on the communication protocol used by the Communication Layer. In our prototypical implementation we map the logical system IDs to IP addresses.

Another issue is the distribution of deployment information. For our prototype the deployment information has been provided using one configuration file for each system participating in an interaction. This is error-prone and may prove inappropriate for more complex systems. Registry servers may be an alternative to local configuration files with the deployment information being automatically downloaded.

Interaction Invocation and Execution

Upon start-up of an application, the interaction infrastructure parses the deployment information and initialises the application's system role and ID. The infrastructure loads the interaction definitions from the XML file and constructs the corresponding interaction element graphs conforming to the metamodel from Fig. 7. The Communication Layer has to be configured with the deployment information about possible interaction participants. For all systems the system ID, system role and physical address has to be known to enable a connection.

An application invokes an interaction by passing the interaction's unique name to its embedded interaction processor. Subsequently, an interaction context for that specific interaction is created, which is valid for one invocation only.

The interaction context contains all information required for the execution of the interaction. The most important part of this information is the data exchanged by the domain operations and rules. When invoking an interaction, this data is extended by additional runtime information, which encompasses system role to ID mappings as well as the progress of the interaction.

Each role of an interaction has to be assigned to one specific system. Upon invocation of the interaction, the infrastructure assigns the role in the interaction to its system ID. This assignment does not change for the duration of the interaction invocation. After assigning the system role, the infrastructure starts to traverse the interaction object graph. If the current interaction element is to be executed locally, the interaction processor instantiates the according domain operation class and sets the data properties defined upon modelling. The data is read from the interaction context and passed to the domain operation or rule by invoking its setter methods. Then, the domain operation or rule is executed. After successful completion, the values of the operation or rule' output data properties are retrieved and written to the interaction context. Finally, the progress indicator in the interaction context is advanced to the successor of the just processed interaction element

If the current interaction element has to be executed by a system with another system role, the execution focus must be transferred to an appropriate participant. If the needed role has not been assigned to a system yet, the infrastructure tries to assign the role automatically. Ambiguities (i.e. there are more participants potentially fulfilling a role) can be resolved by predefining the participating systems upon invocation or by routing algorithms making the choice. The interaction infrastructure provides extension mechanisms to support routing.

The execution focus is transferred by transmitting the interaction context containing interaction data, system role to ID mappings and the current progress indicator to the system that has been assigned the role the current interaction element was modelled in. The Communication Layer translates the system ID to an actual physical system and performs the context transfer. To reduce the network load, only new or changed data in the context is transferred. The interaction infrastructure of the target system is responsible for merging the changes into its interaction context. Once the other participant has the execution focus, it may continue with processing the interaction until it runs into an interaction element, which again needs to be processed by another participant. While the interaction is continued by other systems the local application waits for the interaction context and execution focus to return or the interaction to end.

This process is repeated until the interaction ends successfully or an error occurs. Upon termination, all participants are informed about the end of the interaction and if it was successful or not. If possible, information about the error is added to the context to provide feedback.

4 Discussion

The goal of our work described in this paper was to provide a modelling procedure combined with a reusable infrastructure to process and control interactions and/or collaborations between distributed systems. While there are a number of possible approaches to model interactions such as UMLi [SP03] or MoLIC [PBL03], we use activity diagrams, since UMLi and MoLIC both focus on human computer interaction (HCI). UMLi provides its own diagram type for modelling interactions between users and the user interface. MoLIC also introduces diagram types for modelling interactions. It focuses on the user’s actions and goals and deals with system actions (domain operations in our sense) from a very abstract point of view. Moreover, MoLIC does not intend to generate machine-executable or interpretable interaction definitions, but classifies itself as a means of communications between software development and potential users. We take a different approach to interaction modelling, as we do not aim at UI design, but primarily focus on the aspect of distributed execution and control of interactions. . Interactions in our sense may be also addressed as collaborations. They take place between arbitrary systems, which may incorporate UI clients as well as autonomous systems. We concentrate on modelling the distribution of activities and actions and the corresponding data. Moreover, we did not want to introduce new types of diagrams

Strengths

We argue that using a de-facto standard modelling language is an advantage of our procedure. We refrain from introducing special diagram types, which allows utilizing readily available tools without further adaptations. Another benefit we see in our procedure is that we require only few rules to cover all aspects for a distributed execution of interactions, keeping our procedure simple.

Using a separate, reusable infrastructure has two major advantages: it reduces the effort for implementation as the functionality for executing and controlling interactions as well as distributing corresponding data is already provided. Furthermore, it supports architectural layering suggested by Evans [Ev04]. We extend Evans' suggestion with the inclusion of interactions as a separate layer. Having a layered architecture increases flexibility by limiting the dependencies to a few well known points, thus improving maintainability and facilitating changes.

As propagated in MDSD [SV06], models are considered equal to code. We follow a similar point of view, generating a machine-readable representation of our models and subsequently use an infrastructure to interpret them. As with MDSD, our procedure mandates the (transformed) model as an integrated part of an interactive or collaborative application avoiding extra effort to keep the documentation up-to-date.

Limitations

Besides its strengths, the procedure has some weaknesses which need further consideration.

One major issue concerns the current form of modelling interaction data. Lexical mapping for connecting data properties is error prone as even simple typos may cause fatal runtime errors. Furthermore, when modelling complex interactions, the sheer number of properties might become increasingly confusing. The improvement to modelling interaction data should provide a clear and straightforward method to model data properties while not cluttering the model.

Another aspect has to do with distribution of data. At the moment, interaction data is distributed over system boundaries, regardless if any interaction element executed in the context of a system needs to access a specific data property at all. In other words, interaction data is available globally. An improvement would be to distribute data only to systems, which in fact need to access it. The information of data property usage by systems participating in an interaction could already be drawn from the current model, yet this information is not considered upon transforming the model or at execution time.

A third potential improvement would be supporting parallel execution of interaction elements within an interaction. This applies for parallel execution of interaction elements on one system as well as on different systems. At present, only one system can hold the execution focus at any given time during the processing of an interaction. While this suffices for most cases of client-server applications, collaborations of autonomous systems may require such a mechanism. A key challenge regarding this issue will be the merging of data written by different execution paths to guarantee consistent data for the interaction.

5 Summary

In this paper we have presented a procedure based on descriptions of interactions in form of activity diagrams to create a machine interpretable model of interaction. We described how UML activity diagrams could be extended to attain such a model.

Despite a number of issues, we think this procedure allows us to achieve our goals of reducing the effort to create and maintain interaction implementations and having a clear separation between domain-code and interaction handling. The procedure proposed does not require specialised modelling tools or new diagram types but only extensions to activity diagrams. We introduced a simple metamodel for representing interactions and explained the transformation of the created UML model into a machine-readable instance of this metamodel.

We also presented a prototypical implementation of a reusable infrastructure which takes the transformed model as input, allowing the distributed execution of the modelled interactions. We explained how this prototype handles the challenges of coordinating distributed interaction execution and distributing data required during the process.

Acknowledgement

This project was sponsored by the initiative “Regionale Wettbewerbsfähigkeit OÖ 2007-2013” funded by the European Regional Development Fund and the state of Upper Austria.

References

[ASQP05] Joao Paulo Almeida, Marten van Sinderen, Dick A.C. Quartel, and Luis Ferreira Pires. Designing Interaction Systems for Distributed Applications. IEEE Distributed Systems Online, 6(3), 2005.

[BS09] Anup Kumar Bhattacharjee and R.K. Shyamasundar. Activity Diagrams: A Formal Framework to Model Business Processes and Code Generation. Journal of Object Technology, 1:189–220, January-February 2009.

[CBJ02] E. Cariou, A. Beugnard, and J.M. Jezequel. An Architecture and a Process for

Implementing Distributed Collaborations. In Enterprise Distributed Object Computing Conference, 2002. EDOC ’02. Proceedings. Sixth International, pages 132–143, 2002.

[PBL03] María Greco de Paula, Simone Diniz Junqueira Barbosa, and Carlos José Pereira de Lucena. Relating Human-Computer Interaction and Software Engineering Concerns: Towards Extending UML through an Interaction Modelling Language. In Proceedings of the IFIP INTERACT 2003 Workshop, 2003.

[SP03] Paulo Pinheiro da Silva and Norman W. Paton. User Interface Modeling in UMLi. IEEE Software, 20(4):62–69, 2003.

[EF97] E. Evans and M. Fowler. Specifications. In Proceedings of PLoP 97 Conference, 1997.

[Ev04] Eric Evans. Domain Driven Design: Tackling Complexity in the Heart of Business Software. Addison-Wesley, 2004.

[GDW09] A. Grosskopf, G. Decker, and M. Weske. The Process: Business Process Modeling using BPMN. Meghan Kiffer Press, 2009.

[OMG09a] OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.2, 2009.

[OMG09b] OMG Unified Modeling Language (OMG UML), Superstructure Version 2.2, 2009.

[Se03] Bran Selic. The pragmatics of model-driven development. IEEE Software, 20(5):19–25, 2003.

[SH05] Harald Störrle and Jan H. Hausmann. Towards a formal semantics of uml 2.0 activities. In Software Engineering, pages 117–128. Gesellschaft fuer Informatik, 2005.

[SV06] Thomas Stahl and Markus Voelter. Model-Driven Software Development. John Wiley & Sons, 2006.