Project DEPLOY
Grant Agreement 214158

“Industrial deployment of advanced system engineering methods for high productivity and dependability”

DEPLOY Deliverable D21

D4.1 Report on Pilot Deployment in Business Information Sector

Public Document

31 January 2010

http://www.deploy-project.eu
Contributors:
Andreas Roth  SAP
Vitaly Kozyura  SAP
Wei Wei  SAP
Sebastian Wieczorek  SAP
Andreas Fürst  ETHZ
Thai Son Hoang  ETHZ
Jeremy Bryans  NCL

Editors:
Andreas Roth  SAP

Reviewers:
Jeremy Bryans  NCL
Renaud de Landtsheer  CETIC
Abstract

The deliverable D4.1 reports on the pilot deployment for the Business Information Sector. It summarises the work done in the first two years of the project for the business information sector led by SAP. The pilot deals with modelling service choreographies. The approach taken is to reuse existing development models for service choreographies which are written using a domain specific diagrammatic modelling language and translate these models automatically to Event-B. The proofs, validation and exploitation of the models (e.g. to produce test cases) take place in the background. Topics of relevance include how to guide the developer through patterns and improve the degree of automation, as well as how to give modellers feedback on the diagrammatic level.
For the chapters 1, 2, 3, 4, 7, 8, and 9, as well as Sections 10.2 and 10.3.1 the following Copyright notice applies:

Copyright 2010 SAP AG.
All rights reserved.
No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice. Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors. SAP, R/3, xApps, xApp, SAP NetWeaver, Duet, SAP Business ByDesign, ByDesign, PartnerEdge and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other product and service names mentioned and associated logos displayed are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary. The information in this document is proprietary to SAP. This document is a preliminary version and not subject to your license agreement or any other agreement with SAP. This document contains only intended strategies, developments, and functionalities of the SAP® product and is not intended to be binding upon SAP to any particular course of business, product strategy, and/or development. SAP assumes no responsibility for errors or omissions in this document. SAP does not warrant the accuracy or completeness of the information, text, graphics, links, or other items contained within this material. This document is provided without a warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. SAP shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials. This limitation shall not apply in cases of intent or gross negligence. The statutory liability for personal injury and defective products is not affected. SAP has no control over the information that you may access through the use of hot links contained in these materials and does not endorse your use of third-party Web pages nor provide any warranty whatsoever relating to third-party Web pages.
## Contents

1 **Summary** 6  
   1.1 Business Software 6  
   1.2 Pilot Deployment Strategy 7  
   1.3 Pilot: Choreography Models 8  
   1.4 Extensions of the Pilot 9  
   1.5 Experiences and Challenges 10  
   1.6 Overview 11  

2 **Pilot Description** 12  
   2.1 Summary 12  
   2.2 Choreography Modeling 13  
      2.2.1 A simple buyer-seller example 13  
      2.2.2 Motivation for Using a Domain-specific Modeling Lan- 
             guage 14  
      2.2.3 MCM Structure 16  
      2.2.4 MCM Syntax 18  
      2.2.5 MCM Viewpoint Semantics 21  
      2.2.6 Pilot Instances 23  
   2.3 Opportunities for Formal Methods 23  
      2.3.1 Enforcing Model Consistency 24  
      2.3.2 Goal of the Pilot 25  

3 **Translating Choreography Models to Event-B** 26  
   3.1 Summary 26  
   3.2 A Translation from MCM to Event-B 26  
   3.3 Implementation of the Translation 30  

4 **Refinement between Global Choreography Models and Local Partner Models** 32  
   4.1 Summary 32  
   4.2 Local Enforceability and Absence of Inconsumable Messages 33
4.3 Gluing Invariants ........................................ 34
  4.3.1 Types of Gluing Invariants .......................... 34
  4.3.2 Some Rationale of the Types of Gluing Invariants .... 36
  4.3.3 Automatic Generation of Gluing Invariants .......... 37
4.4 Integration in Diagrammatic Front-End .................. 37
4.5 Verification Procedure .................................. 39
4.6 Experience ............................................. 40

5 Middleware Models in Event-B 41
  5.1 Study 1: Middleware Models ............................ 43
  5.2 Study 2 ................................................ 44
  5.3 Study 3 ................................................ 45

6 Application of Patterns 47
  6.1 Application to Selected Choreography Models .......... 47
    6.1.1 Phase 1 ........................................... 49
    6.1.2 Phase 2 ........................................... 49
    6.1.3 Discovering Patterns ............................... 50
    6.1.4 Single-Message Communication ...................... 51
    6.1.5 Multiple-Message Communication ..................... 51
    6.1.6 Question/Response .................................. 52
    6.1.7 Refinement Strategy ................................ 52
    6.1.8 Proof Statistics .................................... 56
  6.2 Integration with Diagrammatic Front-ends .............. 56
    6.2.1 Description ........................................ 58
    6.2.2 Patterns .......................................... 59
    6.2.3 Matching as Seen from Choreography Model ........... 60
    6.2.4 Conclusion ......................................... 62

7 Feedback Loop for Diagrammatic Modelling 64
  7.1 Summary ................................................ 64
  7.2 Simulation ............................................. 65
  7.3 Model Checking and Error Visualisation ............... 66
    7.3.1 Visualisation Strategies .......................... 66
    7.3.2 Realisation ....................................... 72
    7.3.3 Example Application ............................... 74
  7.4 Towards Visualisation Support for Proofs .............. 76
    7.4.1 Running Example and Proof Assistance ............. 76
    7.4.2 Automatic Trace Derivation from MCM Proof Obligations ... 81
    7.4.3 Example Application ................................ 83
8 Model-based Integration Testing 85
  8.1 Summary ................................. 85
  8.2 Requirements ............................. 86
  8.3 Model-based Integration Testing with Event-B and ProB .... 89

9 Results and Lessons Learnt 93
  9.1 Appropriateness of Deployment Strategy .................. 93
  9.2 Suitability of Choreography Modelling as Pilot Application ... 94
  9.3 Suitability of Event-B for the Pilot ....................... 94
  9.4 Suitability of Rodin for the Pilot ......................... 95
  9.5 Suitability of WP4 Adaptations ......................... 96
    9.5.1 Suitability of Model Simulation and Model Checks ... 96
    9.5.2 Proof Assistance Tools ....................... 97
    9.5.3 Model-based Testing ......................... 97

10 Outlook 99
  10.1 General Directions .............................. 99
  10.2 Modelling of Business Objects ...................... 99
  10.3 Modelling of Business Processes ..................... 101
    10.3.1 Towards faithfully reflecting BPMN models in Event-B . 102
    10.3.2 Ensuring Consistency of Business Processes ........ 103
Chapter 1

Summary

Business software is any software which helps companies improve their business. In contrast to typical areas of application of Formal Methods, this area has little, e.g. safety critical, aspects which have been significant incentives to make the use of formal methods attractive, as e.g. in the transportation industry. On the other hand business software is often highly mission critical and customers expect high qualitative software which is efficiently developed — goals Formal Methods could be helpful to achieve. Nevertheless Formal Methods are (to our knowledge) not used routinely in the development of business software.

1.1 Business Software

Business software is a wide field, covering

- technical components (e.g. process integration, master data management, etc.)
- business applications (e.g. Enterprise Resource Management (ERP), Customer Relations Management (CRM), Supply Chain Management (SCM), etc.)
- industry solutions (adaptations of business applications for e.g. Banking, Automotive, Chemicals, Retail, Public Sector, etc.)
- analytical applications (tools for collecting, integrating, analysing, interpreting and presenting business data)

Formal Methods could play a role in all of these areas, but business application development is a—though so far widely ignored—promising domain of Formal Method application. The main reason is that business application development
incorporates the routine development of hundreds of structurally similar components. This allows us to focus on domain-specific requirements and helps to come from one application of Formal Methods to the next one because the method can be adapted to these specifics. There is thus a high potential of “second usage” of Formal Methods in that area. Moreover, business applications incorporate interesting “business logic” of processes without dealing (in general) with complex algorithms. The reasoning on a “model-level”, without the need to go formally down to source code, is already a welcome support to further increase development efficiency. Applying Formal Methods in this area seems thus to be relevant, sustainable, and feasible.

Finally, a good tradition of (informal) model-based development exists [KP09] which can be built on. Especially model content which is very expensive to collect is often already present and can be re-used via generating formal models from existing ones. The re-use and integration of such already existing methods seems to us a key factor to achieve deployment of Formal Methods in the presence of successful and established development processes.

We have been conducting a main pilot study on formal modelling in the area of service-based systems and two further rather experimental studies extending this (and which we intend to continue in the future). With the help of these we investigate the use of Formal Methods for Business Applications.

1.2 Pilot Deployment Strategy

Introducing formal modelling and verification into the development of business information application is a challenging task. There are specialised established domain-specific languages for the development of this software in place which developers need to use. Switching to non-domain-specific languages and working with mathematical syntax is typically considered not to be feasible.

One approach to make business application developers nevertheless benefit from formal verification techniques is to hide the mathematical formalisms, such as Event-B, behind the languages normally used in business software development and to rely on the high automation of today’s verification tools, so that developers do not have to directly interact with the formalism.

Our approach to hide the mathematical formalism behind domain-specific front-ends imposes a number of challenges. First there must be a semantics preserving translation from the state transition systems of global and local MCM models to Event-B (see Chapter 3 and Chapter 5). Second, since formal verification in Event-B heavily relies on the addition of additional invariants enriching the model, we need to cater for the addition of invariants in MCM (see Chapter 4). Third, one needs to take care that results of the verification process, e.g. failed
proof attempts, are adequately presented in the MCM diagram (see Chapter 7).

1.3 Pilot: Choreography Models

The allowed ordering of messages exchanged between independent service components is usually described in (diagrammatic) message choreography modelling languages [WRS+09] (similar to a specification of a communication protocol).

These models are similar to extended finite state machines and consist of a global model defining the messages exchanged from a global perspective and a local model which describes the send- and receive events of the involved components. The used model input is described in Chapter 2.

We have formalised these models by providing an automatic translation from them to Event-B [AH07]. We report on this translation in Chapter 3 and on the aspect of different middleware configurations in Chapter 5.

Based on this formal model, the following “exploitations” of the Event-B model are possible:

- Checking the local enforceability by proving a refinement relation between the global and a (general) local model. We report on this work in Chapter 4.

- Checking the (global) choreography model for internal well-definedness, such as checking for the absence of inconsumable messages [KRW09], i.e. messages which cannot be received at all times when they are being transmitted. In our work we restrict ourselves to generic properties we require from all such models, but are open for more “instance-specific” properties. This analysis is set into relation with the topic of providing an appropriate feedback loop to the modeller as discussed in Chapter 7.

- Interactively simulating the choreography model by stepping through the diagram. This is also covered in Chapter 7.

- Deriving test cases with a certain model coverage (model-based integration testing) which can then be executed on an implementation of the system which includes the choreography [WKR+09]. This is described in Chapter 8.

All these analyses have been efficiently implemented based on the in-house Eclipse plugin of the choreography model editor and Rodin [ABHV08a]. We made use of the Rodin provers and the ProB model checker [LB08]. Since we deal
with unbounded data types, we use the latter for initial checks (and simulation and test derivation) while the former serves for the final verification (see Sect. 2.3.2).

1.4 Extensions of the Pilot

We have started to investigate ways to extend the pilot in two directions:

- Checking the local models for consistency with component models (business object models).

  Business objects (e.g. a purchase order, a sales order, an invoice, a delivery, a project) are at the core of business applications. Typically these objects are structured as a tree (e.g. a purchase order consists of a root node and a number of (ordered) item nodes). Each of the nodes has a lifecycle where user actions (e.g. create, approve, release) lead a path through a set of status (created, approved, released) of the business object nodes.

  We are conducting experiments with modelling business objects as Event-B models and started to automatically translate models in a proprietary in-house notation to Event-B. Our goals are to (1) prove properties about the models such as “if an invoice is posted all items of the invoice are released” and to (2) prove (via refinement) that the business object conforms to a given message choreography.

- Checking the global model against consistency with the business process it is embedded in.

  Business process models are usually the starting point when designing business applications. They model the (cross-organisational) processes which should be supported by business applications. There are a number of notations for modeling business processes, e.g. BPMN [Obj08], which are mostly not backed by a formal semantics.

  We are experimenting with modelling business processes in Event-B with the goal to (1) automatically translate from an established process modelling language into Event-B and (2) prove properties such as timed data consistency (i.e. consistency among distributed components that is achieved not immediately but after a certain time span) which go beyond classical properties in the verification of business processes.

First details on these extensions are reported in Chapter 10.
1.5 Experiences and Challenges

In the pilot and the experiments mentioned above it was possible to model the given problem with the help of the chosen formal modelling language Event-B. For choreography models and business object models we have realized an automated translation, work on business process models is still work in progress because of the complexity of the used source modelling language BPMN.

It was also possible to mechanically analyse the models. The analysis was based on proving the proof obligations generated by the Event-B modelling platform Rodin with the help of the provers provided there. Though techniques like automatically generating invariants through templates [KR09] helped in closing a considerable amount of proofs automatically, the degree of proof automation is currently not so high that Formal Methods unexperienced users could do the verification work. Pattern-based approaches [HFA09] proved to can further increase proof automation. Progress on this is reported in Chapter 6.

Analyses with the help of the model checker ProB were very helpful during the creation of the models, especially the interactive simulation. We got most positive feedback about the possibility to derive test suites from the models, achieved from using ProB. Using this approach would require least overhead in adapting quality assurance processes in the company because testing is an established QA technique.

A major drawback of our approach to generate a formal model from existing models is that users expect that the feedback from the analysis tools are incorporated in the source model. With the help of the open Rodin platform we could create plugins which extract model checking results, animation snapshots, or even proof results, and mark-up the original diagrammatic model for specific typical cases. However to deal with this in general is a notoriously difficult endeavour which requires further investigations. Our progress on this is reported in Chapter 7.

A further issue is that the central concept of refinement intrinsic to approaches like Event-B is not present in the original diagrammatic models. Therefore our generated Event-B models were initially flat (consisting of one or two layers of refinement). The power of refinement-based approaches to introduce complexity (of proofs) incrementally might thus not be fully exploited yet. Our strategy will be to make use of modelling features (like sub-processes in BPMN) to derive a suitable refinement structure or to introduce these into the modelling language (as done in UML-B [SBS09]). An important aspect to this is that Event-B users need stronger guidelines and documentation from the Event-B community on how to obtain a good refinement hierarchy and what the main rules are to change a refinement hierarchy during a development – this will be a necessary precondition to find good generic refinement strategies for our domain-specific problems.
1.6 Overview

In summary, this deliverable is structured as follows.

Chapter 2 gives an overview of the pilot including the choreography modeling approach used. In Chapter 3 the basis for the analysis, that is the automated translation from the choreography models to Event-B is described. In Chapter 4 we discuss how to come to a model which allows for proving refinement relation by imposing gluing invariants through domain specific patterns. We furthermore report on ways to configure the formalization using different middleware modes (Chapter 5) and using a pattern approach (Chapter 6). Then we investigate the way to communicate with the modeller in our translation based approach, also dealing with properties to establish during choreography modeling in Chapter 7. The model-based integration testing approach based on the generated Event-B models is described in Chapter 8. Chapter 9 shows results and lessons learnt for further extensions of deployment as described in Chapter 10.

In each chapter we give an introduction which summarises the main results. Details are given in the subsequent part of the section.
Chapter 2

Pilot Description

2.1 Summary

Service choreography modelling is an important development step in the development of business applications which are based on service-oriented architectures. It describes the allowed ordering of messages exchanged between independent distributed components. Usually, special ("domain-specific") diagrammatic modelling languages are used to describe this behaviour; the use of such languages increases acceptance of modelling for the developers.

Message Choreography Modeling (MCM [WRSC08]) is a proprietary language developed by SAP Research for modeling service choreographies in the domain of the SAP Application Platform. MCM consist of three model types defining different aspects of service composition.

Global Choreography Model. The global choreography model (GCM) is a labeled transition system specifying a high-level view of the conversation between process components. Its purpose is to define every allowed sequence of observed messages.

Local Partner Model. The local partner model (LPM) specifies the communication relevant behaviour for exactly one of the participating process components. Each LPM is a structural copy of the GCM with additional constraints on some of the local transitions.

Channel Model. The channel model (CM) describes the characteristics of the communication channel on which messages are exchanged between the participating process components. The most relevant modes for the sequencing context of the channel model are: Exactly Once (EO) and Exactly Once in Order (EOIO).

Modelling for business applications requires an incremental development process which supports the creation and exploitation of these models. It is the goal of this pilot to support such a process via Formal Methods as provided by Event-B.
2.2 Choreography Modeling

According to the W3C Web Service Glossary [W3C04] "a choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state". We introduce a motivating example. Then we present the message choreography modelling language MCM as well as its realisation in an MCM editor.

2.2.1 A simple buyer-seller example

In this subsection a running example from the enterprise world is introduced, describing a simplified communication between the service components of a buyer and a seller. The example will be used throughout the deliverable to explain the different aspects of our pilot deployment. The structural information of the communication and its corresponding interaction protocol are depicted in Figure 2.1.

When a buyer is interested in placing a sales order it starts a conversation with the respective seller, by sending a Request message that provides the details of
the order. This message will be answered by the seller using an Offer message with information about the price and terms of delivery for the desired goods. Afterwards the buyer has the choice to either accept or decline the offer. In the first case, it sends an Order message that successfully concludes the communication and triggers the execution of the production and/or delivery process at the seller side. In the other case, it sends a Cancel message that rolls back the previous communication and releases the reserved resources at the seller. In this case the protocol allows the buyer to restart the negotiation with a new request.

Even though the presented protocol description might be precise enough for a high-level business view of the process, some semantical subtleties have to be considered. For example, the semantics of message sending and receiving has to be clearly defined based on the specific channel assumptions. In our example we assume a (reliable) communication channel that is not necessarily preserving the message order. Therefore it might be observed that a Cancel message is delivered to the seller only after it received the Request message of a new negotiation process, even though these messages were sent in the opposite order. Consequently the protocol depicted in Figure 2.1 only applies if we assume that the transitions symbolize the sending of messages.

2.2.2 Motivation for Using a Domain-specific Modeling Language

For modelling choreographies there are the following high-level objectives:

- **Comprehensibility.** Choreography modelling should support process integration experts, developers, and testers to get an unambiguous common picture of interaction of communicating service components. This is a crucial goal for software development with globally distributed development teams.

- **Verifiability.** Choreography models should allow applying static verification techniques to discover inconsistencies in the design of the communication itself and in correlation with the behavioural descriptions of the involved service components. This ensures the discovery of problematic design decision before the actual implementation and hence avoids expensive corrections in later development stages.

- **Suitability for automatic test generation.** Choreography models should enable the derivation of integration tests using model-based testing techniques. After some initial effort for building test adapters, MBT promises an optimised test generation with a controlled test coverage and a high degree of automation.
These objectives raise a number of considerations which advocate the use of a domain-specific language like MCM:

- **Detailed message description.** Determinism is a prerequisite for most MBT approaches. In the context of SOA, the allowed communication sequences may vary depending on certain data values in the exchanged messages. Assuming that the Cancel and Order message from Section 2.2.1 have the same message type, MCM provides detailed message descriptions that help to distinguish them.

- **Infinite state space.** Suppose we want to model that a number of requests are sent asynchronously from the buyer while the seller should respond with the same number of confirmations. This immediately requires to have integer or set-valued variables available in choreography modelling, hence introducing an infinite state space.

- **Interaction termination.** For choreography modelling an important step is to define those states, in which the communication is allowed to terminate. For the communication between buyer and seller the Start state is such a state, even though the communication is allowed to be continued by the buyer. MCM’s notion of termination therefore conflicts with the definition of end states of other choreography languages, where termination means that no further interaction can take place, but it is similar to the classical notion of accepting/final states from finite automata theory.

- **Channel modelling.** Choreography modeling has to reflect the heterogeneous and distributed nature of SOA. In the given example an asynchronous communication channel that does not preserve the message order for the Request and Cancel message sent from the buyer would imply that the implementation of the seller component will have to be more robust as it has to consider receiving a second Request message before a (deprecated) Cancel message.

- **Explicit message send and receive.** Describing a send event together with its corresponding receive event as an atomic action restricts the choreography model significantly as it prohibits to specify that a message is "send after receive" constraints. Considering an atomic modelling approach, for example the following constraint: 'the seller shall not be allowed to send an Offer after receiving a Cancel message’ cannot be distinguished from the following stronger "send after send" constraint: 'the seller shall not be allowed to send an Offer after the buyer sent a Cancel message’. For an unambiguous interpretation of the model therefore the stronger "send after
send" constraint would have to be considered. As this constraint can further only be enforced when both participants synchronize their communication, it also contradicts the SOA paradigm of loosely coupled services.

- **Global and local views.** In order to design, verify and test a service-based application, next to a global model also the envisioned local behavioral models of the participating components have to be given, as they specify the corresponding implementations. Keeping all these perspectives consistent is a major challenge of choreography modeling.

- **Explicit concurrency.** Concurrent interactions between two components exists, if both parties are able to act independently in triggering a message exchange. For interacting business components in an asynchronous environment it is quite a common pattern to negotiate while enjoying equal rights, rather than enforcing actions in a clients-server relation. By providing means for modeling concurrent interactions separately, much redundancy can be omitted in the models.

- **Pairwise choreographies.** Most interaction processes in service-based systems span over multiple components. In the vast majority of cases no information is lost when projecting such multi-party choreography to pairwise choreographies, as the underlying processes are utilizing the components sequentially. Modelling pairwise instead of multi-party choreographies helps to reduce the modeling complexity.

- **State-based modelling.** Two major directions can be followed for choreography modelling: an activity-based or a state-based one. In the activity-based approach, the interactions between the parties and their ordering is the primary focus. In the state-based approach, the states of the choreography are modelled as first-class entities together with the interactions, which are then modelled as transitions between states. Since activity-based models may become cluttered by variables for bookkeeping of the choreography state and the message contents, MCM follows a state-based approach.

### 2.2.3 MCM Structure

MCM consists of different model types each defining distinctive aspects of service choreographies:

- **Global Choreography Model.** The global choreography model (GCM) is a labelled transition system which specifies a high-level view of the conversation between service components. Its purpose is to define every allowed sequence of observed messages.
- **Local Partner Model.** The local partner models (LPMs) specify the communication relevant behaviour for exactly one participating service component. Due to the design process of MCM, each LPM is a structural copy of the GCM with extra constraints on some of the local transitions, usually leading to the affected sending actions being deactivated.

- **Channel Model.** The channel model (CM) describes the characteristics of the communication channel on which messages are exchanged between the service components. Such characteristics are formalised by WS-RM standard [OAS07] and describe for example whether messages sent by one component preserve their order during transmission.

Figure 2.2: GCM (top) of the choreography and LPMs of the buyer (left) and the seller (right)

Figure 2.2 shows how the example described above can be described using the MCM artifacts. In the GCM at the top of Figure 2.2, the arrows labeled with an envelope depict the interactions Request, Offer, Cancel, Order, and Cancel (deprecated) which are ordered with the help of the states Start, Requested, Reserved, and Ordered. The states Ordered and Start are so-called target states (thus connected with the filled circle). Only in these states, the communication between the partners is allowed to terminate.
To keep the model deterministic, a set variable called \( \text{ID\_SET} \) is declared and initialized with \( \emptyset \). It stores the transaction IDs from the header of Request messages that have not yet been addressed by Cancel, Cancel\( (\text{deprecated}) \) or Order messages (the headers of these messages also store the ids). Whenever a Request interaction takes place, an assignment

\[
\text{ID\_SET} := \text{ID\_SET} \cup \{ \text{msg\_Header\_ID} \}
\]

is executed referring to the ID stored in the header of the Request message. This assignment is needed to distinguish between a deprecated and an actual Cancel in state \text{Reserved}. Thus for the interaction Cancel an additional necessary guard

\[
\text{ID\_SET}\{ \text{msg\_Header\_ID} \} = \emptyset \land \text{msg\_Header\_ID} \in \text{ID\_SET}
\]

can be modeled in MCM while for Cancel\( (\text{deprecated}) \) we add the guard

\[
\text{ID\_SET}\{ \text{msg\_Header\_ID} \} \neq \emptyset \land \text{msg\_Header\_ID} \in \text{ID\_SET}.
\]

In Section 2.2.4 the formal syntax and the complete set of guards and assignments for our example is described.

The LPM of the buyer of our example is depicted in the lower left part of Figure 2.2. It is a structural copy of the GCM, but the interaction symbols now represent either send or receive events of the buyer. Moreover some send events are “inhibited” by special local constraints. It is for example inhibited that a Cancel\( (\text{deprecated}) \) is ever sent and that a Request is sent in the \text{Reserved} state. Therefore in the figure these send-events have been erased. However, due to possible message overtaxing on a channel that does not guarantee to enforce the message order during transmission\(^1\), receiving a deprecated Cancel is possible on the seller side. The LPM of the seller is depicted in the lower right part of Figure 2.2.

**MCM Editor.** To guide the creation of MCM models there is an Eclipse-based editor (Figure 2.2 is a collage of screenshots) that is utilizing the Meta-Object Facility (MOF) standard [OMG06] for the definition of the underlying MCM meta-model. The editor is integrated within SAP internal tooling and allows for importing existing SAP models (hence taking full advantage of domain specific languages.

### 2.2.4 MCM Syntax

In this section, we present the abstract syntax of MCM. As explained in Section 2.2.2 we assume that all choreographies consist of exactly two participating components.

\(^1\)In harmony with [OAS07] the channel in Figure 2.2 is therefore called \textit{exactly once}. 

18
Then, a message choreography model $M = (G, L_1, L_2, C)$ consists of a GCM $G$, two LPMs $L_1$ and $L_2$ and a CM $C$. $G$, $L_1$, and $L_2$ are extended finite state machines, i.e. they have additional variables as well as additional guards and actions at their transitions, which may reference these variables. We refer to $L_1$, $L_2$, and $C$ as composed system. For expressing guards and actions, a message constraint language is needed which is informally described below, before giving the whole definition of the global and local models.

**Message Constraint Language.** The messages we are considering are XML documents. Logically they are of some hierarchical record data type representing the schemas the messages comply with. An important feature of the set of terms $\text{Term}$ of our message property language is to reference elements of messages that are structured this way. For example, $\text{msg.Header.ID} \in \text{Term}$ points to the ID of the Header element of the message referenced by the $\text{msg}$ variable. The terms further contain the global variables (defined for the extended state machine, see below) and constants (including e.g. $0, 1, 2, \ldots, \emptyset$, etc.). $\text{Term}$ is closed under application of arithmetic or set-theoretic operators. In a natural way, a simple type system can be built into $\text{Term}$ so that syntactically illegal function applications can be excluded. As an example, assuming that $\text{ID\_SET}$ is a variable, $\text{ID\_SET/msg.Header.ID} \in \text{Term}$.

The set $\text{Form}$ of formulas of the message constraint language is then the set of first order formulas over $\text{Term}$ and the predicates $=, <, >, \subseteq, \in$. For instance

$$(\forall x. x \in \text{order.items} \rightarrow x.\text{status} = \text{Released}) \in \text{Form}$$

illustrates that quantified expressions are rather typical since it is often the case that all positions within a message (here: order) should satisfy a certain condition.

For a more detailed account on the message constraint language please refer to [WKR+09].

**Global Choreography Model.** The GCM $G$ is essentially an extended finite state machine represented by the tuple $(S, E, s_0, T, I)$ where

1. $S$ is a finite set of states;
2. $s_0 \in S$ is the initial state;
3. $T \subseteq S$ is a set of target states;
4. $I \subseteq \mathcal{P}(S) \times S$ is a finite set of interactions indicating the transitions of the state machine.
5. $V$ is a finite set of variables; For each interaction $i \in I$ there is a special variable $msg_i \in V$ referring to the message exchanged during an interaction.

With each interaction $i \in I$ we associate a function $sender(i) \in \{P_1, P_2\}$ that indicates which partner is responsible to send the message of this interaction. Furthermore each interaction $i$ is associated with

- a guard $pre(i) \in Form$ which describes a condition under which the interaction can be observed,
- a side-effect $act(i) \in (V \rightarrow Term)$ which describes assignments of variables from $V$ to terms during the transition.

Furthermore there is an initial assignment of terms to variables $E$.

**Example 1** As explained before, the GCM of our example includes the set $V = ID\_SET$ and the following guards and actions:

$$
\begin{align*}
pre(\text{Request}) &= \text{msg.Header.ID} \notin ID\_SET \\
pre(\text{Order}) &= \text{msg.Header.ID} \in ID\_SET \\
pre(\text{Cancel}) &= ID\_SET \setminus \text{msg.Header.ID} = \emptyset \\
& \quad \land \text{msg.Header.ID} \in ID\_SET \\
pre(\text{Cancel}(\text{depr.})) &= ID\_SET \setminus \text{msg.Header.ID} \neq \emptyset \\
& \quad \land \text{msg.Header.ID} \in ID\_SET
\end{align*}
$$

$$
\begin{align*}
act(\text{Request})(\text{ID\_SET}) &= \text{ID\_SET} \cup \{\text{msg.Header.ID}\} \\
act(\text{Order})(\text{ID\_SET}) &= \text{ID\_SET} \setminus \{\text{msg.Header.ID}\} \\
act(\text{Cancel})(\text{ID\_SET}) &= \emptyset \\
act(\text{Cancel}(\text{depr.}))(\text{ID\_SET}) &= \text{ID\_SET} \setminus \{\text{msg.Header.ID}\}
\end{align*}
$$

**Local Partner Model.** Like a GCM, a local partner model LPM $L_j$ is an extended finite state machine with a finite set $S_j$ of states, an initial state $s_0 \in S_j$ and a set of target states $T_j \subseteq S_j$. In addition it entails two finite disjoint sets $I_j^1$ and $I_j^2$ with $(I_j^1 \cup I_j^2) \subseteq \mathcal{P}(S_j) \times S_j$ of send and receive (resp.) events forming the transitions of the state machine. As in GCMs there is a finite set $V_j$ of variables and for each send/receive event $e \in (I_j^1 \cup I_j^2)$ there is a special variable $msg_e^j \in V_j$, referring to the message sent or received during $e$ as well as a message type $MT(e)$. Guards and side-effects are assigned to send and receive events as for interactions of the GCM.

A basic means to ensure consistency, is to demand that for every send/receive event $!i$ or $?i$ of an LPM there is a corresponding interaction $i$ with the same
message type in the GCM. If it is a send event $!i$ in $L_j$ then $\text{sender}(MT(i)) = j$ and if it is a receive event $?i$ then $\text{sender}(MT(i)) \neq j$. Different other ways to ensure consistency among GCMs and LPMs will be defined later in Section 2.3.1.

**Example 2** The lower part of Figure 2.2 shows the two LPMs $L_1$ (left) and $L_2$ (right) of our running example. For the interaction $\text{Request}$ in GCM, we have $!\text{Request}$ in $L_1$ and $?\text{Request}$ in $L_2$. $L_1$ contains a set $V_1 = \{\text{ID}_\text{SET}1\}$ and $\text{pre}$ and $\text{act}$ of the LPMs are copied accordingly from the GCM, e.g.:

$$\begin{align*}
\text{pre}(!\text{Request}) &= \text{msg}_1.\text{Header.ID} \not\in \text{ID}_\text{SET}1 \\
\text{act}(!\text{Request})(\text{ID}_\text{SET}1) &= \text{ID}_\text{SET}1 \cup \{\text{msg}_1.\text{Header.ID}\}
\end{align*}$$

However not in all cases we want to transform the interactions one-by-one into send/receive events. For instance, it should not be possible to send $\text{Request}$ in the state $\text{Reserved}$. So we have $!\text{Request} = \{\text{Start} \mapsto \text{Requested}\}$ while $\text{Request} = \{\text{Start} \mapsto \text{Requested}, \text{Reserved} \mapsto \text{Requested}\}$.

**Channel Model.** Given a set of message types $MT$ used in the GCM, the channel model $C$ is a total function from a sequence of messages (of types $MT$) to a sequence of messages (of types $MT$). With $MT' \subseteq MT$ and a message sequence $s$, $\pi_{MT'}(s)$ denotes the projection of $s$ to a sequence of messages of types $MT'$. Let $\pi_{MT'}(s)$ be canonically extended on the channel model. The channel model $C$ is then based on assignments of disjoint subsets $MT'$ of $MT$ to channel reliability guarantees which enforce that $\pi_{MT'}(C)$ satisfies certain properties. Reliability guarantees such as those from WS-RM standard [OAS07] can be modeled: exactly once in order (EOIO) where $\pi_{MT'}(C)$ is the identity function on interaction sequences and exactly once (EO) where $\pi_{MT'}(C)$ is a permutation on an interaction sequence.

**2.2.5 MCM Viewpoint Semantics**

It is not too complicated to formally define a semantics for the extended finite state machines $G$ and $L_i$ in conjunction with a channel model. This can for instance be done with the help of a translation to Event-B, as presented in Chapter 3. An important question in defining the overall semantics of GCM comes with the relation of global and local models.

It basically has to be fixed how send and receive events are glued together. Since these models have different alphabets we need to map the alphabet of interactions used by $G$ to their corresponding send and receive events in $L$, in order to define a consistency relation between them. How this is done is determined by the three possible viewpoints.
For simplicity, we fix the consistency relation between $G$ and $L$ to trace inclusion and assume $G$ to be an over-approximation of $L$, such that the traces of $L$ need to be included in $G$. Further let $MT(e)$ denote the message type of an interaction as well as of a send or a receive event $e$.

- A GCM $G$ is a send-viewpoint of a composed system $L$ if for each trace $(e_1, \ldots, e_k)$ of $L$ there exists a trace $(i_1, \ldots, i_k)$ of $G$ such that if $(s_1, \ldots, s_k)$ is the projection of $(e_1, \ldots, e_k)$ to the send events, then
  \[(MT(i_1), \ldots, MT(i_k)) = (MT(s_1), \ldots, MT(s_k))\]

**Example 3** If we project the traces of the composed system in Figure 2.2 to send-events we obtain the set (written as regular expression)

\[
\{(!\text{Request}!\text{Offer}!\text{Cancel})^*(!\text{Request}!\text{Offer}!\text{Order})^*\}
\]

From this we may construct a send-viewpoint simply by using the LPM of the buyer (and transforming send/receive events into interactions).

Though send-viewpoints give rise to quite simple models, they do not adequately cover the asynchronous nature of choreographies, i.e. assumptions on the channel are not reflected in the GCM: an EOIO channel would result in the same GCM as an EO channel. We therefore define the notion of a receive-viewpoint.

- A GCM $G$ is a receive-viewpoint of $L$ if for each trace $(e_1, \ldots, e_k)$ of $L$ there exists a trace $(i_1, \ldots, i_k)$ of $G$ such that if $(r_1, \ldots, r_k)$ is the projection of $(e_1, \ldots, e_k)$ to the receive events, then $(MT(i_1), \ldots, MT(i_k)) = (MT(r_1), \ldots, MT(r_k))$.

**Example 4** The GCM in Figure 2.2 is a receive-viewpoint of the composed system with an EO channel in that figure, because it covers the projection of the traces from the LPMs to receive events.

- In [WRS+09] we discussed an additional observe viewpoint which plays a role in monitoring choreographies. Note that if one considers EOIO for all messages, the three viewpoints are equivalent. However, already when there are two EOIO channels for messages in the same direction the equivalence does not hold anymore.

The receive-viewpoint exhibits exactly those orders of receive events that are possibly observed by the partners. This is an important aspect of integration verification and testing, is supposed to uncover also message racing problems. Thus, for this pilot we will assume a receive viewpoint semantics.
2.2.6 Pilot Instances

We have worked with around 6 pilot instances, i.e. concrete realistic message choreographies.

These models do not differ too much in size and complexity from our running example introduced in this chapter.

We have introduced variants of them already as mini-pilots in the M12 Deliverable D5 in Section 5.3.

![A typical message choreography model. Message type names are obfuscated.](image)

Figure 2.3: A typical message choreography model. Message type names are obfuscated.

For illustration, Figure 2.3 shows a typical (obfuscated) message choreography model which we used for our experiments. When the picture was taken, the model was in animation mode as shown in Section 7.2.

2.3 Opportunities for Formal Methods

The modeling language described above allows developers to

- reason very early in the development process about their design (thus avoiding cost intensive correction loops after implementation) and
• to validate their implementation with an appropriate degree of certainty.

Our goal with this pilot is to show that, with the help of formalizing the choreography model and an appropriate set of tools, it is possible to provide developers with additional means in these two steps which would increase efficiency and quality of development.

This is detailed in the remainder of this section.

2.3.1 Enforcing Model Consistency

From the definitions of MCMs one can deduce that message choreography models can be inconsistent or not fitting to the context or environment. This can be the case for the following reasons:

1. GCM and LPMs could syntactically not fit to each other. For instance there could be send events in the LPM for which there are no interactions in the GCM. Such simple consistency properties are in danger to be violated especially in an agile environment where both levels are modified simultaneously and global and local views easily get out of sync.

2. The composed system could not realize the behaviour of GCM, i.e. the composed system and the GCM are not receive-viewpoint trace equivalent. A special case is that for the GCM no composed system exists (at all) which is receive-viewpoint trace equivalent. The choreography is then said to be not locally enforceable.

3. The choreography could not be consistent with the modeled behaviour of individual, already pre-existing, partner components.

4. The modelled choreography could not fit to the informal business requirements, i.e. the model is wrong with respect to the intentions of solution architects or the model contains other unintentional errors.

While the last issue is clearly only resolvable by human, though tool supported, two competing approaches exist for ensuring consistency between local and global views: a generative approach where the LPMs are generated from the GCM, or a checking approach where global and local models are created separately, but having a consequent consistency check installed. While the first approach ensures that global and local views are always consistent, it makes changes to the local models considerably more difficult, since these would be overridden by re-generation from the global model. The latter approach allows for such “asymmetric” changes, but requires manual effort to update the global view when changes to the local models are made. We suggest a third course of action which
attempts to offer a pragmatic mix of these approaches. We address the first of the issues mentioned above by always maintaining the two LPMs as structural copies of the GCM while allowing for additional guards on the LPM. Technically the LPMs and the GCM are just views on a common model. Semantic consistency can however not be ensured by these simple means. We therefore apply model checking and theorem proving techniques by applying formal techniques as described in this deliverable. Moreover we are interested in verifying the MCM models towards local development models (Section 10.2) and validation (Section 7.2).

2.3.2 Goal of the Pilot

In this pilot, we would like to provide methods and tools which support the development process of service choreographies with the help of Event-B and Rodin. The validation and verification activities should be directed towards incrementally leading to a consistent model.

According to our experience, modeling starts with a first sketch of the communication behavior, possibly without incorporating any difficulties arising from messages being exchanged on channels, etc. By using an interactive simulation tool, discrepancies between the intuitive understanding and the modeled behavior should be uncovered and could then be corrected in the model. Sometimes also inconsistencies like inconsumable messages can already be detected.

Methods providing automated systematic checks should be available. If the model checker detects a flaw in the model, the modeler is pointed to the situation by providing a trace to the situation in which the error occurs. This would offer the modeler means to correct the error. Moreover consistency checks can be executed against the local business component models, which may uncover errors either in the latter or in the MCM model and may trigger corrections on both sides.

When the modeler is certain that the model is right with respect to the targeted receive viewpoint, final verification with a second verification tool (such as a theorem prover) could be conducted. This may uncover situations which cannot be detected by the model checker because the choreographies have an infinite search space.

Finally, integration test cases can be generated. When the implementation is finished these test cases can be concretized with test data and other information not contained in the model and can be run against the implemented system.
Chapter 3

Translating Choreography Models to Event-B

3.1 Summary

Event-B fits quite naturally to MCM: interactions can be seamlessly expressed as events and the relationship between GCM and LPMs can be formulated as Event-B refinement. Also other formalisms, such as UML [SBS09] have been successfully translated into Event-B, so that we were able to utilise past experiences and practices. Another distinguishing aspect is the tool support in form of the Eclipse-based Rodin tool [ABHV08a]. Due to the extensible architecture, various plugins for Rodin exist. The tool can be integrated with other Eclipse-based tools such as the MCM editor. The basic concept is have two Event-B machines: one for the global model and one for the composed system (i.e. local models and channel model). The latter machine refines the former one depending on the viewpoint chosen. For the global model, each interaction from the MCM model translates into an event in the Event-B model. For the local model, each interaction produces two events a send-event and a receive event. The states of each of the involved components is maintained as is the channel state.

3.2 A Translation from MCM to Event-B

We are interested in a formal representation of both, the GCM and the two local LPMs with a connecting channel model. Therefore the subsequently described translation generates two Event-B machines which use a common context: the Global Model describing the GCM and the Local Model, describing the composition (defined as in [But09]) of the two LPMs and the CM. Both machines describe the exchange of messages — the first in terms of observing a message, and the
latter in terms of sending and receiving messages. As messages with the same type and content may occur more than once, to each message a unique natural number is assigned, which is incremented when a new message is sent. Further to each message a type is assigned while it is possible to specify the content of the message as functions on the message. Because we aim at the use of a model checking technique the translation result is designed to be as deterministic as possible. We experimented with an assignment of types to messages which is nondeterministically initialized upfront; however this resulted in an indigestible state space for the model checker.

**Global Model.** For each transition in the GCM we generate exactly one event. For representing the states we define a global variable status with elements from a set type $s_1, \ldots, s_k$, with constants $s_1, \ldots, s_k$. It is initialized with $\text{init} \in S$. The basic translation of an Interaction $i \in I$ with $\langle \{s_1, \ldots, s_k\}, I, s_m \rangle \in \Rightarrow$ is as follows:

\[
\begin{align*}
\text{i} \\
\text{when} \\
\text{guard}1: \text{status} = s_1 \lor \ldots \lor \text{status} = s_k \\
\text{then} \\
\text{act}1: \text{status} := s_m \\
\text{end}
\end{align*}
\]

This basic translation must be augmented with preconditions and actions associated with that interaction. Therefore we have to represent data types, constants, variables, terms and formulae used in MCM in terms of Event-B. This is done as follows. For each data type $t \in T$ we define a set in the Event-B context without explicit characterisation of elements. These sets are named in Event-B according to their type name $\text{name}(t)$. For each complex data type $t = (f, t')$ we define a partial function $f : \text{name}(t) \rightarrow \text{name}(t')$. f is initialised with $f := \emptyset$. The constants and global variables are defined in a standard way. For each constant $c \in Ct$ an element is added to the set $\text{name}(t)$. For the interactions $I = i_1, \ldots, i_n$ we additionally define a set $\text{MESSAGES} = \{\text{name}(\text{itype}(i_1)), \ldots, \text{name}(\text{itype}(i_n))\}$.

**Example 5** Consider the interaction Request with

\[
\text{pre(}\text{Request}) = \text{msg.Header.ID} \not\in \text{ID\_SET}
\]

and $\text{act(}\text{Request})(\text{ID\_SET}) = \text{ID\_SET} \cup \{\text{msg.Header.ID}\}$ of our running example. For it, we define the functions

\[
\begin{align*}
\text{Header} : \mathbb{N} \rightarrow \text{MessageHeader} \\
\text{ID} : \text{MessageHeader\_InstanceID}
\end{align*}
\]
(MessageHeader and InstanceID here are the names from name(T)), and the local variables \( t_1 \) and \( t_2 \) in order to choose appropriate values to be assigned in the functions. Because \( ID\_SET \in \mathcal{TSet}(InstanceID) \) we define an Event-B variable \( ID\_SET \) of type \( \mathcal{P}(InstanceID) \).

Request

\[
\begin{align*}
\text{any } & t_1, t_2 \text{ where} \\
grd1 : & status = \text{Reserved} \lor status = \text{Start} \\
grd2 : & t_1 \in \text{MessageHeader} \\
grd3 : & t_2 \in \text{InstanceId} \\
grd4 : & t_3 \notin ID\_SET \\
grd5 : & t_1 \in \text{dom}(ID) \Rightarrow ID(t_1) = t_2
\end{align*}
\]

\[
\text{then}
\begin{align*}
act1 : & status := \text{Requested} \\
act2 : & \text{Header}(msg) := t_1 \\
act3 : & ID(t_1) := t_2 \\
act4 : & \text{type}(msg) := \text{Request} \\
act5 : & ID\_SET := ID\_SET \cup \{t_3\} \\
act6 : & msg := msg + 1
\end{align*}
\]

\[
\text{end}
\]

In the example, the guard \( grd5 \) describes a consistency property: if the function is already defined on an element, then the value must be the corresponding term.

For the target state \( e_i \subseteq S \) we define a special event terminate with a guard \( status = c_1 \lor \ldots \lor status = c_n \) (for all \( c_i \in e_i \)) and an action \( targetstate := \text{true} \), where \( targetstate \) is a global variable. In each event from the translation of GCM we additionally add an action \( targetstate := \text{false} \). As a result, \( targetstate \) equals true iff the system state is a target state.

**Local Model.** In the local model we generate events representing sending and receiving of messages. Depending on the viewpoint either the send or the receive event can be defined to be a refinement of the corresponding interaction in GCM. By definition of LPMs, the variables from \( V \) and the status variable are duplicated (one for each partner). The variable \( msg \) is translated as for the GCM in order to keep the unique message enumeration. It is only used by send events, where it is set in the same way as in the GCM. In receive events, local variables (parameters) are used in order to obtain some message from a channel. A channel is defined as a global variable of type \( \mathcal{P}(N) \) denoting the set of messages on the being exchanged. It is initialised with \( \emptyset \). Typically, we have two partners \( P_1 \) and \( P_2 \) and two sequencing contexts (EO and EOIO). In that case we obtain four possible channels in the model (two in each direction).
Example 6 Below we show a translation of the interaction Request from the LPMs for the partners buyer (B) and seller (S) of the example. The duplicated variables can be distinguished by the corresponding subscripts $B$ and $S$. The channel from buyer to seller having the sequencing EO is called channel$_{BS \_EO}$.

```
send_Request
any $t_1, t_2$ where
  grd1 : status$_B = Reserved \lor status$_B = Start$
  grd2 : $t_1 \in MessageHeader$
  grd3 : $t_2 \in InstanceID$
  grd4 : $t_3 \notin ID\_SET_B$
  grd5 : $t_1 \in dom(ID) \Rightarrow ID(t_1) = t_2$
then
  act1 : status$_B := Requested$
  act2 : Header(msg) := t_1
  act3 : ID(t_1) := t_2
  act2 : type(msg) := Request
  act5 : ID\_SET$_B := ID\_SET_B \cup \{t_3\}$
  act3 : channel$_{BS \_EO} := channel$_{BS \_EO} \cup \{msg\}$
  act4 : msg := msg + 1
end
```

```
receive_Request
any $m$ where
  grd1 : status$_S = Reserved \lor status$_S = Start$
  grd2 : $m \in channel$_{BS \_EO}$
  grd3 : type(m) = Request
  grd4 : $m \in dom(Header)$
  grd5 : $Header(m) \in dom(ID)$
  grd6 : $ID(Header(m)) \notin ID\_SET_S$
then
  act1 : status$_S := Requested$
  act2 : ID\_SET$_S := ID\_SET_S \cup \{ID(Header(m))\}$
  act3 : channel$_{BS \_EO} := channel$_{BS \_EO} \backslash \{m\}$
end
```

The translation of a send event is very similar to the translation of the corresponding event in GCM. In receive events all function values are already set so the purpose is to find a suiting message $m$ in the channel and “receive” it (delete
from the channel). If a sequencing context is EOIO then we need an additional guard that checks, that the message \( m \) has a smallest number in the channel. For inhibitor conditions \( \text{inhib}(i) = C \) (with \( i \in I \)) we add a guard \( \text{status} = C \) to the event \( \text{send}_i \).

In our example, we add the guard

\[
grd6 : \text{status}_B \neq \text{Reserved}
\]

to \( send\_Request \). It remains future work to optimize the translation by simplifying this and \( grd1 \) to \( \text{status}_B = \text{Initial} \).

Target states are treated similar to the translation of GCM except that we additionally demand \( \text{channel} = \emptyset \) for all of them. Only if all channels are empty the system can enter into a target state. For all other events of the translation from the LPM we add an action \( \text{targetstate} := \text{false} \).

### 3.3 Implementation of the Translation

The MCM Editor is based on the SAP NetWeaver Developer Studio (NWDS), which is an Eclipse IDE extended by SAP specific plug-ins. The MCM component is implemented via several Eclipse plug-ins (i.e. OSGi bundles) for the NWDS.

The OSGi based plug-in architecture of Eclipse makes the editor as well as the Rodin platform extensible and allows to integrate MCM, ProB and RODIN seamlessly into the SAP NWDS. Figure 3.1 shows an FMC\(^1\) Block Diagram of the embedding of the translation plugin within the Rodin and Choreography Editor plugin.

---

Figure 3.1: The embedding of the Event-B Generator plugin within the Rodin and MCM choreography model editor.
Chapter 4

Refinement between Global Choreography Models and Local Partner Models

4.1 Summary

This chapter describes how to check local enforceability (refinement) property for Message Choreography Models (MCM) using Event-B. We consider the definition of MCM and its translation to Event-B as defined in the previous two chapters.

It is possible (at this stage) that the GCM $G$ and the composed system $L$ (consisting of two local partner models and a channel model) are inconsistent in the sense of Chapter 2.

We have therefore integrated into the MCM editor ways to prove consistency, based on checking the obtained formal model with Rodin [ABHV08b] and Rodin based plugins.

To show consistency between global and local model boils down to showing the refinement of the Event-B machine generated for the combined system towards that of the GCM and vice versa.

Another important property is the absence of inconsumable messages (see Chapter 7): Whenever a message is being exchanged the receiver of the message must be ready to receive it. This property can be encoded as an invariant of the generated Event-B machine for the LPMs.

Since proving these properties still requires a considerable amount of user interaction we also rely on the model-checking tools provided by ProB [LB08] which will not be able to completely prove the refinement relation (because of the unlimited size of the channel) in general, but will give good feedback in cases where the model still contains errors.
The challenge to the verification between global and local model is to find appropriate gluing invariants which connect global and local model. The invariants are crucial in verifying MCM because the result of the translation from MCM are two Event-B models, one for the local model and one for the global model. Since we would like to show that the local model is a refinement of the global model, gluing invariants are needed which link the two models together.

Since our approach assumes that the formalism is hidden from the user, we have to provide means to also hide the gluing invariant authoring.

Our approach is as follows: we generate (heuristically) a number of gluing invariants between local and global models, which allow one to prove the refinement property. The invariants are constructed partially automatically from the translation of MCM to Event-B. Our experiments have shown that for the most of the considered message choreographies the refinement property could be successfully proved using only the proposed types of invariants.

Our approach relies on templates of gluing invariants which we typically find for MCM. A user of our tool may then select instances of these templates which she/he believes are applicable for the concrete MCM instance. This selection is considered much more feasible for the tool users than creating the invariants from scratch. It moreover exploits the domain-specific information reflected in MCMs. Finding the gluing invariant templates is a heuristic task. We have thus conducted a number of experiments with concrete realistic MCM examples from SAP and report on our experience with the approach.

4.2 Local Enforceability and Absence of Inconsumable Messages

Interesting properties for choreographies are the following:

(1) Local enforceability: any behavior that the local model permits is also possible to observe in the global model.

(2) Absence of inconsumable messages: the intended receiver of a message which is being exchanged is always ready to receive the message. This property is discussed in Chapter 7.

We have shown in [KRW09] that (2) implies (1). Moreover, since there are various ways to connect local and global models with each other [WRS+09], i.e. does a transition in the global model correspond to a send or to a receive event in the local model, we show that this holds regardless of this choice. In addition, this result is independent from whether messages can change order while being
transferred; in the case that the order is preserved and a send-viewpoint is assumed
the absence of inconsumable messages is even equivalent to local enforceability.

However this result only holds if there are no additional preconditions and
side-effects annotated and if the LPMs are structurally equal to the GCMs. In
the case of models for which these conditions are not satisfied we are obliged to
prove the local enforceability property by the refinement relation between GCM
and LPM.

To this end, the relation between GCM and LPM must be specified with the
help of gluing invariants as described in the following sections.

4.3 Gluing Invariants

In this section we describe seven types of gluing invariants that we propose to add
in the translation of MCM to Event-B in order to prove the refinement property
described above. We also present some considerations about rationale and usage
of invariants. Following notation is used below: $P_1$ and $P_2$ are two partners,
$S = \{s_1, s_2, ..., s_n\}$ is a global status variable, $P_1$-$S$ and $P_2$-$S$ are the copies of
the status variable for both partners, $m(e)$ is a message produced by the event $e$,
and channel is a generalized channel between partners.

We try to obtain a classification of gluing invariants used in order to prove the
local enforceability of choreographies. The obtained types of gluing invariants
between local and global models are found heuristically and allow one to prove
the refinement property in some cases. The list of types is not necessarily complete
and can be extended when needed. An extension is required if new types of gluing
invariants are needed for some proof obligations. Our experiments have shown
that for the typically considered choreographies the refinement property could be
successfully proved using only the proposed types of invariants.

4.3.1 Types of Gluing Invariants

The following types of gluing invariants are currently considered:

Type 1 (Local state implies global state):

$I_1(s_j, P_i) \equiv (P_i$-$S = s_j) \Rightarrow (S = s_{k_1} \lor S = s_{k_2} \lor ... \lor S = s_{k_n})$.

The meaning of the invariants of this type is that given a state of a single
partner the possible global states are represented.

Type 2 (Message in the channel implies state): An event $e$ sent by the partner $P_i$
is called a non-cyclic event if it does not produce, together with other events sent
by $P_i$, a cycle. Consider an event $e$ such that
(1) The event $e$ is a non-cyclic event.

(2) There exists a non-cyclic $e'$ of another partner $P_j$, such that $s_t \rightarrow_e s_m$ and $s_m \rightarrow_{e'} s_n$.

The second type of invariants is:

$I_2(e) \equiv (m(e) \in \text{channel}) \Rightarrow ((S = s_{k_1} \lor S = s_{k_2} \lor \ldots \lor S = s_{k_n}) \land (P_1 S \neq s_1)).$

Usually we use the form $I_2(e)$, but there exist variants, which can also be useful for proving local enforceability.

**Type 3 (Bounded Event):** The following invariant holds if the event $e$ is bounded. In a more complex case the event can be bounded by a constant $k \neq 1$ and one can change the invariant $I_3$ correspondingly.

$I_3(e) \equiv \text{channel}(m(e)) \leq 1.$

**Type 4 and Type 5 (Initial Value without Inputs):** If an initial state has no input (i.e., there exists no event with an action $S := \text{initial}$), then the following types of invariants can be useful:

$I_4(P_i) \equiv (P_i S = \text{initial}) \Rightarrow (S = s_{k_1} \lor S = s_{k_2} \lor \ldots \lor S = s_{k_n}) \land$

$(m(e_1) \notin \text{channel} \land m(e_2) \notin \text{channel} \land \ldots \land m(e_k) \notin \text{channel}),$

for all non-cyclic events $\{e_1, e_2, \ldots, e_k\}$ of the partner $P_i$.

$I_5(P_i) \equiv (S = \text{initial}) \Rightarrow (P_i S = \text{initial}) \land$

$(m(e_1) \notin \text{channel} \land m(e_2) \notin \text{channel} \land \ldots \land m(e_k) \notin \text{channel}),$

for all non-cyclic events $\{e_1, e_2, \ldots, e_k\}$ of the partner $P_i$ with initial value in the guards.

**Type 6 (Concurrent Events):** If two events ($e_1$ and $e_2$) of the same partner have the same status value in guards and both events are non-cyclic (with respect to the partner), then the following type of invariants can be needed:

$I_6(e_1, e_2) \equiv m(e_1) \notin \text{channel} \lor m(e_2) \notin \text{channel}$

**Type 7 (Variable modified by Partner):** If a status variable is changed (in send or receive events, corresponding to the point of view) only by the partner $P_i$, then we can write

$I_7(S, P_i) \equiv (S = P_i S)$. 
4.3.2 Some Rationale of the Types of Gluing Invariants

In this section we give some rationale for the first three types of invariants. For each event $e$ and the corresponding send or receive (dependent on used send or receive point of view) event $e'$ in order to show the refinement relation we obtain the following proof obligation:

$$\text{Guard}(e') \Rightarrow \text{Guard}(e) \quad (GRD\ PO).$$

For the rest of the section we fix a send point of view.

**Type 1**: Let the partner $P_i$ send $e$ with $s_1 \rightarrow_e s_2$. In order to prove the GRD PO we consider the invariant $I_1(s_1, P_i)$, which is usually enough.

**Type 2**: Let us consider a receive-event $e_r$ corresponding to the acyclic event $e$ with $s_1 \rightarrow_e s_2$ and an invariant of a first type $I_1(e', P_i)$ corresponding to the acyclic event $e'$ with $s_2 \rightarrow_{e'} s_3$. Before receive-event $e_r$ the invariant holds because the event $e$ is acyclic. After the receive event we have to prove

$$(s_2 = s_2) \Rightarrow (S = s_{k_1} \lor S = s_{k_2} \lor \ldots \lor S = s_{k_n}).$$

This can be proved only if we have additional information about global states of status during the receive-event $e_r$. Since global states are changed by send events, we need a connection between send and receive events.

The following invariants can be formulated in this case:

$$I_2^0(e) \equiv m(e) \in \text{channel} \Rightarrow (S = s_{k_1} \lor S = s_{k_2} \lor \ldots \lor S = s_{k_n}).$$

Now this invariant cannot be proved by the acyclic send-event $e'_s$ with $s_2 \rightarrow_{e'} s_3$. We need to show that if a message $m(e')$ is sent, then the message $m(e)$ is already received, i.e.:

$$I_2^{add}(e) \equiv m(e') \in \text{channel} \Rightarrow m(e) \notin \text{channel}.$$ 

This can be done by extending $I_2^0(e)$ to $I_2(e)$ or by using $I_2^{add}(e)$. The invariant $I_2(e)$ can be usually proved by contradiction using the first types of invariants.

**Type 3**: The invariant of the second type $I_2(e)$ cannot in general be proved in the case of receive-event $e_r$. We have to prove that after deleting a message from the channel the invariant for an event $e$ still holds. In general the right side of the implication in $I_2(e)$ is false and the implication holds only if $m(e) \notin \text{channel}$, i.e., we have to prove that after one deletes from the channel a message corresponding to the event $e$, there will be no more such messages in the channel. To prove this...
we should demand additionally that \( \text{channel}(m(e)) \leq 1 \). The invariant holds if the event \( e \) is bounded. In a more complex case the event can be bounded by a constant \( k \neq 1 \) and one changes the invariant \( I_3(e) \) correspondingly.

4.3.3 Automatic Generation of Gluing Invariants

In this section we consider the construction of the invariants on the example of invariants of first type. We show that in the case of the send point of view the construction is automatic and in the case of the receive point of view some user interaction is needed. This is also the case with the second type of invariants. All other types of invariants can be trivially constructed. In the case of the send point of view the following algorithm can be used for the calculation of the invariant \( I_1(s_j, P_i) \):

1. Start with \( X = s_j \).
2. Repeat until \( X \) is fixed:
   - Add all \( s' \) to \( X \) such, that \( s \rightarrow e' s' \), where \( e' \) is an event sent by another partner \( P_j \) and \( s \in X \).
3. Write \( I_1(s_j, P_i) \) for all \( s_j \in X \).

The intuition here is that in the case of the send point of view the global state is either \( s_j \) or another partner \( P_j \) has sent some messages starting from \( s_j \).

In the case of receive point of view calculating the possible global states from the partner’s state \( s_j \) one should consider the situation, where another partner is still not in the state \( s_j \). This corresponds to the situation where some messages were not received. Compared to the send point of view, where a forward calculation starting from \( s_j \) is needed, in the case of receive point of view one should calculate backward starting from \( s_j \). But in this case also the possible channel content should be considered. This makes the later calculation to a hard (if at all decidable) problem. Therefore in the case of receive point of view we give user a possibility to correct the proposed set \( \{ s_{k_1}, ..., s_{k_n} \} \) manually.

4.4 Integration in Diagrammatic Front-End

Because the types of invariants and their usage are obtained in the heuristic way, we are not sure that the proposed set of invariants is optimal in each concrete case. In our approach we give users the possibility to correct the set of proposed invariants.
An invariant is in general connected (logically) with the following modelling elements: model itself, status values, status variables (note that in general there is more than one of them), interactions, local partners. Each modelling element has also its graphical representation (graphical element) in the tool. In Fig. 4.1 we show how the graphical user interface for working with invariant proposal looks like: for the selected interaction Cancel an automatic proposal is made (ticked types). For example in the case of the second type of invariants the automatic proposal is: Model - always, Variable - always, Value - always, Partner - always, Interaction - only if the interaction is acyclic and has an interaction sent by another partner as a successor.

![GUI for participation of gluing invariants](image)

Figure 4.1: GUI for participation of gluing invariants

As we have mentioned already, in the case of the receive viewpoint not all types of invariants can be constructed automatically. It was shown in the previous section, that the status values in invariants of the first and the second types cannot
be calculated automatically in general. In this case some graphical interface similar to the one in Fig. 4.1 can be provided to the user in order to fulfill (correct) the automatically generated versions of invariants. The tool still makes an initial proposal.

The invariants are represented in terms of a textual description, which we expect users of the tool to become easily familiar with. Of course a graphical representation of the invariants might be desirable, but the realizability of this remains to be investigated.

4.5 Verification Procedure

In this section we describe the whole verification procedure for MCM using the types of invariants proposed in this paper. Given an MCM model the purpose is to check if the model is local enforceable (by fixed send or receive point of view).

In Fig. 4.2 the whole verification procedure is schematically depicted. First of all an MCM is translated to Event-B model as described above (both global and local parts). After this the automatic invariant proposal is generated. The proposal can be changed each time using the graphical user interface (GUI) as described above. Having an invariant proposal the Event-B model can be extended with automatically generated invariants.

In the case of receive point of view, where we have no guarantee, that the constructed invariants are correct, the GUI can help one to edit the sets of status values needed for invariant generation. Having an Event-B model with generated invariants, one can start with the proving of the generated prove obligations. In
Rodin this can be done in both automatic and manual modes. If during the proofs the necessity of new invariants (or corrected versions of the existing invariants) appears, then the modeler can (through the GUI) help either by changing the invariant proposal or by correcting content of the invariants.

4.6 Experience

In the framework of the verification procedure described above we have tried to verify a number of typical and realistically sized choreographies. The following statistics was obtained with Rodin 0.9 and is based on seven considered case studies.

We have used the send point of view allowing automatic generation of proposed invariants and all considered examples could be successfully verified by using only the automatically proposed types of invariants. The numbers of generated proof obligations were between 300 and 600 depending on the example. About 70% of them could be proved automatically (with repeated runs of the auto-provers). Up to 80% of the remaining proof obligations could be successfully solved by manually calling the ML, P0 and P1 provers of Rodin. The fact that these proofs have not been obtained during the runs of auto-provers can be partially explained by the small timeout number set in the auto-provers (which currently cannot be changed from the GUI). The remaining proof obligations were solved manually (about 6% from the whole number of proof obligations).
Chapter 5

Middleware Models in Event-B

We report our experience in using Event-B to enhance processes and tools for development of business information software based on service-oriented architectures. We focus on the configuration of middleware, verifying application-level requirements in the presence of faults. In instances of the choreography pilot we used the Event-B formalism and the open Rodin tools platform to prove properties of models of business protocols and expose weaknesses of certain middleware configurations with respect to particular protocols.

The work summarized here is published in [BFRR09a], in which we report our experience in using formal methods to enhance processes and tools for development of business information software based on service-oriented architectures. In our work, using the pilot studies provided by SAP, we focus on the configuration of middleware, verifying application-level requirements in the presence of faults. We prove properties of models of business protocols and expose weaknesses of certain middleware configurations with respect to particular protocols. We then extended the approach to use models automatically generated from diagrammatic design tools, opening the possibility of seamless integration with current development environments. Increased automation in the verification process, through domain-specific models and theories, is a goal for future work.

The challenge we address is that of gaining the benefits of formal modelling and analysis in the development of SOA-based business information systems, while retaining a beneficial trade-off between the effort invested by developers insights that they gain. Our approach aims for a smooth transition between the traditional development processes and formal approaches. Initially, developers might not directly interact with a formal modelling tool, but continue to use pre-existing diagrammatic domain-specific modelling environments. The models developed in these environments could be automatically translated into a suitable formal notation and treated with automated analysis tools. While the insights gained from such purely automatic analysis might be less than those arising from
a more thorough adoption, we expect that the formal methods would be seen by developers as a benefit demanding little additional effort. In the long run, we expect that subjectively experienced and objectively measurable benefits will lead to a positive attitude towards the methods and tools and then to their more extensive direct use.

The purpose of our study is to determine the technical feasibility of using Event-B/RODIN to support the analysis of design models of SOA-based business information applications. The specific focus is on selecting an appropriate configuration for middleware from among alternatives offering different levels of fault tolerance (EO or EOIO). The applications models should be derived automatically from existing graphical design tools and there should be a good level of automation in the analysis of the models.

Our approach is to use a series of case studies, with the aim of producing a proof-of-concept of the automated analysis discussed above. In our studies, we needed to:

1. Establish that Event-B/RODIN can indeed support the comparison of alternative middleware components with respect to application-level properties.
2. Define the process for interfacing alternative middleware models (e.g. EO or EOIO) to pre-existing application-level models.
3. Develop appropriate strategies for combining middleware models with application models derived from the pre-existing graphical design tools so as to yield a good degree of automation in the analysis.

These are the subjects of the three studies described below. The studies used both the B2B and A2A pilot studies.

In the business-to-business (B2B) choreography (or protocol) [WRSC08] two components, a buyer and seller, exchange messages in order to negotiate the price of a product or service. The negotiation is initiated by a proposal from the buyer detailing purchase conditions such as price, quantity, or delivery date. The two parties may then arbitrarily exchange further proposals. A party indicates agreement to a proposal by returning that proposal. The negotiation may be cancelled at any time. The critical property that B2B is designed to establish is:

**Property 1** When a run of the protocol terminates, either the buyer and seller should have agreed to the same price, or they should agree that the negotiation has been cancelled.

In the application-to-application (A2A) choreography two components interact to meet a requirement from a customer. The *ordering component* is responsible
for managing customer requirements, and the supply chain requirements component coordinates the services used to process these requirements. The protocol starts when the supply chain component receives customer requirements from the ordering component. The supply chain component may then send notification of (partial) fulfilment of these requirements (e.g. delivery) back to the ordering component. The ordering component may also send queries and preliminary reservation requests and the supply chain component sends current supply planning and delivery information to the ordering component.

5.1 Study 1: Middleware Models

The aim of the initial study, using the B2B protocol, was to confirm that the Event-B/RODIN tools could support the comparison of alternative middleware components with respect to application-level properties. The application is built from a buyer and a seller component, and either EO or EOIO middleware. Our method was first to build Event-B models of EO and EOIO middleware, and an abstract model of the B2B protocol that did not contain an explicit component representing middleware. Each of the middleware models was composed in turn with the B2B model, and the Event-B/RODIN tools were used to compare the two combinations.

Each application-level event involves both a protocol party (buyer or seller) and the middleware. It is therefore partially described in each model. The protocol model describes the effects of the action local to the buyer or seller and the middleware model describes the effects of the action local to the middleware. Composing the middleware with the protocol involves composing each of these actions.

For example, the application-level action of the buyer sending a proposal \( p \) to the seller appears in the protocol model as \texttt{Buyer\_send} (see Figure 5.1).

Figure 5.2 shows the EO model invariants. These describe the structure of the middleware.

In the case where the proposal is not yet in the middleware, the local effects of the buyer sending a proposal are defined as \texttt{Buyer\_send\_mw}, shown in Figure 5.3. A separate event describes the case where the proposal being sent is already in the middleware.

Combining two events into one retains shared parameters (in this case \( p \)), and conjoins guards and actions. In this way, each event in the protocol model is combined with the appropriate event from the middleware model to create a model of the application. This process is automated by a composition plugin available for the Event-B tool [Sil].

When using the EO middleware, Property 1 is formulated as in Figure 5.4.
Buyer\_send
\begin{verbatim}
  any p where
  p ∈ PROPOSAL
  p ≠ {empty, cancel}
  p ≠ last_s_o_rec
  BAgreeStatus = NoAgreement
  BCancelStatus = NotCancelled
  then
  curr_b_o := p
end
\end{verbatim}

Figure 5.1: Buyer\_send in the protocol

invariants:
\begin{align*}
  \text{inv1} & : mware\_to\_seller ∈ PROPOSAL ↦ N_1 \\
  \text{inv2} & : mware\_to\_buyer ∈ PROPOSAL ↦ N_1
\end{align*}

Figure 5.2: EO invariants

When the B2B protocol runs on EO middleware, Property 1 cannot be proved.

5.2 Study 2

In this study our aim was to further investigate how to develop models of business applications which allow for the introduction of middleware representations from a range of components and to develop standards for the integration of middleware into application models.

In this study, we integrated the middleware models with an independently developed model of the B2B protocol. This allowed us to identify more clearly the interface between the middleware and the protocol parties, and to develop a set of guidelines for protocols developers wishing to use the middleware.

The identified guidelines include:

- **protocol parties** should be developed in one machine, with no representation of middleware. Each send or receive event should instead use a reserved variable name as a parameter (e.g. “p” in Section 5.1).

- **correctness criteria** for the protocol should be expressed as application invariants, (although they will not in general be provable before a middleware
Buyer_send_mw

any p where
   p \notin \text{dom}(\text{mware\_to\_seller})
then
   \text{mware\_to\_seller}(p) := 1
end

Figure 5.3: Buyer_send in EO middleware

invariants:

\text{inv20 (BuyerAgStatus} = \text{Agreement} \land
\text{SellerAgStatus} = \text{Agreement} \land
\text{mware\_to\_buyer} = \emptyset \land
\text{mware\_to\_seller} = \emptyset)
\Rightarrow \text{curr\_b\_o} = \text{curr\_s\_o}

Figure 5.4: Property 1 in EO

model is integrated).

- complex message sets should be defined in a new context visible to both the protocol model and the middleware model.

5.3 Study 3

In the final study, we moved closer to our goal of using Event-B models that had been automatically generated from existing graphical modelling tools used in SAP, rather than hand-crafted. As might be expected, a major challenge here is in achieving a good degree of proof automation with auto-generated models, so it is necessary to consider alternative ways of introducing the middleware into the refinement chain in order to gain insights into their benefits and disadvantages.

We used two different techniques to produce a new machine containing EOIO middleware. The first technique produced the new machine by refining a machine in the original model containing the abstract choreography, and the second by refining a machine containing the low-level behaviour of the protocol.

An Event-B model of the A2A protocol was automatically generated from existing diagrammatic domain-specific modelling languages, rather than hand-crafted. It contained two machines, \text{m\_choreography} and \text{EO\_A2A}. The first
machine, m_choreography, had a high-level view of behaviour and no explicit component representing the middleware. It contained seven events and two invariants, and produced no proof obligations. The second machine, EO_A2A, contained the local behaviour of the ordering and supplier components and a model of EO middleware. In EO_A2A each of the events from m_choreography was refined by a “send” and a “receive” event, giving 14 events. It had 22 invariants and 268 proof obligations of which 263 were proved automatically and 5 required (trivial) intervention.

To investigate the modelling options, two machines containing EOIO middleware were developed by hand. EOIO_A2A_ONE was a refinement of m_choreography and EOIO_A2A_TWO was a refinement of EO_A2A.

There were 257 proof obligations in EOIO_A2A_ONE, 162 of which were proved automatically. The remaining 95 were significantly more complex than the invariants in EO_A2A. The primary source of the increased complexity was ten invariants that relate messages in middleware to states within the machine. In EO_A2A the process components exchange messages from the set MESSAGES. The EO middleware does not offer an ordering guarantee, so the representation is an unordered bag \textit{channel} of type MESSAGES $\rightarrow \mathbb{N}$. The quantity of a message $M$ in the middleware is given as \textit{channel}(M).

EOIO_A2A_TWO contains 186 proof obligations, of which 116 were proved automatically. The remainder required about 6 hours of effort. The bulk of this effort went in proving the linking invariants between the two middleware representations.

The benefit of the first approach is that there is no need for the local machine EO_A2A. It represents the case where a machine containing a representation of EO middleware is not available, or the developer is interested exclusively in EOIO middleware. The (overwhelming) disadvantage is that the level of manual intervention required to prove the proof obligations is too high. Conversely, the second approach requires an intermediate machine (EO_A2A) to be built and proved. It represents the case where a machine containing a representation of EO middleware is available to be used.
Chapter 6

Application of Patterns

The generative approach reported in the previous chapters deals with the original choreography models on a very fine-grained basis: each MCM modeling entity (status value, interaction,...) is translated into a specific part of an Event-B machine. While this approach works in principle the degree of automatic proving can still be improved. This is often due to the fact that different levels of abstraction (which could be represented as refinement levels in Event-B) are not reflected appropriately in the models.

On the other hand, patterns in Event-B [HFA09, Für09] are a means to express more coarse-grained units of design. In this section we report on our attempts to ease the formal development of choreography models with the help of the Event-B pattern mechanism. We do this in two steps: First we investigate the applicability to the choreography models in general. Then we take into account that Event-B models are systematically generated from a diagrammatic front-end modelling language.

6.1 Application to Selected Choreography Models

This chapter outlines the applicability of the design pattern approach to the business protocols provided by SAP. For more information on the theoretical aspects of design patterns in Event-B we refer to [Für09]. A short summary of the idea of design patterns as well as instructions on tool usage are given on the wiki page\(^1\). You may also have a look at the Deploy Deliverable D23 of WP9.

At ETH Zurich, we have extensively studied the pilot instance called Order/Supply Chain A2A Communication provided by SAP. A description of it is contained in Deploy Deliverable D5 in Section 5.3.3. We will refer to it in the

\(^1\)http://wiki.event-b.org/index.php/Pattern
following as A2A protocol. In order to see the possible usage of patterns within the A2A protocol we give an overview of the protocol behaviour first.

In Figure 6.1 you see a state diagram of what we extracted out of the specifications given in form of a requirements document.

The A2A protocol describes the communication between two business components within the ordering process of certain goods. The ordering component (OC) collects orders from buying organizations and sends these in form of requirements to the supply chain component (SCC). In return, SCC provides OC with information on the availability of the required goods.

The protocol goes on until either the required goods are available or OC aborts.
the protocol by deleting the before stated requirement. The protocol can be seen as a two phase protocol in which phase 1 can take forever and phase 2 finally terminates. In Event-B one would say phase 2 converges whereas phase 1 does not.

6.1.1 Phase 1

Phase 1 is kind of a negotiation phase, where \( OC \) asks for the availability of certain goods. In return \( SCC \) informs \( OC \) about the availability of this goods and reserves them provisionally. Although provisionally reserved, \( OC \) can still change the requirements by sending a new query, or simply abort the protocol when not interested in this goods anymore. This negotiation is stated in the upper part of the state machine in Figure 6.1.

The states are a combination of two separate status: RequirementStatus and CancellationStatus. The RequirementStatus represents the flow of the ordering process whereas the CancellationStatus simply states whether the protocol is still running or not. Every transition between the states is linked with a message that is transferred. The A2A protocol starts in phase1 in the state ReqInit (left-most state in Figure 6.1).

When \( OC \) transfers an AvailabilityCheckRequest (ACR), RequirementStatus changes to ReqQueried. Now it is at \( SCC \) to transfer an AvailabilityConfirmation (AC) to \( OC \). When the goods are available RequirementStatus will change to ReqProv stating that the goods are provisionally reserved. The duration until the goods are available is smaller than infinity (\( d < \infty \)). Otherwise, if the goods are not available at all (\( d = \infty \)), RequirementStatus is set back to ReqInit.

6.1.2 Phase 2

After negotiation and being either in ReqInit or ReqProv, \( OC \) can decide to trigger the reservation by transferring a RequirementFulfillmentRequest (RFR), as can be seen in the lower part of Figure 6.1. The RequirementStatus is now reserved (ReqResv) and \( OC \) has to wait until \( SCC \) has the goods available.

Until fulfillment, \( SCC \) has the option to transfer FulfillmentUpdate messages. This updates are for information only. They, for example, inform \( OC \) when the goods are available earlier than promised.

As soon as the goods are available, \( SCC \) will transfer the FulfillmentConfirmation (FC) and the protocol ends in the accepted state ReqFulfilled.

During phase 2, \( OC \) has the possibility to abort the protocol by transferring a RequirementDeletion (RD) that leads the protocol to the state CANCELLED.
this state the protocol does not take the RequirementStatus into account as it does not matter.

Note that when the protocol is in the state ReqFulfilled/NOT_CANCELED its state can change to CANCELED. This happens if OC sends a Requirement-Deletion message unaware of the already sent but not yet arrived FulfillmentConfirmation.

The same story happens in the other direction. OC aborts the protocol and the state changes to CANCELED. Before SCC gets aware of the transferred RD it transfers FC as a direct consequence of the just available goods. Unlike before the state will not change, it remains in CANCELED.

In phase 2, the protocol will finally terminate except for one single case that will be discussed below. Termination in terms of state machine means that no loop exists. In the lower part of Figure 6.1, representing phase 2, one can see that almost all transitions are pointing towards the accepted states CANCELED and ReqFulfilled/NOT_CANCELED.

The only loop is the transfer of FulfillmentUpdate (FU). But this transition can only occur within a certain time, namely until the reserved goods are available. This time was limited during the negotiation phase by the promised duration until the goods are available. Therefore the state machine cannot be caught within this loop forever.

The only exception that can violate the termination property is the case where OC directly reserves the goods without a negotiation in advance. If the goods are not available at all, the protocol may stick in ReqResv (Reserved Status) until OC decides to abort the protocol.

### 6.1.3 Discovering Patterns

The A2A protocol involves by its nature a lot of communication. There are three different forms of message transfer in this example.

**Synchronous Two-way Communication**

In phase 1, during negotiation, the two involved partners OC and SCC communicate in a fair dialogue. OC asks for a good and waits for an answer of SCC. OC cannot ask another question before it has received SCC’s answer to the former one. This is a synchronous communication behaviour. Every question is treated as a new message regardless of its content. Therefore every message is sent only once. This is covered by the pattern called Question/Response. See Section 6.1.6 for more details on this pattern.
Single One-way Communication

All message transfers in phase 2 are one-way. Most of the messages are commands and therefore an answer is neither expected nor desired. This applies for the transfer of RequirementDeletion messages as well as RequirementFulfillmentRequest messages. All this messages can only appear once for which reason the SingleCom pattern (see Section 6.1.4) perfectly fits. The FulfillmentConfirmation message is more like an information than a command. Yet there is no need for a response and the message can only be sent once. Therefore the SingleCom pattern works also here.

Multiple One-way Communication

Like the FulfillmentConfirmation message, the FulfillmentUpdate message is just an information that needs no feedback. But the FulfillmentUpdate message can be sent more than once for which reason the SingleCom pattern is not applicable. We use the MultiCom pattern, described in Section 6.1.5, instead.

6.1.4 Single-Message Communication

The description of the protocol is as follows.

(1) There are two parties: Sender and Receiver

(2) The Sender can transfer exactly one single message to the Receiver.

(3) The transfer of the message is represented by a boolean.

\[
\text{trans.} = \begin{cases} 
F & \text{if message transferred} \\
T & \text{if message not transferred} 
\end{cases}
\]

Figure 6.2: Single-Message Communication

6.1.5 Multiple-Message Communication

The description of the protocol is as follows.

(1) There are two parties: Sender and Receiver

(2) The Sender can send many messages (multiple message) to the Receiver.
(3) The messages are different, in other words, there is no resend.

(4) To distinguish the freshness of the message, each message is stamped with a sequence number.

(5) The Receiver can only receive new messages.

(6) The Receiver can discard any message.

Figure 6.3: Synchronous Multiple-Message Communication

6.1.6 Question/Response

The description of the protocol is as follows.

(1) There are two parties: Questioner and Responder

(2) The protocol consists of an unbounded number of rounds.

(3) In each round, the Questioner transfers a message (question) to the Responder, and in return, the Responder transfers a message (response) to the Questioner.

(4) The question message and the corresponding response message have the same number.

The communication of the Questioner and Responder is synchronous. Therefore the Questioner cannot transfer a new question before the Responder has transferred a response and vice versa. The pattern was developed by using two synchronous multiple-message patterns.

6.1.7 Refinement Strategy

The A2A protocol was developed as follows. Of special interest (and therefore highlighted) are of course those refinement models where patterns were used.

context ctx_0
Figure 6.4: Question/Response protocol with two rounds

- the types of the status `RequirementStatus` and `CancellationStatus` are defined as sets containing the possible values

**context ctx_1**

- `infinity` is defined as a certain natural number

**context ctx_2**

- `g` and `f` are defined, two functions mapping the status to natural numbers (this is needed to specify the variant in phase 2)

**machine a2a_00**

- variables `RequirementStatus` and `CancellationStatus` are defined
- events `progress_phase1` and `progress_phase2`, and their possible status changes are defined

**machine a2a_01**
• phase 1 is modelled as there is a question and a response event changing the status

• progress_phase1 is refined and therefore disappears, since all possible transitions in phase 1 are stated

machine a2a_02

• message exchange is modelled with transferred variables and connected to event question and response

• the exchange of the messages and therefore event question and response are defined to be synchronous

machine a2a_03

• question/response pattern is applied for question and response

machine a2a_04

• guard refinement (instead of checking the status, the message numbers are compared)

• event send_AC and receive_ACR are split into two events each (separated status ReqInit and ReqProv)

machine a2a_05

• variable duration is introduced to distinguish whether the status changes to ReqInit or ReqProv when receiving an AC

machine a2a_06

• the RD message transfer is modelled (three different events for ReqProv, ReqResv and ReqFulfilled)

• progress_phase2 adapted

machine a2a_07

• 3 single communication patterns are applied for transfer of RD
- receive_RD events are collected to one

**machine a2a_08**
- CancellationStatus refined and therefore disappears (the guards concerning CancellationStatus are replaced by OC_sent_RD)
- progress_phase2 adapted and split into two separate events (progress_phase2_fromOC and progress_phase2_fromSCC)

**machine a2a_09**
- the RFR message transfer is modelled (two different events for ReqInit and ReqProv)
- progress events adapted

**machine a2a_10**
- 2 single communication patterns are applied for transfer of RFR
- receive_RFR events are collected to one
- progress_phase2_fromOC is refined and therefore disappears

**machine a2a_11**
- the FC message transfer is modelled
- progress_phase2_fromSCC is refined and therefore disappears

**machine a2a_12**
- a single communication pattern is applied for transfer of FC

**machine a2a_13**
- RequirementStatus refined and therefore disappears (guards concerning RequirementStatus are replaced by local variables)

**machine a2a_14**
• OC_send_RD_fromResv and OC_send_RD_fromFulfilled are merged together, in order to remove the guard about \textit{SCC\_sent\_FC}, which is not in the view of \textit{OC}

• guard added to the merged event \textit{OC\_send\_RD} following the requirements (\textit{OC\_received\_FC} = \text{FALSE})

machine a2a_15

• the \textit{FU} message transfer is modelled

machine a2a_16

• a multiple-message communication pattern is applied for transfer of \textit{FU}

machine a2a_17

• channel introduced to send the remaining duration when \textit{FU} is transferred

• SCC\_send\_FU is adapted to also feed the new duration channel

6.1.8 Proof Statistics

Table 6.1 on the facing page lists the number of proofs for each model of the A2A development without using patterns. The number of proofs that can be saved using patterns are highlighted.

The automated proofs were discharged within Rodin 1.1 using the default settings for auto-tactic and post-tactic.

Using patterns in the development of the A2A protocol results in saving 97 proofs, which is 34.5\% of all the proofs. About the same ratio (35.5\%) is saved when only concerning manual proofs.

6.2 Integration with Diagrammatic Front-ends

In this section, we investigate the connection between the choreography models (as mentioned in Chapter 2) and the application of patterns: The input for pattern application is generated from matching of choreography models and the resulting
<table>
<thead>
<tr>
<th>model number</th>
<th>number of proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
</tr>
<tr>
<td>contexts</td>
<td>0</td>
</tr>
<tr>
<td>ctx_0</td>
<td>0</td>
</tr>
<tr>
<td>ctx_1</td>
<td>0</td>
</tr>
<tr>
<td>ctx_2</td>
<td>4</td>
</tr>
<tr>
<td>machines</td>
<td></td>
</tr>
<tr>
<td>a2a_00</td>
<td>3</td>
</tr>
<tr>
<td>a2a_01</td>
<td>5</td>
</tr>
<tr>
<td>a2a_02</td>
<td>5</td>
</tr>
<tr>
<td><strong>a2a_03</strong></td>
<td><strong>34</strong></td>
</tr>
<tr>
<td>a2a_04</td>
<td>12</td>
</tr>
<tr>
<td>a2a_05</td>
<td>9</td>
</tr>
<tr>
<td>a2a_06</td>
<td>20</td>
</tr>
<tr>
<td><strong>a2a_07</strong></td>
<td><strong>19</strong></td>
</tr>
<tr>
<td>a2a_08</td>
<td>21</td>
</tr>
<tr>
<td>a2a_09</td>
<td>23</td>
</tr>
<tr>
<td><strong>a2a_10</strong></td>
<td><strong>16</strong></td>
</tr>
<tr>
<td>a2a_11</td>
<td>17</td>
</tr>
<tr>
<td><strong>a2a_12</strong></td>
<td><strong>9</strong></td>
</tr>
<tr>
<td>a2a_13</td>
<td>38</td>
</tr>
<tr>
<td>a2a_14</td>
<td>1</td>
</tr>
<tr>
<td>a2a_15</td>
<td>6</td>
</tr>
<tr>
<td><strong>a2a_16</strong></td>
<td><strong>19</strong></td>
</tr>
<tr>
<td>a2a_17</td>
<td>20</td>
</tr>
<tr>
<td>total</td>
<td><strong>281</strong></td>
</tr>
<tr>
<td>savings</td>
<td><strong>97</strong></td>
</tr>
<tr>
<td></td>
<td>(34.5%)</td>
</tr>
</tbody>
</table>

Note: The highlighted machines are “generated” by the pattern approach, hence are correct by construction. Therefore we can consider the proofs associated with these machines are “saved”.

Table 6.1: Number of proofs in each refinement level.
Event-B model is generated accordingly to the technique mentioned in the previous section. We illustrate the idea through an example of a business communication protocol. We present first the description of the protocol in Section 6.2.1 with the corresponding choreography model. We present the patterns that are going to be used for the protocol. Finally, we present the development of the protocol using these patterns with the information collected from the graphical choreography model.

### 6.2.1 Description

There are two partners participating in the protocol: a *Buyer* and a *Seller*. They communicate asynchronously through different channels.

- The *Buyer* starts the protocol by sending a *request*.
- On receiving the *request*, the *Seller* can send a *confirmation*.
- However, after sending the *request*, the *Buyer* can send a *cancellation* message.

The global status of the protocol can be enumerated as follows: *INIT*, *REQ*, *CONF* and *CANC*. The global view of the protocol is in Figure 6.5. Note that cancellation can occur either from *REQ* or *CONF*, since the *Buyer* can start sending *cancellation* even before receiving *confirmation* message.

![Global model](image)

**Figure 6.5: Global model**

We now present the formal specification of the protocol in Event-B. Assume that we have a variable called *state* to represent the current global status. Initially, the state is *INIT*, and we have the corresponding events for *requests*, *confirms* and *cancels*. 

58
6.2.2 Patterns

For this protocol, we are going to use the Single-Message Communication pattern as described earlier in Section 6.1.4. For the purpose of our investigation, we present here the graphic choreographic model (in Figure 6.6) and the detailed specification of the pattern (in Figure 6.7).

![Figure 6.6: Single-Message Communication Pattern](image)

![Figure 6.7: Single-Message Communication Pattern Specification](image)
Moreover, we are going to use another pattern called Request-Confirm. The pattern is developed using the Single-Message Communication pattern twice (as presented in [HFA09]). The summary of the pattern including a choreography model (in Figure 6.8) and the formal specification in Event-B (in Figure 6.9) is as follows.

![Request-Confirm Pattern](image)

variables: req, conf

invariants:  
\[ \text{conf} = \text{FALSE} \Rightarrow \text{req} = \text{TRUE} \]

requests
when
req = FALSE
then
req := TRUE
end

confirms
when
req = TRUE
conf = FALSE
then
conf := TRUE
end

Figure 6.8: Request-Confirm Pattern

Figure 6.9: Request-Confirm Pattern Specification

### 6.2.3 Matching as Seen from Choreography Model

Given the problem at hand as shown in Figure 6.5 and the two patterns in Figure 6.6 and Figure 6.8. Our ideas is to use the Single-Message Communication pattern for cancellation phase and the Request-Confirm pattern for the request and confirm phase. The information on how the patterns are used is gathered in the choreography model by matching the events of the patterns with the events of the problem. For our example, the detail of the event matching is as follows.

<table>
<thead>
<tr>
<th>Pattern events</th>
<th>Problem events</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Request-Confirm) requests</td>
<td>requests</td>
</tr>
<tr>
<td>(Request-Confirm) confirms</td>
<td>confirms</td>
</tr>
<tr>
<td>(Single-Message) transfers</td>
<td>cancels (from REQ)</td>
</tr>
<tr>
<td>(Single-Message) transfers</td>
<td>cancels (from CONF)</td>
</tr>
</tbody>
</table>
Given the above matching of events, the matching of the states can be derived as follows.

<table>
<thead>
<tr>
<th>Problem state</th>
<th>Pattern state</th>
</tr>
</thead>
<tbody>
<tr>
<td>state = INIT</td>
<td>req = FALSE ∧ conf = FALSE</td>
</tr>
<tr>
<td>state = REQ</td>
<td>req = TRUE ∧ conf = FALSE ∧ trans = FALSE</td>
</tr>
<tr>
<td>state = CONF</td>
<td>req = TRUE ∧ conf = TRUE ∧ trans = FALSE</td>
</tr>
<tr>
<td>state = CANC</td>
<td>trans = TRUE</td>
</tr>
</tbody>
</table>

The matching can be viewed from the choreography models as follows in Figure 6.10.

![Figure 6.10: Matching for Choreography Models](image)

From the above matching information, we construct the following refinement in Event-B for the protocol as follows. Here, we split the event cancels into two different events according to the starting state, i.e. REQ or CONF.

**variables:** req, conf, trans
**invariants:**

\[
\begin{align*}
\text{conf} &= \text{FALSE} \Rightarrow \text{req} = \text{TRUE} \\
\text{trans} &\in \text{BOOL} \\
\text{state} &= \text{INIT} \Leftrightarrow \text{req} = \text{FALSE} \land \text{conf} = \text{FALSE} \\
\text{state} &= \text{REQ} \Leftrightarrow \text{req} = \text{TRUE} \land \text{conf} = \text{FALSE} \land \text{trans} = \text{FALSE} \\
\text{state} &= \text{CONF} \Leftrightarrow \text{req} = \text{TRUE} \land \text{conf} = \text{TRUE} \land \text{trans} = \text{FALSE} \\
\text{state} &= \text{CANC} \Leftrightarrow \text{trans} = \text{FALSE}
\end{align*}
\]

\[
\begin{align*}
\text{init} &\begin{align*}
\text{begin} \\
\text{req} &:= \text{FALSE} \\
\text{conf} &:= \text{FALSE} \\
\text{trans} &:= \text{FALSE} \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{requests} &\begin{align*}
\text{when} \\
\text{req} &:= \text{FALSE} \\
\text{trans} &:= \text{FALSE} \\
\text{then} \\
\text{req} &:= \text{TRUE} \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{confirms} &\begin{align*}
\text{when} \\
\text{req} &:= \text{TRUE} \\
\text{conf} &:= \text{FALSE} \\
\text{trans} &:= \text{FALSE} \\
\text{then} \\
\text{conf} &:= \text{TRUE} \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{cancels\_from\_REQ} &\begin{align*}
\text{when} \\
\text{req} &:= \text{TRUE} \\
\text{conf} &:= \text{FALSE} \\
\text{trans} &:= \text{FALSE} \\
\text{then} \\
\text{trans} &:= \text{TRUE} \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{cancels\_from\_CONF} &\begin{align*}
\text{when} \\
\text{req} &:= \text{TRUE} \\
\text{conf} &:= \text{TRUE} \\
\text{trans} &:= \text{FALSE} \\
\text{then} \\
\text{trans} &:= \text{TRUE} \\
\text{end}
\end{align*}
\]

Given the above model, the application of two patterns, i.e. *Single-Message Communication* and *Request-Confirm* is straightforward. The resulting model is obtained by combining the refinement models of the two patterns with the technique as described earlier. We will not go into further details for this development.

### 6.2.4 Conclusion

In this section, we sketch our initial idea for combining pattern approach with graphical interface. The key aspect is to generate the input for the pattern approach by matching different choreography models: the problem model and pos-
sibly several pattern models.

We propose here that the matching of the choreography models based on events and the matching of the states can be generated. Here, the matching of the states are generated as invariants which need to be proved. In general, additionally, developers might need to annotate further information about the matching of the state. However, we expect this annotation to be quite easy and straightforward.
Chapter 7

Feedback Loop for Diagrammatic Modelling

7.1 Summary

The modelling of service choreographies in the domain of business applications is non-trivial and often error prone. The use of formal methods for model verification is emerging as a promising approach. A prerequisite for the formal verification of choreography models, is the transformation of the choreography model in a formal language of a verification tool as described in previous chapters.

In case of model errors the results provided by such verification tools usually are inappropriate to localise the source of the error within the choreography model. Theorem provers such as Rodin describe proof situations; in case of model errors a specific proof goal, which is part of a proof obligation, cannot be derived. A proof goal is a logical expression, which consists of logical connectives and quantification, making it hard to identify a certain model element in the choreography model. Although verification by means of model checking would result in a counterexample consisting of a sequence of actions, the result cannot be mapped statically back to the choreography model.

The goal of the work described in this chapter is to explore ways to visualise the dynamics of models and model analysis results within a translation-based approach to formal modelling. The scope is limited to the domain of choreography modeling for, so that domain-specific (meta-)information can be used; there is no ambition to have a generalised solution for all kinds of domains or properties.

There are three active areas which we consider:

(1) Visualising model behaviour in order to validate whether a model fits the expectations of the modeller using interactive model simulation.
(2) Visualising model checking results in order to enable developers best to localize and fix a model flaw.

(3) Methods for supporting expert users in interactive proving and analysis of undischarged proof obligations with the help of visualising proof states in the choreography diagram.

We apply the ProB model checker to satisfactorily solve the first two items for two error classes: the inconsumable messages property and deadlock-freeness. The implemented framework is however much more general and can deal with all kinds of model problems.

The third challenge is by nature quite intricate and though the technique – visualising state information found in the hypothesis of a proof situation within the choreography diagram – works well, the impact on the proving process remains questionable: in most situations, the visualisable parts of the proof are not the biggest problems.

All prototypes described in this chapter have been implemented as described in [Sch09]. The experiments have shown that Rodin can be very nicely extended with the intended functionality.

### 7.2 Simulation

We need to be sure that the MCM model corresponds to what the modeler intended to express and what has been informally described in requirements documents.

Using the formal representation in Event-B, we have used ProB as a model checker backend to interactively simulate sending and receiving of messages. In each state of the simulation the active states of the partners as well as of the channel and the currently possible messages to send or receive are presented to the modeller. He/she may then manually select one of them which leads the simulation into the next state.

Figure 7.1 illustrates this. The highlighted state depicts the current state of the partners and the highlighted interaction can be clicked by the user to perform send/receive actions (here: sending Offer is possible).

On the left hand side, the state of the channel (EO and EOIO channel in both directions) and the history of events is depicted.
7.3 Model Checking and Error Visualisation

7.3.1 Visualisation Strategies

We are taking a pragmatic approach to error visualisation by classifying errors in classes and specifying concrete visualisation strategies for these classes. To support MCM users in model verification without introducing further complexity, the representation of model errors must be consistent across different error types. Generic information such as the type and the description of errors has to be presented uniformly for all error types.

Since the MCM Editor is based on the Eclipse platform, the error representation is compliant to the Eclipse UI guidelines, e.g. using the standard Eclipse Problem View (Figure 7.2).

Error information concerning the state of the Message Choreography Model has to be presented directly within the MCM diagram by highlighting the corresponding diagram elements.

Traces generated by model checking, cannot be statically mapped to the MCM diagram and have to be simulated dynamically. A dedicated Error Trace View (Figure 7.3) that displays the send and receive events leading to the error and provides graphical animation functionality (in the same style as reported in Section 7.2) is implemented.
We now consider the error types typically expected in choreography models:

**Inconsumable Messages**

In MCM interactions between the partners (i.e. process components) participating in the choreography are modeled via message exchange. A message is called inconsumable if it can be received according to the channel model and cannot be received in the local partner model. In other words an inconsumable message is a message that has been sent, i.e. it is in the message channel, and the addressed partner is in, or enters, a state in which he cannot receive the message, i.e. take the message out of the channel. Since the inconsumable message property deals with the receipt of messages it has to be checked on the local model.

**Example 7** The Request interaction in the running example introduced in chapter 2 is constrained by an additional precondition pre:

\[
\text{msg.Header.ID} \notin \text{ID\_SET}
\]

claiming that the ID of the message to be sent respectively received during the interaction must not be in ID\_SET, which is a global set variable that contains the identifier of all messages that have been sent and not cancelled or shipped so far. This prevents the buyer from sending the same Request twice. In the local model there is exactly one ID\_SET variable for each partner, which is referred to as id_B for the buyer respectively id_S for the seller in the remainder of this section. Since the sequencing context of the communication channel between the two partners does not guarantee the preservation of the message order, a (cancellation) message sent by the buyer might be overtaken by another message. If the buyer
sends a cancellation message, the ID of the cancelled Request $m_1$ is removed from $id_B$ and the buyer can send $m_1$ again. Until receiving the cancellation message, the seller is not able to receive (process) $m_1$ since $id_S$ still contains the ID of $m_1$ and the additional precondition pre does not hold. Hence the second Request is inconsumable.

**Detection** To detect inconsumable messages, two invariants have been defined representing the different sequencing contexts of the MCM channel model, i.e. *Exactly Once* (EO) and *Exactly Once in Order* (EOIO). These invariants are generated automatically during the translation process described in Chapter 3.

$$\forall m \cdot m \in C_{EO} \land m \in \text{dom}(type) \land type(m) = t \Rightarrow (\text{guards}(e_1) \lor \ldots \lor \text{guards}(e_n))$$

The invariant for EO states that if a message $m$ of type $t$ is in the EO channel, the guards of at least one receive event $e_i$ of type $t$ must hold.

$$\forall m \cdot m \in C_{EOIO} \land m \in \text{dom}(type) \land type(m) = t \land (\forall y \cdot y \in C_{EOIO} \Rightarrow y \geq m) \Rightarrow (\text{guards}(e_1) \lor \ldots \lor \text{guards}(e_n))$$

The invariant for EOIO additionally requires that $m$ must be the first message in the EOIO channel, i.e. the identifier of all other messages $y$ in the EOIO channel must be higher than the identifier of $m$.

**Analysis** In general there are two reasons for inconsumable messages. First the addressed partner is in a state (status value) with no outgoing receive interaction $I$ of type $t$. Second the receive interaction $I$ is disabled due to an additional precondition that does not hold in the current state. Depending on the case either a receive interaction has to be added to the corresponding status value or the precondition disabling the receive interaction has to be modified. To distinguish between the two cases, the violated invariant has to be analyzed in detail. An inconsumable message invariant is a logical implication $\phi \Rightarrow \psi$ consisting of an antecedent $\phi$ and a consequent $\psi$. The antecedent $\phi$ is defined as

$$\forall m \cdot m \in C \land m \in \text{dom}(type) \land type(m) = t$$

and restricts the consequent $\psi$ to messages $m$ of type $t$ within one of the channels. This is done via the function $type : \mathbb{N} \rightarrow T$. Since $type$ is a partial function and therefore has a limited domain, $m \in \text{dom}(type)$ ensures that $m$ is within the domain of types. The additional claim of the EOIO invariant can be ignored, since
it is a direct consequence of the channel type (sequencing context) and therefore does not provide any further information. So the first part $\phi$ of the invariant essentially provides information on the type $t$ of the inconsumable message and the message channel including the direction of communication and the sequencing context. The consequent $\psi$ is defined as

$$
guards(e_1) \lor guards(e_2) \lor \ldots \lor guards(e_n)
$$

for all receive events $e_i$ of type $t$ and states that there exists a receive event $e_i$ of type $t$ that is enabled, i.e. the guards of $e_i$ hold; or in a more formal way $\exists e.\text{type}(e) = t \land guards(e)$. Guard conditions $guards(e) : stat \land pre$ consist of a part $stat$ restricting the allocation of the status variables via equality relations, e.g. $st_B = \text{reserved}$, and the optional additional preconditions $pre$ of the interaction represented by $e_i$, which are connected via logical conjunction. To determine if a message cannot be received due to an additional precondition or the lack of a receive interaction in the current state, it has to be analyzed which part of the guards in $\psi$ do not hold. If there exists a guard where the equality relation $stat$ on the status variables holds, the inconsumable message is caused by the additional precondition of the interaction related to this guard; otherwise there is no receive interaction connected with the current status value.

**Representation** Based on the information that can be obtained from the invariant violation as discussed above, the representation of inconsumable messages should be as follows (Figure 7.4). The state, i.e. the enabled status values of the partner addressed by the inconsumable message $m_{inc}$ and the interactions of $\text{type}(m_{inc})$ should be displayed directly within the LPM diagram of the addressed partner, by highlighting the corresponding model elements. Furthermore the send interaction of the inconsumable message should be highlighted. For the information on the reason for the inconsumable message we have to distinguish the two cases discussed above. The first case where none of the $stat$ conditions holds is trivial and does not need additional representation. For the second case caused by an additional precondition, an error marker should be added to the affected interaction, giving information on the blocking precondition.

The information on the error context should also be displayed in the Eclipse Problem view. For the example introduced above, an inconsumable message error should be displayed, including the following description.
A message of type ProductAvailableToPromiseCheckRequest that has been sent by buyer during the interaction Request cannot be received by seller.

Reason: Additional precondition in the selected interaction Request does not hold.

Lock-Freedom

In contrast to our running example introduced in Chapter 1 the participating services in many real-world choreography models can send messages concurrently. For instance, consider an extended model where both partners concurrently can send a change request that has to be confirmed by the other partner; if the buyer sends a change request with a new quantity and the seller concurrently sends a change request with a new delivery date, both partners would wait infinitely for a response. This situation is called deadlock. If we consider a situation where the choreography terminates with at least one partner being in a non-target state, deadlocks may even occur in the running example.

A livelock is a special kind of deadlock that describes a situation with at least one partner being in an infinite loop that does not progress towards a target state and cannot be leaved. In both situations at least one of the partners never reaches a valid target state; the choreography is considered as locked.

Deadlocks

Deadlocks are a well understood problem in the domain of model checking and detection mechanisms for deadlocks usually are an integral part of model checking tools such as ProB. Therefore no additional properties have to be defined to detect deadlocks.

Real-life MCM models usually have an infinite state space and therefore cannot be exhaustively checked via model checking. A limitation of deadlock detection to the GCM, which handles sending and receiving of messages as a single step and ignores aspects of the communication channel, could reduce computation time. The local model refines the GCM by shifting the global observer perspective to a local partner perspective (local partner models and channel model) with distinct send- and receive-events. If the aspects of the communication channel are ignored, deadlocks caused by race conditions, such as the one in the opening example cannot be detected. Indeed the inconsumable message property discussed in 7.3.1 ensures that all messages in the communication channel can be received at all times, but since there is no central control instance in choreographies it cannot
be ensured that the seller receives and confirms the change request of the buyer before he sends his change request.

Other points that have to be considered are additional preconditions and inhibiting constraints in the LPMs; such constraints inhibit a partner from sending a message in a certain state. For instance a local constraint in the ordering choreography model, inhibiting the seller from sending a Confirm message in the state requested would lead to a deadlock.

Figure 7.4 shows a deadlock in a modified version of the model introduced in Chapter 2. An additional precondition inhibits the seller from sending the Confirm message.

**Representation** A possible representation of a deadlock in MCM is shown in Figure 7.4. The deadlocked status values and the last executed interaction are highlighted in the diagram. Additionally the sequence of send and receive interactions leading to the deadlock is shown in the Error Trace view and can be animated in the diagram.

![Figure 7.4: Deadlocked Seller/Buyer Choreography](image)

**Resolution Strategies** Typical resolution strategies for race problems in service choreographies are introduced in [Dec08]. The authors argue that a total sequentialization of choreographies, where only one partner is allowed to send messages at a time, typically is not feasible in real-world scenarios. Instead they list different strategies applied in practice, where either (a) the outcome upon conflicting messages is predefined, e.g. considering one message while ignoring the other.
messages, or (b) the outcome is negotiated between the partners or determined by one or several partners.

**Liveloocks**

To detect livelocks with ProB for Eclipse, the livelock-freedom property has to be specified via linear temporal logic (LTL). The LTL formula

\[ \phi := \Box \Diamond \{ \text{targetState} = \text{true} \} \]

where `targetState` is a global Boolean variable that is true iff the choreography is in a target state, implements this property. The formula can be interpreted as “for each reachable state there must exist a path to a target state”. Since the formula does not refer to the specifics of livelocks, which is the infinite loop, it also detects deadlocks.

**7.3.2 Realisation**

We have implemented a research prototype demonstrating the feasibility of the above described approach.

Figure 7.5 shows an FMC\(^1\) Block Diagram that illustrates the architectural structure of the verification prototype and the integration into the MCM tool suite.

The MCM tool suite is technologically based on Eclipse. SAP NetWeaver Developer Studio is SAP’s environment for developing Java-based, multiple-layered business applications and is based on the Eclipse Rich Client Platform (Eclipse RCP). It leverages the Eclipse extension framework Equinox\(^2\), which is an implementation of the OSGi R4 core framework specifications [OSG]. Eclipse is a highly modular IDE that can be extended via plug-ins, which are the Eclipse implementation of OSGi bundles. The implementation is based on the OMG MetaObject Facility (MOF) specification [MOF]. In a nutshell, MOF defines a language to define metamodels, which describe the abstract syntax of domain-specific languages such as MCM. Furthermore MOF defines methods for model serialization, transformation and code generation, and is the foundation for the OMG’s Model Driven Architecture (MDA [MDA]).

MCM consists of several components build around a common metamodel. Each component, including the MCM metamodel, is implemented as Eclipse plug-in. The core component of MCM is the Choreography Editor, which provides graphical modeling functionality for service choreographies, based on the

\(^2\)http://www.eclipse.org/equinox
MCM metamodel. Other components, such as the Event-B Generator, integrate into the editor, using the extension mechanisms provided by Eclipse.

The Event-B Generator plug-in generates an Event-B representation for service choreography models (see Chapter 3). The generated Event-B models are the foundation for the formal verification of MCM service choreographies; just like the choreography models, the Event-B representation is stored in a dedicated repository within the workspace.

**MCM Verification Prototype**

The functionality of the verification prototype is separated in three plug-ins:

- The Model Checking & Animation (core) plug-in integrates model checking functionality, provided by the ProB model checker, into MCM and provides an interface for accessing choreography models and the associated diagrams. The model checking and the model animation functionality is separated in two subcomponents, both of them operate on the Event-B representation of the choreography model, retrieved from the repository.

- The Choreography Animation plug-in uses the model animation functionality provided by the core plug-in and implements an interactive animation
The architecture separates the core model checking functionality, which integrates ProB into the MCM framework, from the application logic implemented by the Choreography Animation and the Choreography Verification plug-ins; this simplifies reusing the model checking functionality in other applications and reduces implementation and maintenance effort. The model animation and the model checking subcomponents of the core plug-in basically could also be separated in different plug-ins. However, this would introduce further complexity, since both subcomponents are based on the same foundation and a tightly coupled third plug-in that implements the common functionality would be necessary.

The internal structure of the Choreography Verification plug-in is realized by loosely coupled subcomponents, separating error detection from error handling; this allows adding new error handlers without consequences to error detection. Vice versa new error types can be added without modifying error handling.

### 7.3.3 Example Application

Figure 7.6 shows a screenshot of the inconsumable message error discussed in Example 7, visualized in the Buyer/Seller choreography. The status values of buyer and seller are displayed and the affected interaction Request is highlighted. The inconsumable message error is shown in the Eclipse problem view (Figure 7.7) and a detailed error description is displayed in a dedicated view.

The error description states that the message of type OrderRequest sent by the buyer during the interaction Request cannot be received by the seller. Furthermore generic solution proposals for inconsumable messages are given. The modeler either has to ensure that the guards of the highlighted interaction Request hold for the seller, or has to prevent the buyer from sending a Request in the previous state.

The next step for the modeler is to analyzer why the seller cannot receive the Request. A look at the error visualization in the diagram shows that seller is in reserved, so the additional precondition of Request must block. If an additional precondition of an interaction blocks, a small warning icon is displayed at the
Figure 7.6: A visualised “inconsumable message” error in the diagram.

Figure 7.7: An inconsumable message problem in the problem view

affected interaction (see Figure 7.6). The warning message for the Request interaction states:

```
Additional Precondition does not hold because
msg.Header.ID ∈ ID_SET
where msg.Header.ID == Id1
where ID_SET == {Id1}
```

So the seller cannot receive the Request because the ID is already in the seller’s id set. A look at the sequence of events leading to the inconsumable message error (Figure 7.8) reveals that the buyer canceled his first Request and immediately sent another Request with the same id. Since the seller did not receive the buyer’s cancellation message, the additional precondition for the receive Request interaction blocks.
7.4 Towards Visualisation Support for Proofs

For real-life Message Choreography Models, the Rodin provers often fail to automatically discharge all invariant preservation statements and some proofs have to be done manually.

In the case of a failed automated proof attempt the modeler is asked to check whether the proof obligation is in fact not provable or whether the proof could be completed because the prover was not strong enough. In the latter case, manual interactions with the prover are necessary or the model has to be enriched with additional invariants making the proof automatic. In any case, user interaction is required. Since our deployment strategy demands a translation into the formalism a modeler is used to (here: the choreography modeling language) we need to map the prover feedback back to that formalism.

In this section we report on an initial experiment we have done in this respect.

7.4.1 Running Example and Proof Assistance

The selected hypotheses of the sequent, generated for the Send Request event $e$ and the inconsumable message invariant $I$ for Request messages, of the Seller/Buyer choreography model introduced in Chapter 2, comprise $I$ and the guards of $e$ where

$I$: $\forall m' \cdot m' \in C_{cv} \land m' \in \text{dom}(\text{type}) \land \text{type}(m') = \text{Request} \\
\Rightarrow (st_a = \text{reserved} \lor st_a = \text{start}) \land \\
m' \in \text{dom}(ID) \land ID(m') \notin ID\_\text{SET}_a$

$\text{grd}_1: \quad st_b = \text{reserved} \lor st_b = \text{start}$
$\text{grd}_2: \quad t_1 \notin ID\_\text{SET}_b$
$\text{grd}_3: \quad m \in \text{dom}(ID) \Rightarrow ID(m) = t_1$
$\text{grd}_4: \quad st_b \neq \text{reserved}$

Sets and variables associated with one of the choreography partners are denoted by the subscript $b$ for the buyer and $s$ for the seller system. The message channel from the buyer to the seller system is denoted by $C_{cv}$. The variable $st$
denotes the partner status. $ID\_SET$ denotes the set of uncanceled $Request$ messages sent by the buyer respectively received by the seller. The variable $m$ denotes the id of the $Request$ message sent during $e$; $t_1$ denotes the ID parameter of $e$, and $m'$ denotes the id of an $Request$ message in the channel $C_{cv}$.

The goal $G$ of the sequent is generated by applying the before-after predicate on $I$, which leads to the predicate $I_g$:

$$
\forall m' \cdot m' \in C_{cv} \cup \{m\} \land m' \in \text{dom}(type \triangleleft \{m \mapsto Request\}) \\
\land (type \triangleleft \{m \mapsto Request\})(m') = Request \\
\Rightarrow (st_a = reserved \lor st_a = start) \\
\land m' \in \text{dom}(ID \triangleleft \{m \mapsto t_1\}) \\
\land (ID \triangleleft \{m \mapsto t_1\})(m') \notin ID\_SET_s
$$

The before-after predicate represents the actions of $e$; in detail $m$, which denotes the id of the current message, is added to the message channel $C_{cv}$ and the type function is extended by the binary relation $m \mapsto Request$, which defines a mapping between the message id and the message type. Furthermore the $ID$ function is extended by the relation $m \mapsto t_1$, which connects the ID $t_1$ of the Request message header with the message $m$.

In first stage the hypotheses can be simplified by doing a case distinction on $grd_1$:

- The first case with $st_b = reserved$ contradicts $grd_4$; since there is a contradiction in the sequent hypotheses, the goal of the sequent is immediately proved automatically.

- In the second case $grd_1$ is replaced by $st_b = start$ and $grd_4$ can be removed from the selected hypotheses:

  $grd_1 : \quad st_b = start$
  $grd_2 : \quad t_1 \notin ID\_SET_b$
  $grd_3 : \quad m \in \text{dom}(ID) \Rightarrow ID(m) = t_1$

In the next step $I_g$ can be simplified by eliminating the universal quantifier (universal instantiation) and doing a case distinction on the consequent in $I_g$ leading to the following three cases, which have all to be proved:

(1) $\Rightarrow \quad st_s = reserved \lor st_s = start$

(2) $\Rightarrow \quad m' \in \text{dom}(ID \triangleleft \{m \mapsto t_1\})$

(3) $\Rightarrow \quad (ID \triangleleft \{m \mapsto t_1\})(m') \notin ID\_SET_s$
Since the theorem prover fails to automatically discharge the three cases, each of them has to be reviewed in detail. The sequents of the three cases can be interpreted as follows: “If the guards of the Send Request interaction and the invariant $I$ hold, then

1. the seller system must be in status $start$ or $reserved$.”

2. the $Request$ message $m'$ must be in the domain of the $ID$ function.”

3. the ID of the $Request$ message $m'$ must not be in the set of received $Request$ messages of the seller.”

By reviewing the choreography model, the hypotheses of the sequents can be considered as true, since there is at least one state where the guards of Send Request hold and the preservation of $I$ before $e$ is executed is considered as true anyway (this has to be proved for each event of the model, including the initialization).

The first case can be validated by searching for a counterexample, where the seller system is in status $requested$ or $shipped$. At initialization the seller is in $start$, there is no message in the channel, and none of the interactions is enabled for the seller. If the buyer sends a $Request$ message, the seller is in $start$ and switches to $requested$ after receiving the $Request$. The only chance for the buyer to send a new $Request$ is to cancel the old one, therefore the buyer has to be in status $reserved$. The only way for the buyer to reach the $reserved$ status is to receive a $Confirm$ message from the seller; this means before buyer can switch to $reserved$, the seller must be in $reserved$. Since the buyer is in $requested$ and the seller is in $reserved$, there is no shipment request in the message channel and the seller cannot switch to the $shipped$ status. So the seller either must be in $start$ or in $reserved$ before the buyer can send an $Request$, which means case one is valid and must be provable.

The second case with $m' \in \text{dom}(ID \prec \{m \mapsto t_1\})$ is trivial, since $m'$ is added to the domain of the $ID$ function in the actions of $e$; in this case $m' = m$. After instantiating $m'$ in the universal quantifier of $I$, case two is proved automatically.

For the third case where the ID of the $Request$ message must not be in the set $ID_SET_S$ of received $Request$ messages of the seller, the additional effects of the interactions have to be analyzed in detail. $Receive Request$ is the only interaction where an ID is added to $ID_SET_S$. The only way for the seller to remove the ID from $ID_SET_S$ is by receiving a cancellation message. So the seller must have received a $Request$ but no $Cancel$. The buyer must be in $start$ to send a new $Request$, so he must have sent a $Cancel$ that has not yet been received by the seller and the seller therefore is in status $reserved$. Now since the ID of the canceled $Request$ has been removed from the buyer’s set $ID_SET_B$ of sent
Request messages, the buyer can send another Request with the same ID. Since the seller has not received the Cancel, the ID is still in ID_SET. So case three is not valid and disproves the invariant preservation statement.

Considering the example proof discussed above, manual proving of MCM invariant preservation statements can be supported by the following tools, which we have prototypically implemented.

A. Proof Context Visualization Visualizing the state of the choreography described by the sequent hypotheses helps to understand the proof obligation. There are two possible approaches for visualization:

1. Direct visualization of the choreography partner status values based on pattern matching

2. Animation of traces derived from the proof sequent by model checking techniques

The first approach directly visualizes status information extracted from the hypotheses by pattern matching, such as

- \( grd_1 : \quad st_b = \text{reserved} \lor st_b = \text{start} \)
- \( grd_4 : \quad st_b \neq \text{reserved} \)

in the proof in Section 7.4.1. In contrast, the second approach uses model checking techniques to determine a state \( s \) that fulfills all sequent hypotheses and visualizes the status information extracted from \( s \). While the first approach can be applied on-the-fly, unaffected from the size of the model state space, the second approach relies on the performance of model checking; especially for MCM models with infinite state space this might be a problem. Nevertheless the trace derivation approach has some advantages over direct visualization:

1. States derived by model checking can be considered as reachable, while direct visualization also visualizes unreachable states.

2. Trace animation always results in complete state information, whereas the pattern based approach relies on the completeness and correctness of the pattern basis. Information that is not directly included in the hypotheses, such as the state of the message channel, cannot be displayed with a pure pattern based visualization approach; in contrast, the trace animation approach complements missing state information by searching for a complete example state that fulfills the hypotheses.
(3) Trace derivation allows to determine the state information after $e$ is executed, while direct visualization is restricted to the state information given in the hypotheses, which is the state before $e$ is executed.

Thus, both approaches are useful for supporting MCM modelers.

B. Case Distinction Support

Case distinction on the status of the choreography partners is a technique which we experienced to be useful when doing manual proofs. Also this technique is quite intuitive to present on the graphical level. Two variants are possible:

1. **Manual Case Distinction** on status values within the MCM diagram

2. **Automatic Case Distinction** based on trace derivation by model checking

In the manual approach the decision on the cases completely remains with the user. The main difference between the case distinction functionality provided by Rodin is the integration into the MCM diagram; this enables the user to do a case distinction directly on a selected status value within the MCM diagram, and complements the direct status visualization approach A1.

In contrast, the second approach uses model checking to determine a state $s$ that fulfills the hypotheses and automatically does a case distinction based on relevant status information extracted from $s$. For instance for the example discussed in Section 7.4.1, the hypotheses already contain the status of buyer ($grd_1$ and $grd_2$), the automatic case distinction approach therefore determines a possible status value for the seller.

Such as the model checking based visualization approach, automatic case distinction might fail for models with large state spaces.

C. Sequent Disproving

To find a counterexample that disproves a proof sequent, such as in case three of the goal in the motivating example, is usually a tedious task that requires consideration of all constraints and possible state variations; even for relatively small MCM models this is hard to achieve. Since the state space of MCM models usually is infinite, model checking cannot be used for proving MCM sequents, but it can be used to support the process of searching for a counterexample that disproves a given sequent.

Disproving is possible with the help of the ProB Disprover plug-in introduced in [LBL07], which translates sequents into Event-B machines with a single event `disproveHypotheses` with the guard

$$\exists x_1, \ldots, x_k : \mathcal{H}(x_1, \ldots, x_k) \implies \neg \mathcal{G}(x_1, \ldots, x_k)$$
If the guard holds, the disprover extracts a counterexample by finding a valuation for $x_1, \ldots, x_k$ that makes the implication true.

The drawback of this approach is that the generated counterexamples include no trace information; this means it is left unclear if the generated state is reachable within the actual model, i.e. if there exists a valid trace to the state. In contrast, counterexamples generated by model checking on the original Event-B model include trace information and only comprise states that actually can be reached. A disprover based on the second approach combined with graphical trace animation therefore seems worthwhile.

7.4.2 Automatic Trace Derivation from MCM Proof Obligations

The model checking based approaches in the use cases discussed above try to answer the question: how (i.e. with which trace) can the current proof situation be reached?). They thus rely on trace derivation from proof sequents, which is discussed in the following.

An MCM invariant preservation proof obligation is a sequent $\mathcal{H} \vdash \mathcal{G}$. To generate traces from such sequents with the ProB model checker for Eclipse, the sequents have to be represented as linear time properties in form of linear time temporal logic (LTL) formulae.

A straightforward translation of sequents to LTL formulae of the form $\Box (\mathcal{H} \Rightarrow \mathcal{G})$, which can be interpreted as “each state that satisfies the hypotheses $\mathcal{H}$ must satisfy $\mathcal{G}$” is insufficient since:

- The generated traces do not include $e$ as the last executed event, but just lead to a state in which the guards of $e$ hold (provided the guards are part of the selected hypotheses) and $I_e$, which is $I$ modified by the before-after predicate of $e$, does not. This means the state $s$, the trace is leading to, does not violate the original invariant $I$, rather the execution of $e$ in $s$ violates $I$.

- The intention behind invariant preservation statements is to prove that $I$ still holds after $e$ is executed. The execution of $e$, which is represented in form of the before-after predicate in $I_e$, is an integral part of the proof goal. Therefore the guards of $e$ have to be in $\mathcal{H}$; if one of the guards is removed from the selected hypotheses, the modified invariant $I_e$ is also checked in states where $e$ is disabled.

In addition to $\Box (\mathcal{H} \Rightarrow \mathcal{G})$ it has therefore to be ensured that

1. only states are considered where the guards of $e$ hold, though the guards may not be in $\mathcal{H}$
(2) \( e \) is added to the trace in order to reach a state where \( I \) is violated

These requirements can be solved by the use of the next executed operation predicate \([e]\) of ProB, which has to be used as antecedent of an implication. It specifies whether the next executed operation in the path is \( e \).

We use the formula

\[
\phi_e := □(H ∧ [e] ⇒ G)
\]

which can be interpreted as “each state that satisfies \( H \) and in which the next executed operation is \( e \) must satisfy \( G \)”.

Model checking with LTL results in a counterexample iff the model checker finds a state that violates the LTL formula; the formula therefore evaluates to false iff there exists a state \( s \) where

- the hypothesis predicates \( H \) hold
- the goal \( G \) does not hold
- the event \( e \) can be executed

In contrast to disproving, the remaining approaches - proof context visualization and case distinction support - require an example state that fulfills the hypotheses of the sequent, instead of a counterexample; this can be reached by using

\[
\phi'_e := □([e] ⇒ ¬H)
\]

which considers all possible states that fulfill the hypotheses, including states that violate the goal.

For the case distinction approach, a further modification is necessary; since the case distinction is done in the hypotheses of the sequent, which represent the state before \( e \) is executed, the after execution part in \( \phi'_e \) has to be removed. Nevertheless it has to be ensured that the guards of \( e \) hold; this leads to the formula

\[
\phi''_e := □(enabled(e) ⇒ ¬H)
\]

which can be interpreted as “for each state where the guards of \( e \) hold, return false and produce a trace iff the hypothesis predicates \( H \) hold”.

ProB always finds the same state \( st \) violating \( \phi'_e \), to find another state \( st' \neq st \) violating \( \phi''_e \), the properties \( p \) of \( st \) that should differ in \( st' \) have to be negated and added to \( H \). For instance model checking with the formula □(enabled(e) ⇒ ¬H ∧ st_B \( \neq \) start) produces a trace to a state where the guards of \( e \) and \( H \) hold and the buyer is not in status start.
7.4.3 Example Application

The proof assistance functionality was tested for several MCM models. In the following the results for the proof discussed in example from Section 7.4.1 are discussed exemplarily. Figure 7.9 shows a screenshot of the Seller/Buyer Choreography model with enabled proof assistance for the proof obligation discussed in that example.

![Figure 7.9: Proof Visualization and Manual Case Distinction](image)

The direct visualization feature visualizes the status information in $grd_1$ and $grd_4$ of the root sequent hypotheses. The visualization highlights the Request interaction associated with the proof obligation and shows that the buyer either must be in \textit{start} or in \textit{reserved}, and there is an additional inhibiting constraint that states buyer must not be in reserved. So manual case distinction on the start value can be used to focus on the case $st_B = \text{start}$; the other case can be proved automatically, because of the contradiction in the hypotheses. The status of the seller either can be set by manual case distinction or determined automatically. The automatic case distinction is activated directly from the Rodin Proof Control view (Figure 7.10) and adds the case $st_S = \text{start}$, another activation of the automatic case distinction on the sequent where $st_S \neq \text{start}$ finds another case with $st_S = \text{reserved}$.

The disprover does not find any counterexample for the first case with $st_B = st_S = \text{start}$; the state space of the Seller/Buyer Choreography model is infinite, so this does not mean there exists no counterexample, but at least it gives some confidence that the sequent is probably valid and can be proved. For the second case with $st_B = \text{start}$ and $st_S = \text{reserved}$, the disprover generates the following trace:
Figure 7.10: Rodin Proof Control with automatic Case Distinction

$\rightarrow$ Request (ID1)
$\leftarrow$ Request (ID1)
$\rightarrow$ Confirm
$\leftarrow$ Confirm
$\rightarrow$ Cancel (ID1)
$\rightarrow$ Request (ID1)

$\rightarrow$ denotes the send and $\leftarrow$ the receive interactions.

The generated trace reflects the counterexample discussed in the running example. The buyer sends a Cancel for the Request with the ID 1, followed by a new Request with the same ID. Since the seller did not receive the Cancel, the additional precondition of the receive Request interaction does not hold for ID 1 and the second Request cannot be received by the seller.

The proof assistance features based on model checking techniques do not guarantee a result for models with a very large or even infinite state space, such as the Seller/Buyer Choreography model discussed above; nevertheless the results show that these features are useful, especially in combination with the features that do not rely on model checking.
Chapter 8

Model-based Integration Testing

8.1 Summary

After having obtained a formal representation of the MCM model, we can employ model-based testing (MBT) techniques to derive test suites for integration testing. In this chapter we introduce our general testing approach, starting with a discussion of appropriate test objectives.

For automatic test generation, a local model that incorporates information from both LPMs and the CM (to connect the send and receive events) can be used. Because various cases studies (e.g. [CSH03]) show that state space explosion is the major stumbling point when applying automatic test generation to industrial settings, we decided to use the GCM to drive the test generation instead of the much more complex local model. While transition coverage of the GCM is equivalent to receive event coverage of the LPMs in most cases, the state space that needs to be explored is significantly lower. In [WSG08], we discussed possible coverage criteria that can be used to drive service integration testing and how to choose them accordingly depending on effort and fault assumptions. For this approach, we decided to use transition coverage, i.e. that all interactions are contained in the test suite, because it already uncovers a significant amount of integration faults with relatively small efforts [UL07]. For example in the MBIT approach of [ABR+07], transition coverage of a global communication model was able to detect about 90% of integration related faults.

Practical experience with the integration testing process led to a prioritisation of objectives:

1. Each path should start in the initial state and end in a target state
2. The length of the longest generated path should be minimal.
3. Message racing should be minimal.
The number of test steps should be minimal.

We developed an algorithm which derives test suites from an Event-B equivalent of a global choreography model which satisfies these conditions.

The main steps of this procedure are

1. a set of global paths with minimal length found through state exploration with breadth first search
2. mapping of the global paths to send- and receive events
3. optimization of the test suite (removal of redundant paths)
4. generation of executable test scripts

For the first and second step we efficiently apply the ProB model checker. Therefore we have extended ProB to detect transition coverage, and have made use of the possibility to guide an Event-B model by a CSP process in order to translate high-level traces into low-level ones.

The flexibility of ProB was crucial in addressing the various aspects of choreography models. The test suites of various realistic choreography models have been computed automatically by our implementation. As MCM explicitly considers asynchronous communication, the generation of test suites incorporating message racing is a direct contribution to the research community, as is the utilization of a higher level of abstraction (the global model) to compute an integration test suite, thus avoiding the well known problem of state explosion.

This work has been reported in [WKR+09].

8.2 Requirements

Similar to [Wey98], we define integration testing as testing of an assembly of components that were already individually tested. Assuming the correctness of the participating components, our test objectives focus on showing that each exchanged message in a choreography is correctly interpreted. This is also underpinned in [WS09], where the missing consideration of message racing has been identified as a soft spot in SOA integration testing. Message racing here denotes the changes of the message order during service-based communication, which are due to the various effects of loose coupling.

In [WSG08], we discussed possible coverage criteria that can be used to drive service integration testing and how to choose them accordingly depending on effort and fault assumptions. For this approach, we decided to use transition
coverage, i.e. that all interactions are contained in the test suite, because it already uncovers a significant amount of integration faults with relatively small efforts [UL07]. For example in the MBIT approach of [ABR\textsuperscript{+}07], transition coverage of a global communication model was able to detect about 90\% of integration related faults.

For automatic test generation, a local model that incorporates information from both LPMs and the CM (to connect the send and receive events) can be used. Various cases studies (e.g. [CSH03]) show that state space explosion is a major stumbling point when applying automatic test generation to industrial settings. This is mainly due to the combinatorial complexity introduced when combining the local models. Therefore we investigate a test generation using the GCM to cover the LPMs.

Not very surprisingly transition coverage of the global choreography does not guarantee transition coverage of the LPMs. A counterexample can be found in Figure 8.1 (left) where service $A$ sends the messages $a$ and $b$ to service $B$ in arbitrary order over a single EO channel. As illustrated, a test suite consisting of the two test cases, characterized by the differently dotted lines, is covering the GCM but not the send events of the service $A$. Less intuitive is the fact that transition coverage of the LPMs also does not guarantee transition coverage of the GCM. The counterexample depicted in Figure 8.1 (right) shows the global and local models of two services $A$ and $B$. Each is able to initiate a conversation by sending the message $a$ or $b$ respectively. Both services are allowed to send

---

**Figure 8.1:** Assuming an EO channel, global coverage does not imply local coverage (left) and local coverage does not imply global coverage (right)
their message as long as they have not received anything from their partner. The three test cases, again illustrated by dotted lines, are covering the LPMs of service $A$ and service $B$ but not the global model.

Although transition coverage of GCM does not cover all send and receive events of the LPMs, it does however cover the receive events. This is a consequence of the assumed receive semantics of MCM. So, any test suite with transition coverage of the GCM is covering all receive event of the LPMs. These are in fact our main test targets for integration testing, because they may reveal message racing problems. Therefore we decided for transition coverage of the GCM, and hence exploiting the positive effect that the state space that needs to be explored is significantly lower.

Important from an industrial perspective is that our test generation approach further aims to be optimal regarding the minimization of the effort in the subsequent test phases, i.e., test concretization (e.g., provisioning of test data), test execution and test analysis. Based on practical experience of the testing process at SAP [WSS09], we concluded that the test optimization should be driven by the following objectives sorted from highest to lowest priority:

(1) *Each test case should start in the initial state and end in a target state:* As described in [WSS09], bringing a complex system in the needed state using test preambles is complicated and time consuming. Stopping a test while the system is not in a target state leads to problems with inconsistent data that might hamper consequent test executions.

(2) *The length of the longest generated test case should be minimal:* The longer a test case gets, the harder it is to maintain. Therefore especially for generated tests, a top priority is to carefully control path lengths.

(3) *Message racing in the test suite should be minimal:* Testing the effects that message racing has on the interaction is an important part of each test suite. Tests are mostly carried out in rather idealistic environments where messages are received in the same order they have been sent. Therefore, during test execution, message racing has to be emulated on the channel in a controlled way, usually leading to much higher effort.

(4) *The number of test steps should be minimal:* As the effort increases with the overall length of all test cases, the sum of test steps should be minimized.
8.3 Model-based Integration Testing with Event-B and ProB

The core idea of MBT is to use formal specifications for test generation. This implies that tests can only be as precise as the modeled content they use. By design, MCM offers the necessary information to drive the generation of test suites covering the specified interaction protocol. However, the generated test suites have to be supplemented with additional information, because even though the local behavior is modeled in the LPMs, triggers for the local message sending events are not specified. This information cannot be modeled easily as it is deeply rooted in the internal behavior of the components. However, using MBT for service integration promises to reduce the manual effort by automatically generating suitable sets of test cases for desired coverage of the choreography model. As explained in the previous subsection, we decided to focus on transition coverage of the GCM, which requires the following four-step approach for test generation. For a detailed discussion please refer to [WKR+09].

Step 1: A test generator generates a set of globally observable message sequences that cover each transition of the GCM.

To cover each transition of the global communication model, i.e., each interaction of the GCM. To satisfy the first and second of our testing objectives, ProB was extended to detect when transition coverage is obtained. This is gained by exploring the state space of the model using a breadth-first strategy, stopping when full coverage is achieved by the discovered terminating traces.

From the example given in Chapter 2, the following initial set of test cases is obtained:

```plaintext
< Request, Offer, Order >, < Request, Offer, Cancel >,
< Request, Offer, Cancel, Request, Offer, Order >,
< Request, Offer, Cancel, Request, Offer, Cancel >,
< Request, Offer, Request, Offer, Order >,
< Request, Offer, Request, Offer, Cancel(depr.) >,
< Request, Offer, Request, Offer, Cancel >,
< Request, Offer, Request, Offer, Cancel(depr.), Order >,
< Request, Offer, Request, Offer, Cancel(depr.), Cancel >,
< Request, Offer, Request, Cancel(depr.), Offer, Order >,
< Request, Offer, Request, Cancel(depr.), Offer, Cancel >
```

Note that MCM is not suited for the derivation of component tests because apart from the missing triggers mentioned, the behavior modeled in the LPMs focuses on communication only, leaving out internal steps that may happen in between the communication events.
Step 2: The local event sequences corresponding to the test cases are computed. This is necessary because the GCM only specifies the order of receive events. Therefore the receive sequences have to be enhanced by their corresponding send events, taking the LPMs and channel model into account.

In the case of the path

$$[\text{Request}, \text{Confirm}, \text{Request}, \text{Confirm}, \text{Cancel(depr.)}, \text{Cancel}]$$

the corresponding receive-events are (‘?’ reads “receives”):

$$[ \text{Seller}?\text{Request}, \text{Buyer}?\text{Confirm}, \text{Seller}?\text{Request}, \text{Buyer}?\text{Confirm}, \text{Seller}?\text{Cancel(depr.)}, \text{Seller}?\text{Cancel} ]$$

Afterwards for each receive event a corresponding send event is generated and added to the path in such a way that the local behavior descriptions are not violated. In the mentioned sequence the send event for Cancel(deprecated) has to be added before the second Request as the Buyer is not able to send these messages in the same order as they have to be received for the test. The resulting local sequence from our example therefore is (‘!’ reads “sends”):

$$[ \text{Buyer}!\text{Request}, \text{Seller}?\text{Request}, \text{Seller}!\text{Confirm}, \text{Buyer}?\text{Confirm}, \text{Buyer}!\text{Cancel}, \text{Buyer}!\text{Request}, \text{Seller}?\text{Request}, \text{Seller}!\text{Confirm}, \text{Buyer}?\text{Confirm}, \text{Seller}?\text{Cancel(depr.)}, \text{Buyer}!\text{Cancel}, \text{Seller}?\text{Cancel} ]$$

The message racing in the illustrated local path is underlined. While the Cancel message is sent by the buyer before the Request message, the seller receives the Request message first. Similar to Step 1, it is again infeasible to exhaustively explore the full state space (as the state space of the local model is actually even considerably bigger) to find a suitable mapping from global to local traces. One could encode the problem as an LTL formula, but this formula will be very big with ensuing consequences for the complexity of model checking. The solution we have come up with, is to encode the desired LCM scenarios into a CSP \cite{Hoa78} process. This process is synchronized with the Event-B model (using the technology of \cite{BL05}), suitably guiding the model checker.

The CSP Process is divided into two components. The first process encodes the desired trace of receive events, followed by an event on the goal channel, indicating to the model checker that this is a goal state we are looking for. For the trace given above it looks as follows:

\[
RECEIVER = \text{Seller}?\text{Request} \rightarrow \text{Buyer}?\text{Confirm} \rightarrow \\
\text{Seller}?\text{Request} \rightarrow \text{Buyer}?\text{Confirm} \rightarrow \\
\text{Seller}?\text{Cancel(depr.)} \rightarrow \text{Seller}?\text{Cancel} \rightarrow \\
goal \rightarrow \text{STOP}
\]
The second process encodes the sender events. We know how many send events of each type must occur, but the order of these is unknown.

\[ SENDER(n_1, n_2, n_3, n_4) = n_1 > 0 \& \text{Buyer!Request}\rightarrow SENDER(n_1 - 1, n_2, n_3, n_4) \]
\[ n_2 > 0 \& \text{Seller!Confirm}\rightarrow SENDER(n_1, n_2 - 1, n_3, n_4) \]
\[ n_3 > 0 \& \text{Buyer!Cancel}\rightarrow SENDER(n_1, n_2, n_3 - 1, n_4) \]
\[ n_4 > 0 \& \text{Buyer!Order}\rightarrow SENDER(n_1, n_2, n_3, n_4 - 1) \]

The sender process is now simply interleaved with the receiver process.

\[ MAIN = SENDER(2, 2, 2, 0) ||| RECEIVER \]

Now, ProB will ensure that every event of the Event-B model synchronizes with an event of the CSP process (MAIN) guiding it and stopping when the CSP process can perform an event on the goal channel. For the initial test suite from Step 1, we compute a described mapping for each global trace in 0.064 seconds.

**Step 3:** The produced test suites have to be optimized according to the test objectives described above. Depending on the controllability of the actual test generation technology the effort might vary: We made good experience with a mixed-linear programming encoding of the problem and using state-of-the-art solvers which were sufficient for examples with over 3000 test paths resulting from step 2.

For the given example, an optimal test suite would incorporate the local equivalents of the following global paths:

\[ < \text{Request, Offer, Request, Offer, Order, Cancel(depr.)} >, \]
\[ < \text{Request, Offer, Request, Offer, Cancel(depr.), Cancel} >, \]
\[ < \text{Request, Offer, Request, Cancel(depr.), Offer, Order} > \]

**Step 4:** Generated abstract test cases are translated into executable test suites. This step is semi-automatic. We can automatically generate the concrete test steps as well as checks that the components are in the right state using “glue” information for the local components (see Section 10.2). However, as the local message triggers are not fully modeled in MCM, this information has to be added manually to the test sequences.

According to the terminology of [UL07, ch. 8], we use a mixed approach for the test concretization. The transformation from the abstract test cases to an
internal SAP test language follows the keyword-driven testing principles\(^2\), i.e., it builds upon SAP’s eCATT test script language [HLT07] and was designed to address the requirements of integration testing at a higher level of abstraction. This test language contains constructs that can create and modify local business objects, can trigger the sending of messages between the business components via the available enterprise services and can check the values of the internal local states against the expected values in order to decide the failure or success of a test.

The test data used for the test runs on the system under test (SUT) is usually very complex and additionally has to be compliant with existing master data and the actual system configurations (for details see [WSS08]). Therefore we currently leverage the experience of the testers by manually providing test data. To minimize the manual effort for the test concretization, we transform generated abstract test cases in a modular way. Each test step (i.e. local state checks and message triggering) is transformed in a separate script while for each test case a master script is generated that calls the test step scripts in the appropriate order. In this way we enforce a high reuse that results in less effort and enables parallelization of the manual work, which is a big advantage for integration testing with different development areas concerned. The test execution environment is provided by the SAP Test Workbench and SAP Solution Manager. These frameworks support the whole testing process starting from the test planning, test execution until the final test reporting.

---

\(^2\)Keyword-driven testing (or action-word testing) uses action keywords in the test cases, in addition to data. Each action keyword corresponds to a fragment of a test script (the adapter code), which allows the test execution tool to translate a sequence of keywords and data values into executable tests [UL07]
Chapter 9

Results and Lessons Learnt

In the previous chapters we reported on the WP4 deployment strategy, the use of existing (Rodin, ProB) and emerging (pattern plugin) tools, as well as WP4 specific adaptations of them. In this chapter we summarise our experience and report on lessons learnt.

9.1 Appropriateness of Deployment Strategy

Clearly our deployment strategy turned out to be more ambitious than if we just targeted one application for which we made use of Event-B in a “normal” sense. Instead, our attempt to base all modelling on existing domain-specific model types and translations to Event-B imposes difficulties such as mapping back results of the model checker or theorem prover to the original modelling language. Especially interactive theorem proving turned out to be very difficult in this setting: while we succeeded to do many proofs manually, an average software developer will hardly accept to solve proofs interactively when he cannot see the relation to the original model explicitly. Instead techniques to improve automation such as heuristic invariant templates and patterns turned out to be promising techniques to increase proof automation.

On the other hand, our deployment strategy has put us in a state where developers can see pre-existing modelling content “Formal-Methods-enabled”. Our investment into the area of business applications turned out to be beneficial since we indeed could deal with multiple instances of similar choreography problems instead of singular problems. It has become clear that both points are key enablers of formal methods use in the area of business information software which might justify the required setup costs for Formal Methods use.
9.2 Suitability of Choreography Modelling as Pilot Application

The choice of choreography modelling as a pilot application has been suitable. Other modelling languages in the business application area, as for instance the Business Process Modeling Language BPMN, tend to have quite complex constructs (e.g. a huge number of gateways with different semantics) which we did not need to bother with when exploring the possibilities of Event-B using choreographies. Moreover typical realistic pairwise choreographies are of medium-size complexity so that we could simulate, check, and verify them with realistic efforts.

A disadvantage however is that the choreography modelling language MCM is not in such wide-spread use (as for instance BPMN) and the direct impacts are necessarily limited.

9.3 Suitability of Event-B for the Pilot

In general Event-B fits well to the problems we try to solve. The whole specification is divided into a number of events having preconditions and actions. This suits very well to the modelling language MCM, where the interactions also have precondition and actions. Also further specification languages we want to investigate as modelling front-ends, namely BPMN on the business process modelling level and state machine like formalisms for business objects, go well together with the event-based approach and can be translated to Event-B in a natural way.

The language Event-B is expressive enough and still not overloaded with unnecessary complex features, which allows one to write specifications in a rather natural way. Besides this, Event-B is very well supported by the software tool Rodin and multiple existing plugins.

Of course there are also problems concerned with a translation of the considered specification languages to EventB.

The first and probably the main problem is absence of a notion of refinement (which is actually one of the main paradigms of Event-B) in the specification languages we want to translate to Event-B. For example, in MCM we use refinement only to show the consistency relation between choreography and local models, but not by developing the model itself. As one of the possible ways to tackle this problem we consider the application of the pattern approach (see Chapter 6).

Another possibility to solve the problem would be to try some artificial decomposition of models into some refinement levels. In this way one also faces some problems: For example, in this case the newly introduced variables must be renamed at each refinement level and the gluing invariants between the old and
the new versions of variables must be created. This is in general a non-trivial task, which also decreases readability of the obtained models.

There are also other problems concerned with Event-B language. One problem is absence of records in the language. This leads to the necessity of modelling records by total or partial functions, which increase the complexity and decrease the readability of obtained models. Records are however a natural part of our models: the contents of a message are in fact records, and business object structures could be modelled as records as well. Work on representing records directly in Event-B has already started as part of the mathematical extensions work, but we could not make use of this in our pilot.

Another problem is that using quantifiers in Event-B leads to less automated proofs. Such quantification often occurs because in our models we have additional guards with quantification over items in the messages. Our experience shows that quantifier expressions can be better dealt with in the provers if we formulate them as set theoretic expressions. However currently we have no general recipe to translate every such expression, but have done this only for some examples. We hope to exchange the quantifiers for some set-theory constructions by the translation as far as possible, which remains at the moment a matter of future work.

9.4 Suitability of Rodin for the Pilot

Rodin is a well developed software environment having different plugins (ProB, AtelierB-Provers, Anim-B, UML-B,...) and it provides a good support for Event-B language. The tool is sufficiently well documented and has an intuitive and user-friendly interface.

In our work, we very much appreciated the open platform nature of Rodin. Our experiments have shown that it is easy to add plugins to Rodin via the Eclipse extension framework (see e.g. Chapter 7). The documentation of the APIs was sufficient in this stage of the project to implement the needed extension points, though in situations of (incompatible) changes between different Rodin versions we were happy about the explicit help from the tool builders in Deploy.

It is particularly fortunate that Rodin as well as the tools we are using for choreography modelling are both based on Eclipse. This allowed for a seamless integration of (diagrammatic) modelling and formal analysis with the Rodin platform and plugins.

A major problem in Rodin is the current state of the automated provers. In Rodin there is at the moment no easy way to add new proof-tactics, which is very important in order to automate proofs in the industrial applications. Also some parameters such as timeout cannot be changed from the user interface. This standard configuration of the provers often turned out to be dissatisfying, e.g. the
auto provers had to be started several times because the timeout was set too low.

Even though Rodin is a well-developed, stable and simple-to-use application, still some minor bugs are (as can be expected in a research project) present, which do not allow one to use the whole strength of Rodin at all times. We then reported the bugs and in a lot of cases immediate help and fixes from the Rodin developers were available.

9.5 Suitability of WP4 Adaptations

As described in this deliverable, we developed a number of methods and tools for integrating the existing modelling world with the formal approaches of Event-B.

9.5.1 Suitability of Model Simulation and Model Checks

User feedback was collected within a pilot experiment with 5 MCM users. The experiment was based on the Seller/Buyer Choreography model introduced in Chapter 2, and the real-life model in Figure 2.3. In these models we built artificial errors which violated the inconsumable message invariant (see Chapter 7) which we require the models to satisfy.

Each of the participants was familiar with one of the models, but none of them was aware of the particular inconsumable message errors. The goal for the participants was to check both models for inconsumable message errors. Before using the verification prototype, the participants were asked to search for inconsumable message errors without tool support. None of the participants was able to identify any error within the models; the errors found with the verification prototype therefore can be considered as non-trivial. Supported by the prototype described in Section 7.3, each of the participants was able to identify the inconsumable message errors within a reasonable timeframe.

The participants’ feedback was predominantly positive. The tooling was considered as simple to use and easy to learn. The integration into MCM was considered as very good. The error visualisation helped the participants to localise the error and the error description, as well as the error trace representation and animation was considered as helpful. The participants especially pointed out the interactive simulation functionality, which can also be used during modelling to gain confidence in how the choreography behaves.

Summarising, the concepts of simulation and model checking turned out as feasible and the user acceptance of the tooling was overall positive.
9.5.2 Proof Assistance Tools

Moreover we investigated the proof-assistance tools (see Section 7.4) within a pilot experiment with MCM users which had already expert knowledge in interactive proving with Rodin. The experiment was again based on the Seller/Buyer model. Each of the participants was familiar with the model and familiar with proving, but none of them was aware of the specially built-in inconsumable message error. The goal for the participants was to investigate why the proof obligation associated with the send Request event and the inconsumable message invariant for Request messages cannot be discharged automatically; this means either the proof obligation is valid, but the proof is too complicated for the automated provers, or there is an inconsumable message error in the model.

Although most of the participants presumed that the model is correct, since they were familiar with the specific model, each of the participants was able to identify the inconsumable message error, using the proof assistance tools. The tooling was considered as easy to learn and the integration into MCM and the Rodin platform was considered as good. The simplicity of the tooling was rated average; the participants argued that despite the good integration into the platform, knowledge in interactive proving is still necessary and therefore tooling for interactive proving never can be considered as simple to use. Especially the disprover was well accepted and considered as very useful. In general, the visualization helped the participants to understand the proof context.

9.5.3 Model-based Testing

The investigation of our model-based testing plugin (see Chapter 8 was based on five different use cases from a realistic application space that were intended to derive test scripts. The generated test suite had sizes ranging between 4 and 9 test cases with an average length of 10 test steps \(^1\). For the evaluation the generated abstract test cases were used to automatically generate and load test scripts into the test environment of the development teams. The participating pilot users were further provided with the MCM model and additionally with automatically generated UML message sequence charts for each test case.

The evaluation of the conducted pilots showed, that the envisioned testing approach on the base of MCM is feasible. Especially the fact that despite organisational complications the work was conducted successfully and in time led to a favouring perception. The concretisation effort for the pilot test suites was estimated to be between four and eight hours. Further the participants were able to read, understand and enhance the generated test suites with concrete test data

\(^1\)Only the triggering of messages is counted here. Considering state checks would double the number of test steps.
and message triggers, sometimes even without detailed knowledge of the tested integration. The feedback after test generation was positive. In all the cases the test generation produced reasonably small test suites. The pilot users had confidence in the quality and completeness of the tests and perceived a design-based test generation as beneficial. Although it was impossible to compare the concretisation effort with the effort of implementing the test cases by hand, all participants agreed that the evaluated approach was time-saving, due to the automatic script generation and our concept of enforcing a high reuse of generic scripts for the test steps. Also the seamless integration of our tool into SAP’s testing framework and the consequent usability of the test scripts for automatic regression testing were appreciated.

In order to certify the positive feedback regarding the reuse concept of test scripts, we decided to generate a second test suite (containing different test cases) for the fully concretized pilot. It showed that the generic reuse of concretized test steps allowed us to run this test suite after only 10 minutes of minor adaptations. This result implies that extending previously generated test suites or applying test generators with different (more complex) coverage criteria will only increase the automatic test execution but not the semi-automatic concretization effort.
Chapter 10

Outlook

Given the results and the lessons learnt, we are taking a look ahead into promising application areas. These extend the given pilot on choreography modelling in two ways.

10.1 General Directions

As shown in Figure 10.1, the modelling layers in business applications most related to choreography modelling are on the one hand side going down to the implementation via business object models and on the other hand side going up to the more abstract business process layer.

10.2 Modelling of Business Objects

For each partner business component involved in the choreography more detailed development artifacts, such as implementation code or other models exist.

A business object is1 “a set of entities with common characteristics and common behavior representing a well defined business semantic. This set is generally accepted in the business world (e.g. in an international standard or industry best practices). Examples include: Purchase Order, Sales Order, and Customer.” Business object models [KP09] describe the structure of an important business entity (such as an invoice, an order, or a delivery). Business objects are structured as a tree with a root node and several item nodes. Similar to UML class diagrams, business object models describe the relation between the different nodes, attributes of nodes and their data types, associations among nodes of the same and different business objects, etc.

Moreover, each of the nodes has a lifecycle where a sequence of user actions (e.g. create, approve, release) or service calls creates a trace through a set of statuses (e.g. created, approved, released) of the business object nodes. This lifecycle could be modelled as a state machine per node, but it also has to reflect interrelations with other nodes it is associated with, e.g. the status of an invoice could be set to approved if and only if the relevant status of all items of the invoice are set to approved.

We conduct experiments with modelling business objects as Event-B models and started to automatically translate models written in a typical in-house domain-specific modeling language to Event-B. Our goals are to (1) prove properties about the models such as “if an invoice is posted all items of the invoice are released” and to (2) prove (via refinement) that the business object conforms to a given message choreography.

For the latter case, in comparison to local partner models (LPMs) of the choreography models described above, business object models take into account the messaging behaviour w.r.t. more than one partner component and internal actions like user behaviour. With the help of a translation from these models to Event-B we can obtain a formal representation which allows us to show that they constitute a refinement of the LPM and thus preserve the properties specified in the LPM.

However, LPM and component models operate on a separate state space. Therefore we require modellers to add glue expressions to each of the states $s \in S_j$ of each LPM $L_j$. For this end we equip MCM with an expression language over the state of the business components. It is used to assign an expression $\text{glue}(s)$ to

Figure 10.1: Possible extensions of the pilot towards business process modelling and business object modelling
In our seller/buyer example, the *Reserved* state in the LPM of the Buyer may correspond to the status attribute *OfferReceived* in the buyer's purchase order business component. We will thus specify:

\[
\text{glue(Reserved)} = (\text{Status} : \text{PurchaseOrder.OfferReceived} = \text{true})
\]

Note that the glue information is also an important ingredient for deriving test cases in order to check that business components are in a right state after having received a new choreography state.

In the early stage of formalising business object models in Event-B and proving properties, the main problem was to model sequential steps appropriately. Often business object models describe situations like the following: If an action on the root node is executed, subsequently all effects on all ancestor nodes are realized. Realizing these effects on these nodes may however trigger effects on other nodes (including the root node). In other words, there is a whole chain of effects to consider. Though this chain can be computed and put into the actions of an Event-B event, the resulting model is barely readable—and difficult to prove—because of the very large events. We are currently experimenting with different modelling styles to improve readability and provability. Fortunately, even with the “bad” modelling style, the ProB model checker worked in spite of this and was very helpful to identify invariant violations.

### 10.3 Modelling of Business Processes

Business Process Management (BPM) [SRMS08] is an IT-enabled management discipline that represents a fundamental change in how businesses manage and run their processes. It requires a shift from functional based thinking to process based thinking for creating business solutions. Modelling business processes allows customers to complement their SAP solutions with company specific or local requirements. Thus customers can seamlessly balance SAP delivered best practice and own practices.

Moreover current solutions allows us to link the implementation layer with the business process layer by mapping individual process steps to implementation units which realize them in the business application. If we assume to have formal model of the business process we could prove the consistency of implementation close models (like service choreography models and business object models) with business process models.
10.3.1 Towards faithfully reflecting BPMN models in Event-B

The Business Process Modeling Notation (BPMN) is a broadly adopted graphic language for modelling business process flows and collaborations. It is currently evolving into its second version where several powerful features will be included, such as event-based sub-processes and conversation correlations among many others. These new features offer better support for modeling distributed applications based on message-centric communication, e.g., web-based services and in particular SOA systems. Besides, BPMN 2.0 also attempts to give more precise execution semantics for its modeling elements.

One of the advantages of using BPMN is the ability to model business processes at a high level leaving out implementation details, so that those high-level architectural and behavioural properties can be analysed and verified without the complication to deal with actual implementation code that may sometimes be completely irrelevant to the properties under scrutiny. Considering BPMN’s increasing importance in the development of business management software, we are interested in pursuing automated analysis practices for business process models, including formal verification and model-based testing. However, the analysis of BPMN models is considerably difficult due to the language’s rich expressiveness and high flexibility, and more critically, due to the lack of formally defined semantics.

To solve the above problems, we will give unambiguous formal semantics for an important subset of BPMN notations through an automatic translation procedure that transforms BPMN models into Event-B models. Event-B seems to be a natural and intuitive choice for modelling BPMN semantics, since both Event-B and BPMN are centred around the concepts of events. During the design of the translation procedure, we will bear in mind the following criteria: (1) resulting Event-B models should faithfully reflect the structures of original BPMN models such that the analysis result obtained on an Event-B model can be easily mapped back to its original BPMN model; (2) the refinement feature of Event-B should be utilised to achieve modularised and incremental analysis that corresponds to the object-oriented and hierarchical natures of BPMN; (3) the translation should keep resulting Event-B models simple and fine-tuned to facilitate easier applications of formal methods, e.g., increasing the number of automatically dischargeable proof obligations by automated theorem provers, or minimising the state space that needs to be explored by model checkers.

It is however quite challenging to design a translation procedure that meets all these criteria because of several significant gaps between BPMN and Event-B. One of the major challenges is to model sequential executions of BPMN models. The order in which events/activities are executed is ultimately important in BPMN. On the contrary, Event-B has no intrinsic support for sequential execu-
tions. This problem is further complicated by the potential forking and joining of control flows in BPMN. To model control flows in BPMN, we can make use of auxiliary data structures as tokens that are consumed, duplicated, and passed by events to guard their executions.

Another challenge is to model the hierarchical structures in BPMN using event refinement. A BPMN activity may contain a sub-process that consists of inner control flow objects such as activities and gateways. One idea to model such a hierarchy in Event-B is to have two machines, one refining the other, such that the inner structure and behaviour of the sub-process is modelled through a collection $E$ of events in the refining machine, while the more abstract machine contains only one event $e$ that summarises the effect of the sub-process. However, it is difficult to define the refinement relation between the summarising event $e$ and the set of events $E$ detailing the sub-process. This is because the effect of the sub-process as specified in $e$ may be achieved by a series of steps of executions of the events in $E$. As what Event-B imposes, only one out of the events in $E$ can be defined as the refining event for $e$, and all other events in $E$ are not allowed to make any modification to the data outside the sub-process, which is unavoidable in general.

There are other issues, e.g., Event-B cannot straightforwardly express “something must happen under certain conditions”, which further requires auxiliary data structures and auxiliary events to simulate the behaviour that cannot be directly modelled in Event-B. As preliminary results, we have manually transformed several BPMN models into Event-B and successfully proved some consistency properties for those models. The models for case studies contain many advanced features such as exception handling and compensations as well as some features in BPMN 2.0 like event-based sub-processes. We are also working on clarifying/eliminating semantic ambiguities in BPMN and designing the automatic translation procedure.

### 10.3.2 Ensuring Consistency of Business Processes

In [BFRR09b] we present Event-B patterns that may be used to represent recovery from time-bounded inconsistency and illustrate their use in a business process. We adapt the Time Constraint Pattern developed in Cansell et al [CMR07] to develop a pattern which adds timing information to a model of a business information system. We then develop a pattern for adding error recovery behaviour, and combine the two patterns developed. The patterns that we develop are in the style of Fürst’s approach ([Für09]), since our goal is to allow a developer to automatically apply these patterns.

We will explore further the questions around consistency and time-bounded recovery from inconsistency within business processes.
Bibliography

[ABHV08a] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. A roadmap for the rodin toolset. In Börger et al. [BBBB08], page 347.

[ABHV08b] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. A roadmap for the rodin toolset. In Börger et al. [BBBB08], page 347.


[KP09] Stefan Kätker and Susanne Patig. Model-driven development of service-oriented business application systems. In Hans Robert Hansen,


[OSG] OSGi Service Platform Release 4 Specifications.


