Abstract

Deliverable D4.2 is summarizing the work conducted in WP4 of EU-FP7 project DEPLOY for the last two years. The main objective is to report on the extended scope for the enhanced deployment and detailed descriptions of the relevant components. This includes business information sector specific deployment issues and concepts for their solution. Furthermore D4.2 contains information about remaining pilot deployment activities that were carried out after producing the preceding deliverable D4.1 [FD].
## Contents

1 Introduction ........................................... 5  
   1.1 Pilot Deployment .................................. 5  
   1.2 Enhanced Pilot Deployment ....................... 6  
   1.3 Results .......................................... 7  

2 Retrospective on D4.1 ................................. 8  
   2.1 Reported Activities ............................... 8  
   2.2 Planned Activities ............................... 9  

3 Remaining Pilot Deployment - Modeling of Business Objects 11  
   3.1 Industrial Context .............................. 12  
   3.2 Global Choreography and Local Partner Models .... 13  
   3.3 Implementation Model ........................... 15  
   3.4 Translation of Implementation Models to Event-B .... 17  
       3.4.1 Translation of model structure ............... 18  
       3.4.2 Translation of state machines ............... 19  
       3.4.3 Optimizations .............................. 21  
   3.5 Analysis of Implementation Model ................ 22  
       3.5.1 Checking Consistency Relation .............. 22  
       3.5.2 Checking Application Specific Properties .... 23  
       3.5.3 Experimental Results ........................ 24  

4 Enhanced Pilot Deployment ......................... 25  
   4.1 Introduction .................................... 25  
   4.2 Business Process Modeling ...................... 26  
       4.2.1 Formal analysis of BPMN models using Event-B .... 26  
       4.2.2 Event-B Decomposition ..................... 40  
   4.3 Scenario Testing ............................... 53  
       4.3.1 Background ................................ 53  
       4.3.2 Approach .................................. 60  
       4.3.3 Architecture .............................. 63
4.3.4 Test Model Transformation ........................................... 65
4.3.5 Test Suite Reduction .................................................. 71
4.4 Results ........................................................................ 96
  4.4.1 Appropriateness of Enhanced Deployment Strategy ...... 96
  4.4.2 Suitability of Business Process Modelling as Enhanced
       Pilot Application ...................................................... 96
  4.4.3 Suitability of Event-B for the Enhanced Pilot .......... 97
  4.4.4 Suitability of Rodin for the Enhanced Pilot ............ 97
  4.4.5 Probability of Adoption for Enhanced Pilot .......... 98

5 Evidence Consolidation .................................................. 99
  5.1 Training, transfer and human resources .................... 99
  5.2 Reuse ...................................................................... 100
  5.3 Control Impact of Formalism: DSL and hidden formal methods101
  5.4 Exploiting Models: Model-based Testing .................... 102
  5.5 Other relevant FAQ items ............................................ 102

6 Conclusion ................................................................. 104
  6.1 Summary ................................................................. 104
  6.2 Remaining Activities .................................................. 104
Chapter 1

Introduction

Business software consists of any software which helps companies improve their business. In contrast to typical areas of application of Formal Methods, this area has some, 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. As a result, customers expect high qualitative software which is efficiently developed — goals, which Formal Methods could be helpful to achieve. Nevertheless Formal Methods are (to our knowledge) not used routinely in the development of business software.

In this deliverable we are reporting on our activities towards a successful deployment of Formal Methods developed in the EU-FP7 project DEPLOY in the business sector. As there are specialised established domain-specific languages for the development of this software in place, switching to non-domain-specific languages and working with mathematical syntax is typically considered not to be feasible.

Our approach throughout the project therefore was to make business application developers benefit from formal verification techniques by hiding the mathematical formalisms, such as Event-B [Abr10], 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.

1.1 Pilot Deployment

For the initial pilot deployment, reported in deliverable D4.1 [FD], we focussed on service choreography modeling and integration testing. By providing a domain-specific modeling approach (MCM), including automatic
transformation to Event-B and automatic verification and test generation utilizing the Rodin platform, we were able to demonstrate the feasibility of hiding advanced formal verification techniques from software developers. A more detailed retrospective on D4.1 [FD] can be found in Chapter 2.

In order to demonstrate how to apply these techniques on the implementation level as well, we decided to enhance the described solution with the capability to check consistency between our domain-specific language (DSL) and the already integrated architectural artifacts, thus guiding, and driving the implementation process of business software. The results of this work is described in Chapter 3.

1.2 Enhanced Pilot Deployment

After concluding the work on the pilot for service choreography, we went into an internal evaluation. The aim was to identify the options for an enhanced deployment. After being able to convince our stakeholders about the general applicability of utilizing formal methods in the business software development process, the focus was on identifying the area with the highest potential for a broad productive use.

During the discussions it became very clear that formal techniques should be based on already adopted modeling languages to increase acceptance and lower the need for learning. Furthermore, the automatic test generation based on design artifacts turned out to be the key factor for creating interest in adopting our concepts in the development organization.

As process modeling is by far the most common and mature way of describing the overall scope of business software, we decided to concentrate on the system level for the enhanced pilot deployment instead of extending the comparatively lower abstract service layer, as reported in Chapter 4. This choice was further encouraged by the fact that system-level testing activities are in fact most decisive for business software. This is due to the fact that companies are usually able to correct inconsistent data due to faulty software, but are highly dependent on being capable to execute their standard processes.

Our aim was further to leverage not only our experience from the first piloting phase, but also concrete concepts and implementations where applicable. As described in Chapter 4.2, for the modeling of business processes we again targeted an automated model transformation and verification approach in order to hide complexity from the user, which allowed us to reuse our previous work on formal property checking (e.g. deadlock freedom), as reported in deliverable D4.1 [FD] and further enhance the approach (e.g. by
Also, on the testing side we were able to incorporate much of the previous work by continuing to utilize the Event-B transformation for creating test cases with ProB. This gave us more freedom to work on the maturity of the general testing framework and to punctually improve its building blocks. As the dependence on a specific tool provider was perceived as a major obstacle for adopting the solutions of the first piloting phase, we put our major effort into the generalization of the concepts. As reported in Chapter 4.3, we designed a phased model transformation approach that allowed us to choose between different test generators in a unified way. Furthermore, we enhanced the existing test optimization and integrated our solution into the testing framework at SAP.

1.3 Results

In the last three months we have been invited to conducting a pilot evaluation with central development units on a productive system, as described in Chapter 4.4. During this time we were able to gather a large number of new requirements for our tools, most of them related to usability, and managed to fix the ones with the highest priority. At the end of this activity, the user’s verdict was that the tool is now technically mature enough for a productive use internally. As a consequence, the management decided for a phased roll-out. This outcome is a major achievement for the team working on WP4 and a strong indicator for a successful exploitation of research results beyond the time of the DEPLOY project.
Chapter 2

Retrospective on D4.1

Deliverable D4.1 [FD] reported on the first pilot deployment for the Business Information Sector and basically summarised the work done in the first two years of the project for the business information sector led by SAP. The pilot dealt with modelling service choreographies. The approach taken was 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. This chapter briefly discusses Deliverable D4.1 [FD] by summarizing the reported work (Section 2.1) and the planned activities (Section 2.2).

2.1 Reported Activities

Modelling for business applications requires an incremental development process which supports the definition and realisation of service-based communication. It was the goal of the pilot described in Deliverable D4.1 [FD] to support such a process via Formal Methods as provided by Event-B and Rodin including

- validation of the model against the intuitive understanding of the modeller,

- verification of the consistency of the choreography model, and

- validation of the correctness of the implementation of a message choreography
Choreography Modeling  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.

In Deliverable D4.1 [FD] we introduced the Message Choreography Modeling (MCM [WRSC08]), a proprietary language developed by SAP Research for modeling service choreographies in the domain of the SAP Application Platform as well as its realisation in an MCM editor.

Choreography Verification  The modeling language MCM 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.

In D4.1 [FD] we showed 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.

Integration Testing  MCM was designed in such a way that integration test cases can be generated automatically. The aim was to enable a development process, where these test cases can be concretized with test data and other information not contained in the model and run against the implemented system in order to validate the correctness of the implementation.

In D4.1 [FD] we described the testing process and the necessary model-based testing framework we implemented based on Event-B and the Rodin platform.

2.2 Planned Activities

Modeling of Business Objects  A business object is\(^1\) “a set of entities with common characteristics and common behavior representing a well de-
fined business semantic”. 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.

We planned to conduct experiments with modelling business objects as Event-B models and to automatically translate models written in a typical in-house domain-specific modeling language to Event-B in order 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.

The description and results of these activities can be found in Chapter 3.

**Modeling 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.

The Business Process Modeling Notation (BPMN) [Obj08] 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.

Considering BPMN’s increasing importance in the development of business management software, we planned to investigate automated analysis practices for business process models, including formal verification and model-based testing. The main challenge we analysed was BPMN’s rich expressiveness and high flexibility, and even more critical, the lack of formally defined semantics.

The modeling of business process and consecutive use for model-based testing has emerged as the most promising subject for the enhanced pilot deployment. The description and results of the conducted work will be discussed in Chapter 4.
Chapter 3

Remaining Pilot Deployment - Modeling of Business Objects

The development of business applications according to the Service Oriented Architecture (SOA) principles implies a layered design and implementation. In precursory work we have shown how consistency can be verified for two layers of message choreography models: between global choreography models and local partner models [KRW09]. The work we describe here focuses on another consistency problem: the one between message choreographies (more precisely, local partner models) and their implementation models. Implementation models are not final implementation code. Even though very close to actual implementation details, they specify only the aspects relevant to the changes of internal life cycles of business objects in terms of state transition graphs. Therefore, our work is not concerned with source code analysis. Like our previous approach in [KRW09], we check consistencies through a translation into Event-B [Abr10], a formal specification language supported by the Rodin platform [ROD]. This work has been an extension of the pilot deployment at SAP, reported in deliverable D4.1 [FD].

Section 3.1 reminds about the layered development approach we observed and explains the motivation of ensuring consistency between different layers from an industrial perspective. Section 3.2 briefly summarizes the choreography modeling language MCM and its verification as described in deliverable D4.1 [FD]. Section 3.3 introduces the basic concepts of implementation models used at SAP. Section 3.4 describes the transformation of implementation models to Event-B. How these formal representations are used for enforcing consistency and application specific properties is shown in Section 3.5.
3.1 Industrial Context

Our modeling approach is based on a three-layer architecture. (1) **Global Choreography Models** (GCMs) describe a high-level view of the conversation between components. Based on labeled transition systems, they define every allowed sequence of messages as observed by a global observer. (2) **Local Partner Models** (LPMs) specify the communication-relevant behavior for each participating component. Each LPM has the same control structure as the GCM, and may have extra constraints on its local transitions. There is also a channel model (CM) describing the characteristics of the communication channels on which messages are exchanged between service components. (3) **Implementation Models** (IMs) are used as close abstractions of the final implementation code for business objects contained in local service components. They are described in terms of communicating UML state machines.

GCMs are used as a part of user requirements, and therefore we need to maintain the consistency between GCMs and IMs in order to guarantee that the implementation fulfills the requirements. There are various ways to define consistency relations between models [vG90]. Considering our application domain, we define consistency in terms of trace inclusion. We present a formal approach to keep GCMs, LPMs and IMs consistent, by stepwise checkings between adjacent abstract layers as follows.

The **consistency between GCMs and LPMs** can be enforced by two approaches [DW07]: a generative approach where consistent LPMs are generated from GCMs, or a checking approach where GCMs and LPMs are created separately and their consistency is afterward verified. While the first ensures that global and local views are always consistent, it makes changes to the local models considerably less flexible and more difficult. 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. In [KRW09] we described a mixed approach that takes best of both worlds.

The **consistency between LPMs and IMs** is the main concern of this activity. We use the Event-B specification language [Abr10] and the Rodin platform [ROD] to rigorously verify the consistency between LPMs and IMs. Consistencies are expressed in terms of event refinements in Event-B, and can be verified using either theorem proving or model checking (by ProB [LB08]).

Besides consistency, we also consider model specific properties such as absence of deadlocks and other safety properties. In order to check these properties, we either formulate them as additional invariants and prove their correctness, or express them in LTL formulas which are then validated by
the ProB model checker.

3.2 Global Choreography and Local Partner Models

A message choreography model (MCM) [WKR+09, WRS+09, WRSC08] complements the static information of communication interfaces with dynamic information on message exchanging sequences and dependencies. Due to limited space, we are unable to give a detailed description of the MCM language. We use an example model from [WKR+09] to briefly introduce the modeling elements in MCM.

Two service components, a buyer and a seller, negotiate a sales order. The buyer starts the communication by sending a Request message that will be answered with a Confirm message by the seller. The buyer afterward has the choice of either to send a Cancel message that rolls back the previous communication and allows to restart the negotiation or to send an Order message that successfully concludes the ordering process. We assume a (reliable) communication channel that is not necessarily preserving the message order. Because of this a Cancel message can be delivered after a new negotiation process already started.

Figure 3.1 shows the MCM model for the above example, which consists of one GCM, two LPMs, and a channel model. In the GCM at the top of Figure 3.1, the arrows labeled with an envelope depict the interactions Request, Confirm, Cancel, Order, and Cancel(deprecated)\(^1\) which are ordered with the help of the states Start, Request, Reserved, and Ordered. The states connected with a filled circle, i.e. Ordered and Start are so-called target. Only in these states, the communication between the partners is allowed to terminate.

The LPM of the buyer partner of our example is depicted in the lower left part of Figure 3.1. It is a structural copy of the GCM, but the interaction symbols now represent 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(deprecated) is ever sent (thus these send-events have been erased) and that a Request is sent in the Reserved state. However, due to possible message overtaking on a channel that does not guarantee to enforce the message order during transmission, receiving a deprecated Cancel is possible on the seller side. The LPM of the seller is depicted in the lower

\(^1\)Deprecated here means that the message is out-dated and no-longer relevant as the negotiation has been restarted.
MCM can be naturally translated into Event-B: interactions are simply represented as events, and the consistency between GCM and LPMs is expressed by Event-B refinement. The translation was already implemented, and also easily integrated with other tools, such as an MCM editor, thanks to the extensibility of the Eclipse-based Rodin platform. The details of the translation can be found in [WKR+09]. Here, we sketch the translation as follows.

For each transition in the GCM we generate exactly one event. For representing the states we define global status variables. 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. The global status variable is duplicated for each LPM. 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 $P(T)$, where $T$ is a set of possible message
types, denoting the set of messages being exchanged. It is initialized with \( \emptyset \). Typically, we have two partners \( P_1 \) and \( P_2 \) and two sequencing contexts (exactly once (EO) and exactly once in order (EOIO)). In that case we obtain four possible channels in the model (two in each direction).

The purpose of the verification procedure is to prove local enforceability property for choreographies. In [WRS+09] we have defined a notion of local enforceability as a trace inclusion: Traces of the local model must be a subset of traces of the global model. Trace inclusion can be proved by showing that the local model is a refinement of the corresponding global one, with the help of the translation to Event-B.

In [KR09] it is shown how to generate automatically the gluing invariants between global and local models, which are for the practical examples usually enough in order to prove the refinement relation without adding any additional invariants.

### 3.3 Implementation Model

Business objects (BO) are basic units of business data and logic, which are contained in service components. Their life cycle states can be influenced by inter-BO communication described in choreography models. In this paper, we model the life cycle of business objects as implementation models (IMs). These can be considered as refinements of their communication interfaces (i.e., local partner models), as illustrated in Figure 3.2. A business object contains a number of nodes organized in a tree-like structure. The changes made to each business node can be modeled using a UML State Diagram [OMGb].

![Figure 3.2: Refinement relations](image)

We define implementation models as model “templates” from which many model instances can be generated, which satisfy further constraints specified in templates. Due to limited space, we only give an intuitive introduction to implementation models using the example in Figure 3.3.
An implementation model contains (1) a set of node types such as **Root** and **Item** in the example; (2) one single root node type (**Root** in our example); and (3) a set of node relation types such as **Items** that associates the root with a set of **Item** nodes. Furthermore, there are two sets of constraints: (1) The first set specifies how many nodes of each type are allowed for any BO instance, which is abbreviated by the number in the up-right corner of the corresponding node type. In our example, there can be only one **Root** node per BO instance (as indicated by the number 1) and an arbitrary number of **Item** nodes per BO instance (as indicated by "n"). (2) The second constraint set specifies the multiplicity of each node relation type. In our example, the **Items** relation is a one-to-many mapping, i.e., an arbitrary number of **Items** can be associated with the root with respect to this relation.

Our variant of state machines is demanded by the great complexity of business process logic in the application context of the implementation models. Every node type has a state machine that contains a set of concurrent regions. Each region is an orthogonal part of the state machine that runs in parallel to other regions. Each region reflects the independent update of some system attribute. For example, the state machine of **Root** has two regions **reg1** and **reg2**. Each region has a set of states with one single initial state. Unlike traditional UML state machines, a transition in our variant state machine is no longer a simple pair of source and target state. First, a transition may be either **active** or **passive**. An active transition can be fired as
long as its firing condition is satisfied. On the contrary, a passive transition can only be invoked by other transitions. Active transitions are graphically denoted as solid lines, and passive transitions are dotted lines. Second, the firing condition of a transition may be very complex in the sense that it may reference states across region boundaries or even node boundaries. Moreover, the effect of a transition may enforce the firing of transitions in other regions or in other state machines.

We need the following vocabulary to reference states and transitions of other state machines. Let this refer to the current node being considered. If \( n \) is a node, then \( n.\text{parent} \) refers to the parent of \( n \). If \( f \) is a relation type, then \( n.f \) is the set of children nodes of \( n \) associated with \( n \) via the relation \( f \). Moreover, \( n.\text{reg} \) represents the region \( \text{reg} \) in \( n \), whose current state is represented by \( n.\text{reg}.\text{st} \). Finally, \( n.\text{reg}.t \) refers to the transition \( t \) in \( \text{reg} \).

A transition has a firing guard, a set of \((\text{pre}_i, s_i)\) pairs that selects the next state \( s_i \) according to a certain pre-condition \( \text{pre}_i \), and a sequence of transitions that must be invoked in order. In our example, \textbf{Start} is an active transition, whose guard is that the current state must be \( s_1 \). When it is fired, the next state is \( s_2 \), and it will enforce the transitions \textbf{Begin} in all \textbf{Item} nodes to be taken. The transition \textbf{End} is a passive transition, and can be fired only if it is invoked by the \textbf{Finish} in one of the \textbf{Item} nodes. It ends up in two possible next states: (1) If the regions \textbf{reg} in all \textbf{Item} nodes are in state \( s_{10} \), then the next state is \( s_4 \); (2) Otherwise, the next state is \( s_3 \). The firing of \textbf{End} does not enforce any other transitions to be fired. Note that since the execution of a transition may invoke executions of other transitions, there is no guarantee for termination. One has to prove that the execution of any active transition indeed terminates.

### 3.4 Translation of Implementation Models to Event-B

In this section we describe a translation of IMs (node structures and state machines) into Event-B. The translation is a challenging task because the translation should not only be sound, concise, and well-structured, but also allow as many properties to be automatically proved as possible. In Section 3.4.2 we show how the translation can be optimized in order to simplify Event-B representations and proofs and in Section 4.3.5 we compare results obtained by optimized and non-optimized translations.


3.4.1 Translation of model structure

We first show how to translate the tree structure of an implementation model into Event-B. The example in Figure 3.3 is a simplified version of the example in Figure 3.4. We consider the more complex version of the example in Figure 3.4 only in Section 3.4.1 in order to demonstrate some aspects of the translation of a tree-structure. In Section 3.4.2 and further we will continue with the simplified version in Figure 3.3.

For a node type, if it may have only one node of this type per BO instance, such as the Root node in the example, there is no need to explicitly represent this node in Event-B. Otherwise, we use an abstract carrier set to denote all its node instances, which is the case for node types Item and Info.

Given a node relation \( r \) that associates nodes of type \( t_1 \) with their children nodes of type \( t_2 \), we need to distinguish the following two cases. In the first case, there is only one instance of type \( t_1 \), say, the node \( n_1 \). If \( r \) is a one-to-one mapping, we do not need to explicitly represent \( r \) since there can be only one node of type \( t_2 \) associated with its parent \( n_1 \) through \( r \). Otherwise, if \( r \) is one-to-many, then we can represent \( r \) as the set of all children nodes of \( n_1 \) such that they are associated with \( n_1 \) through \( r \). In the second case in which there are multiple nodes of type \( t_1 \), we denote \( r \) using its inverse relation \( r^{-1} \), which maps each node of type \( t_2 \) to its parent of type \( t_1 \) such that they are related by \( r \). This is because \( r^{-1} \) is a function and results in simpler Event-B representations and proofs. When all parents of \( t_2 \)-nodes are of type \( t_1 \), \( r^{-1} \) is a total function. Otherwise, it is a partial function. Moreover, if \( r \) is one-to-one, then \( r^{-1} \) is injective. The following shows how node relations in Figure 3.4 are translated in Event-B.

variables: root_info, root_item, item_item, item_info
invariants:
\[ \begin{align*}
inv_1 &: \text{root}_{\text{info}} \in \text{Info} \\
inv_2 &: \text{root}_{\text{item}} \in \mathcal{P}(\text{Item}) \\
inv_3 &: \text{item}_{\text{item}} \in \text{root}_{\text{item}} \rightarrow \text{Item} \setminus \text{root}_{\text{item}} \\
inv_4 &: \text{item}_{\text{info}} \in \text{Info} \hookrightarrow \text{Item}
\end{align*} \]

In our example, an \textit{Item} node can be associated with a child \textit{Item} node, which we refer to as a \textit{sub-item} (Sub-items do not have further sub-items)\(^2\).

The following shows how variables denoting node relations are initialized. They are initialized non-deterministically in order to cover all possible model instances. There are also additional constraints: For example, the constraint \(\text{root}_{\text{item}}' \cap \text{dom}(\text{item}_{\text{item}}') = \emptyset\) says that the set of items related to the root is disjoint from the set of items related to other items.

\[
\text{init} \\
\hspace{1em} \text{begin} \\
\hspace{2em} \text{act}_1 : \text{item}_{\text{info}}, \text{root}_{\text{info}} :: \\
\hspace{3em} \text{item}_{\text{info}}' \in \text{Info} \rightarrow \text{Item} \land \text{root}_{\text{info}}' \in \text{Info} \land \\
\hspace{4em} \text{root}_{\text{info}}' \notin \text{dom}(\text{item}_{\text{info}}') \land \text{root}_{\text{info}}' \cup \text{dom}(\text{item}_{\text{info}}') = \text{Info} \\
\hspace{2em} \text{act}_2 : \text{root}_{\text{item}}, \text{item}_{\text{item}} :: \\
\hspace{3em} \text{root}_{\text{item}}' \in \mathcal{P}(\text{Item}) \land \text{item}_{\text{item}}' \in \text{Item} \nrightarrow \text{Item} \land \\
\hspace{4em} \text{root}_{\text{item}}' \cap \text{dom}(\text{item}_{\text{item}}') = \emptyset \land \\
\hspace{4em} \text{root}_{\text{item}}' \cup \text{dom}(\text{item}_{\text{item}}') = \text{Tasks} \\
\hspace{1em} \text{end}
\]

3.4.2 Translation of state machines

In this section we describe a translation of state machines, and show the necessity of introducing optimizations of the translation.

We define an abstract carrier set for the states of each region. For a node type that may have only one node instance, we use a variable for each region of the node to denote the current state of the region. For a node type with multiple instances, we define a function for each region that maps each node instance to the current state of the region in that particular node. As examples, the current state of \texttt{reg1} in Figure 3.3 is translated to a variable \texttt{reg1.st} \(\in\texttt{reg1.States}\); and the current state of \texttt{reg3} is a function \texttt{reg3.st} \(\in\texttt{Items} \rightarrow \texttt{reg3.States}\).

Now we show how transitions are translated. Let \(t\) be an active transition with a guard \(g\) and a set of pairs \((pre_1, s_1), \ldots, (pre_n, s_n)\) defining the next state. Furthermore, \(t\) enforces a sequence of transitions \(t_1, \ldots, t_m\) to be taken. For simplicity reason, each enforced transition \(t_i\) has a guard \texttt{true}, a set of pairs \((pre_{t_i}, s_{t_i}), \ldots, (pre_{t_{ik}}, s_{t_{ik}})\), and does not further enforce other transitions. The transition \(t\) is translated as follows:

\(^2\)Sometimes we may need sequences of item-item relations of arbitrary length. In this case, we can introduce the transitive closure, for example, we can reuse the definition of transitive closure from the Event-B mathematical toolkit [ROD].
EventT
when
  grd1 : g
then
  act1: st, st1, ..., stm := |
    ( (pre_1 ∧ st' = s_1) ∨ ... ∨ (pre_n ∧ st' = s_n) ) ∧ 
    ... 
    ( (pre_i_1 ∧ st'_i = s_i_1) ∨ ... ∨ (pre_i_k ∧ st'_i = s_i_k) ) ∧ 
    ...
end

In the above code we use st and st_1, ..., st_m to denote the current states of those regions that contain transitions t and t_1, ..., t_n, for readability reason. Note that the effects of enforced transitions are specified together with the effect of the transition enforcing them in one Event-B event. This guarantees that the enforced transitions are indeed executed. Since passive transitions can only be invoked by others, their translations are always included in the Event-B events of some active transitions.

As an example, we show how transitions in Figure 3.3 are translated. The relation Items is represented as a subset of all Item nodes since there is only one Root node.

Start
when
  grd1 : reg_1.st = s_1
then
  act1: reg_1.st := s_2 
  act2: reg_3.st := Items × {s_6}
end

Run
any node
where
  grd1 : node ∈ Items
  grd2 : reg_4.st(node) = s_7 
  grd3 : reg_3.st(node) = s_6
then
  act1 : reg4.st(node) := s_8
end

Finish
any node
where
  grd1 : node ∈ Items 
  grd2 : reg_5.st(node) = s_9 
  grd3 : reg_4.st(node) = s_8
then
  act1 : reg5.st, reg2.st : |
    (reg5.st'(node) = s_10) ∧ 
    ( (∀n ∈ Items ∧ n ≠ node ⇒ reg5.st'(n) = reg5.st(n)) ∧ 
      ( (∀n ∈ Items ⇒ reg5.st'(n) = s_10) ∧ reg2.st'(s_4) ) ∨ 
        ( (∃n ∈ Items ⇒ reg5.st'(n) ≠ s_10) ∧ reg2.st'(s_3) ) 
    )
end

Note how complex the translation can be even for a relatively simple transition such as Finish, since the effect of the transition as well as the effects of all enforced transitions must be specified within one Event-B event. In particular, all effects of Finish are specified in one Event-B action act1.
Unfortunately, act1 cannot be broken into several smaller actions, because the next state of reg2 depends on the next state of reg5 in Item nodes. Such high complexity of the translation may significantly reduce the readability and provability of the translated model. Thus, in the next section we explore two possibilities of optimizing the translation of transitions.

### 3.4.3 Optimizations

**Using implications in specifying preconditions of next states.** A transition may have several potential next states, each depending on a certain precondition. The straightforward translation specifies the choice of the next state using disjunctions (see the previous section). However, this results in complex and less readable Event-B action, and also makes automated provers to become less effective. As a solution\(^3\), we may express the dependencies between preconditions and next states using implications in Event-B guards instead of using disjunctions in Event-B actions. This can be illustrated by the optimized translation of the transition Finish in Figure 3.3 as shown below. Two new variables st5 and st2 are introduced to express the dependencies of preconditions and next states for the regions reg5 and reg2. Their values are then used in the update of the next states of the regions.

```plaintext
Finish

any node, st5, st2

where

grd1 : node ∈ Items

grd2 : reg5.st(node) = s9

grd3 : reg4.st(node) = s8

grd4 : st5 ∈ (Item → reg5.States)

grd5 : st2 ∈ reg2.States

grd6 : st5(node) = s10

grd7 : ∀n ∈ Items ∧ n ≠ node ⇒ st5(n) = reg5.st(n)

grd8 : (∀n ∈ Items ⇒ st5(n) = s10) ⇒ st2 = s4

grd9 : (∃n ∈ Items ⇒ st5(n) ≠ s10) ⇒ st2 = s3

then

act1 : reg5.st, reg2.st := st5, st2

end
```

**Using set operators.** In the above model, there are a few guards containing quantifiers (see grd7, grd8 and grd9). This may result in difficulties for automated provers to discharge proof obligations that make use of these guards as hypotheses. The reason is that the quantifiers in these guards need to be instantiated with concrete values during the proof, which requires the

\(^3\)We thank Michael Buttler for his suggestions and discussions for this optimization solution.
automated provers to make a choice in case several concrete values are available for instantiation. As a potential solution, we may consider to transform these guards into quantifier-free forms that use set operators. The advantage of using set operators is that the provers can now apply simplification rules for sets without facing choice of instantiations. As an example, the guard $\text{grd7}$ can be rewritten as below, where $\triangleleft$ is domain subtraction:

$$\text{grd7} : \{\text{node}\} \triangleleft \text{st5} = \{\text{node}\} \triangleleft \text{reg5.st}$$

In a similar manner, the guards $\text{grd8}$ and $\text{grd9}$ can be written as

$$\text{grd8} : \text{ran}(\text{st5}) = \{s_{10}\} \Rightarrow \text{st2} = s_{4}$$
$$\text{grd9} : \text{st5} \nshorteq \{s_{10}\} \nshorteq = \emptyset \Rightarrow \text{st2} = s_{3}$$

In our experiments we did witness more automatically discharged proof obligations after eliminating quantifiers by set operators. However, using set operators is not always helpful, especially when an automated prover employs such a proof strategy that translates set operators back to their quantifier-based versions. In future work we still need to assess how effective such quantifier elimination may improve the performance of automated provers, and which proof tactics should be used to better exploit the advantages of set operators. We will also design an automated procedure to translate guards to their quantifier free versions.

3.5 Analysis of Implementation Model

Using Event-B translations, we show how to check the consistency between local partner models and implementation models, and how to verify application specific properties for implementation models. We will also briefly describe how we conduct our experiments and discuss our experiences.

3.5.1 Checking Consistency Relation

The main purpose of our work is to check the consistency between message choreography models and their implementation models. As the consistency between GCM and LPMs can be verified [KR09], it suffices to show that each LPM is consistent with its implementation model. This means that all behavior of the IM can be also observed in the LPM, which corresponds to proving in Event-B that the IM is a refinement of the LPM.

In Event-B, the refinement between two machines is defined using gluing invariants that describe the relations between abstract variables and concrete
variables. In [KR09] we presented an automated gluing invariant construction method for proving consistency between GCM and LPMs. For the time being, constructing gluing invariants for the consistency between LPMs and IMs is done manually with MCM tool support, using expert knowledge of specific models. A typical gluing invariant relates the states in a local partner model to the states in the corresponding implementation models. An example would be that, if the LPM is in state $s$, then some region in a state machine in the IM must be in one of the states $s_1, \ldots, s_n$. Then, we have to prove that each transition in the IM results in a change of states that preserves the gluing invariants with respect to the change of states by the transition in the LPM which it refines.

The consistency between an LPM and its IM can be verified using either the ProB model checker [LB08] or the Atelier B provers [ROD]. The advantage of model checking is that it is fully automated. However, it suffers from the state space explosion problem. As a solution, we may set bounds for integer variables and set sizes in ProB to reduce the explored portion of the state space. This, however, cannot assure the consistency beyond the bounds that we set.

On the contrary, theorem proving does not need to explore the state space of a model. But it often requires human assistance to discharge proof obligations for large complex models. Besides, we may need to manually introduce auxiliary lemmas. For example, for the IM in Figure 3.3 we need the following additional invariant which says that if the root machine has not started then no items connected to the root has started their work.

\[(\text{Root.reg}_1.\text{st} = s_1) \Rightarrow (\forall \text{item} \in \text{Items} \Rightarrow \text{item.reg}_3.\text{st} = s_5).\]

3.5.2 Checking Application Specific Properties

We can verify application specific properties for implementation models such as deadlock freedom or other general safety and liveness properties. For example, we can check the following properties for the IM in Figure 3.3 expressing relations between states in the root node and in item nodes:

\[(\text{Root.reg}_1.\text{st} = s_1) \Rightarrow (\forall \text{item} \in \text{Items} \Rightarrow \text{item.reg}_3.\text{st} = s_5)\]
\[(\forall \text{item} \in \text{Items} \Rightarrow \text{item.reg}_5.\text{st} = s_{10}) \Rightarrow (\text{Root.reg}_2.\text{st} = s_4)\]

These properties can be expressed as invariants in the Event-B translation, and can be checked by either model checking or theorem proving. In ProB, we can also formulate and check LTL-expressible properties.
3.5.3 Experimental Results

We built an IM editor based on EMF (Eclipse Modeling Framework), and an automated translator from IMs to Event-B, in which translation optimization can be optionally enabled. The translation from LPMs to Event-B was already implemented in earlier work [WKR+09]. Using these tools, we conducted several case studies using real-life software models from the SAP ByDesign development environment. Due to confidentiality reasons, we are unable to disclose the details of the models that we use in experimentation.

We first used the ProB model checker to check both consistency and application specific properties. ProB is powerful enough to verify these models of considerable sizes, with the texts of some actions produced by the translation each spanning one or two full pages in print form. For a typical IM, its Event-B translation contains 25 events, and it took only 2 – 3 seconds for ProB to complete the checking.

Using the automated theorem provers for verification is only possible after applying the translation optimizations (see Sec. 3.4.3). The average number of proof obligations (PO) in our experiments was 150. Without optimization only a few of them could be proven automatically. After applying the optimization, 135 POs (90% of the total) were automatically discharged. We successfully proved consistency and checked certain application specific properties for all models.

The main difficulty here was the introduction of auxiliary invariants, which is an iterative process requiring expert knowledge of the models. The average number of invariants for one system was 27 with 11 type invariants, 7 gluing invariants and 9 auxiliary invariants describing the internal behavior of IMs.

One advantage of layered designs is that assuring consistency between message choreographies and implementation models can be broken down into checking consistencies between adjacent layers. As our previous work examines the adherence between global choreography models and local partner models, this part shows how to check consistencies between local partner models and implementation models through automated translations into Event-B. We have also shown that application specific properties (e.g., deadlock freedom) can be verified at the level of implementation models. Practical evaluations with real-life models show a promising applicability of our approach on the industrial development of business applications.
Chapter 4

Enhanced Pilot Deployment

4.1 Introduction

The main objective of deliverable D4.2 is to report on the extended scope for the enhanced deployment and detailed descriptions of the relevant components. After evaluation of the previous piloting activities on service choreographies, we decided to target business process modeling for the enhanced pilot deployment. In the context of the business information sector, process modeling is the prime method to capture high-level requirements, document functionality and supported usage of software products. Following the successful concept of hiding formal methods behind commonly used DSLs, we centered our work around the Business Process Modeling Language BPMN, which has emerged as the de facto standard for process modeling recently. The detailed description of our efforts can be found in Section 4.2.

Further we again targeted the test generation based on formal specification as the prime business case, which proved to be the most promising target in the previous pilot deployment. As the core concepts of model-based testing could be reused from the previous piloting phase, we concentrated on embedding them in a more generic testing framework and generalizing the existing concepts in order to lower the barriers of industrial adoption. The envisioned approach as well as the resulting work we conducted is described in Section 4.3.

In Section 4.4 we evaluate the results of the enhanced piloting activities and draw conclusions on its success and future impact.
4.2 Business Process Modeling

4.2.1 Formal analysis of BPMN models using Event-B

Complex, large-scale business information systems are critical to the successful operation of many businesses, and SAP is a leading provider of such systems. Business process modeling has become increasingly important to the development of enterprise software applications [KP09]. Nowadays, business applications are usually built by integrating a broad range of highly configurable software components and services, which can be rapidly tailored to satisfy different and constantly changing business needs. Business process models are used to describe such integration scenarios and their work flows, facilitating an intuitive common understanding of the business logic between customers and developers. In addition to their use as documentation, business process models can also be simulated, analyzed and verified to reveal design errors at an early stage in software development. This promises to enhance the efficiency of reaching high-quality software solutions and can save substantial implementation and diagnosis costs which would otherwise be incurred at later development phases.

We use formal methods to improve the quality of business process models within a software design process, and also aim to reduce the extra burden that formal methods induce on designers and developers. Within the context of the DEPLOY project, we choose the Event-B modeling formalism [Abr10] and the Rodin platform [ABHV06] in our pursuit of these goals. The choice is also encouraged by our past successful experiences of using Event-B for describing and analyzing business applications [BFRR09, BFRR10]. Event-B offers many indispensable features for analyzing business process models such as the ability to model data. The Rodin platform is empowered by a large number of plug-ins providing various analysis capabilities like specialized provers, model checking, and simulation.

We report our recent work on the formal analysis of business process models using Event-B and Rodin, and discuss the impact of the analysis results on software design and development. We also investigate the potential to largely automate these analyses in order to accelerate industrial deployment. We designed an algorithmic translation from BPMN, the de-facto standard business process modeling language, to Event-B. The translation covers most of the commonly used BPMN features, also including features newly introduced in the proposed draft of the second version of the language [OMGa]. We also make the Event-B translation structurally faithful to the original BPMN model, which not only improves readability, but also enhances provability and analyzability.
Translating BPMN to Event-B

BPMN is specified using natural and graphic languages, and comes with no rigorous semantics defined. Therefore, there are a lot of ambiguities in BPMN that had to be clarified when we designed the translation into Event-B. These clarifications are according to the specific needs of our use cases, so by no means do they offer the only proper solutions – other semantic variants can be chosen.

Similar to [BS10], the translation covers most of the commonly used BPMN features including comprehensive modeling of control flows, data modeling, compensation, message based communication, error and exception handling, sub-processes, looping and multi-instance activities. The uncovered BPMN features are most notably choreography and conversations as well as some types of flow objects, including call activities, transactions, conditional events and complex gateways. Some of these missing features are rarely used in practice and add significant complexity to the model. Other missing features such as transactions have very vague descriptions in the official BPMN specification and are difficult to interpret.

Our translation was guided by three principles. First, the Event-B translation should be structurally faithful to the original BPMN model so that anyone with knowledge of the original model can easily understand the translation. Also, any analysis result that we may obtain from the Event-B translation can be easily mapped back to the original model. Second, the translation should be designed to improve provability, i.e. it should result in the automatic discharge of as many proof obligations as possible. Finally, we are interested in verifying properties for systems that allow multiple instances of same processes.

The structure of the translation

We take the model in Figure 4.1 as an example to show how its Event-B translation is structured (Figure 4.2). This model describes the management of shift work within a factory: A worker assigned to a shift becomes unavailable, and the manager has to find a replacement from the pool of available workers. The status of each worker is maintained in a database. In this scenario, an attempt is made to automatically choose a replacement. A request is sent to an available worker, who has a fixed length of time to reply. If he accepts, he is assigned to the shift and the database is updated. Otherwise, the process may be repeated up to a maximum of five times. If, after five attempts, a replacement has not been found, a manager steps in to allocate a worker to the shift directly.
Figure 4.1: The shift worker scheduling model.
The contexts in the Event-B translation contain common definitions such as process life cycle states as well as abstract constants and carrier sets that represent process instances, message instances, data types, and so on. The translations of processes and their communication are gradually added to a series of refining machines: The machine at the first level contains nothing but the control flow information of the Factory process. In particular, it has neither data information nor the internal detail of the sub-process schedule. The machine at the second level preserves or refines all information in the first machine, and adds also the data flow information of Factory. Details of schedule and WORKER are added similarly into later refinements. In the end, the communication between the two top processes is added into the last machine.

![Diagram of Event-B translation](image)

Figure 4.2: The structure of the Event-B translation of Figure 4.1.

The above structure preserves the hierarchical structure of the original model through refinements: the information of a sub-process (e.g., schedule) is always added into machines at higher refinement levels than that of the container process (e.g., FACTORY). Our structure also achieves separation of concerns, which is very beneficial for automated provers: A property about the control flow of process Factory can be expressed and proved at the first refinement level since it needs no information from later levels. This means a smaller hypothesis space for automated provers to search.

**Processes**

We allow multiple instances of a process. We use an abstract carrier set to represent all possible instances of each process (e.g. PROC_FACTORY_INSTANCES) in contexts. The machines contain variables recording existing process instances (e.g. instances.Factory); recording the life cycle state of each existing instance (e.g. state.Factory) and, in case of a sub-process, recording the parent of each sub-process instance (e.g. parent_inner_schedule);
and recording which activity instance (outer instance) results in the creation of a sub-process instance (e.g. \texttt{outer_inner_schedule}).

**Control flow.** Our interpretation of sequential and parallel executions of flow objects uses tokens. For each sequence flow, we define a function that maps each process instance to the number of tokens in this particular process instance. Tokens are initialized when a new process instance is created by a start event: all outgoing sequence flows from the start event receive a certain number of tokens (usually 1), and all other flows receive no tokens. Each flow object is guarded by a condition stipulating how many tokens it needs to start execution.

**Control flow convergence.** With a few exceptions like join gateways, a flow object with multiple incoming flows needs only one of the incoming flows to carry enough tokens to start execution. In this case, we use as many Event-B events to represent the flow object as the number of incoming flows: each event describes the situation in which the tokens on the corresponding incoming flow are consumed. This is because otherwise we must express disjunctive choices and updates of tokens in the guard and action of the Event-B event representing the flow object. Automated provers often struggle to deal with disjunctions because they lead to case splitting and a potential explosion in the size of the proof tree. On the contrary, a join gateway requires all incoming flows to carry enough tokens to start execution. Then, it is enough to have one Event-B event to represent the gateway, which consumes tokens from all incoming flows.

**Data.** There are three kinds of data: process attributes, data stores, and activity inputs/outputs. For each process attribute, we define a function that maps each process instance to the runtime value of the attribute in that particular instance. A data store is globally accessible and does not belong to a particular process. Therefore, unlike process attributes, the data structure representing the data store involves no process information. Finally, activities may have input and output parameters. BPMN allows activities to have multiple sets of inputs or outputs. However in our translation we stipulate that any flow object or sub-process has at most one input set and one output set. We also do not explicitly represent inputs and outputs, since the runtime values of inputs/outputs are decided by process attributes or data stores.
Events triggers

An event either throws or catches a certain kind of triggers. The BPMN specification provides no information of trigger structures and how triggers are processed, stored, and discarded. In our understanding, each kind of triggers has its specific processing mechanism. For instance, the trigger of a message receiving event occurs when a desired message becomes available, and it persists until the message is consumed. On the contrary, the trigger of a conditional event occurs when a certain condition is fulfilled. However, if the conditional event is not ready to be triggered, e.g., it has no incoming tokens, then the trigger immediately disappears. Based on the above discussion, we have no explicit and unified representation for all kinds of triggers. Instead, we model the trigger behavior implicitly in their executional contexts.

Messages

Message buffers are implemented simply as sets since message order information is absent. For each type of message, we introduce two variables to record (1) the set of already sent messages of the type and (2) the set of messages still in the buffer (i.e., not yet received). Note that the buffer is shared by all process instances that may send or receive this type of messages. Sending a message is simply to add the message into both the buffer and the set of already sent messages, while receiving a message is to remove it from the buffer. Message fields are defined as functions that map each message instance to the concrete value of the corresponding field in that message.

Some message fields may contain correlation information that identifies the intended receiver which contains matching correlation information. In the model in Figure 4.1, session identifiers (sid) are used as correlation information. Each response message contains an sid field, which can be received only by a process instance with a matching sid as its process attribute. A request message is used to create a new WORKER instance. Therefore, a new request message should contain a new sid. Further detail on the translation of correlation-based message exchanges is available in [BW10].

Sub-Processes

In BPMN, a sub-process can be either collapsed or expanded, with the internal structure of the sub-process either hidden or revealed respectively. These two appearances find their analogies in the refinement hierarchy of the Event-B translation: The sub-process is first specified without internal detail when the control flow of its containing process is added. The internal detail of the sub-process is specified at later refinement levels.
For a looping sub-process, each execution creates a single outer instance, which acts as a container for multiple inner instances. The execution of an inner instance corresponds to a single loop iteration. Further detail on the translation of the “collapsed view” of the loop sub-process in the FACTORY process in Figure 4.1 can be found in [BW10].

The translation of the “expanded view” is shown below. At this level we add the outer instance attribute loop counter, and also introduce an auxiliary variable next to control the creation of the next inner instance. In our example, the loop condition is tested before each iteration, and therefore we initialize next to false to enforce the checking of the loop condition before any inner instance is created. Note that in the following code we leave out all guards and actions inherited from abstract events.

```
MACHINE Level03_Sub_schedule_CF
VARIABLES
......
at_outer_schedule_loopcounter
au_outer_schedule_next
......
EVENTS
Event act.Factory_schedule_activate \(=\)
refines act.Factory_schedule_activate
  any
    pid
    child
    where
      ... : ......
    then
      ...
      act5 : au_outer_schedule_next(child) := FALSE
      act6 : at_outer_schedule_loopcounter(child) := 0
  end
Event act.Factory_schedule_complete \(=\)
refines act.Factory_schedule_complete
  any
    pid
    child
    inners
    where
      ...
      grd6 : inners = dom(outer_inner_schedule \(\supset\) \{child\})
      grd7 : ran(inners \(\cap\) state_inner_schedule) \(\subseteq\) \{completed\}
      grd8 : at_outer_schedule_loopcounter(child) \(\geq\) max_retry
    then
      act1 : state_outer_schedule(child) := completed
      act2 : tk.Factory_schedule_gate(pid) := tk.Factory_schedule_gate(pid) + 1
  end
Event act.Factory_schedule_next \(=\)
  any
    pid
    child
```
where

\[
\text{inners}
\]

\[
\text{where...}
\]

\[
\text{grd6: inners = dom(outer_{inner}\_schedule \triangleright \{child\})}
\]

\[
\text{grd7: ran(inners < state_{inner}\_schedule) \subseteq \{completed\}}
\]

\[
\text{grd8: at_{outer}\_schedule\_loopcounter}(child) < \text{max retry}
\]

end

Event\text{ ext}_{schedule}\_start \triangleq

any

\[
\text{pid parent outer}
\]

\[
\text{where...}
\]

\[
\text{grd8: au_{outer}\_schedule\_next}(outer) = \text{TRUE}
\]

end

END

Compensation

Compensation starts with the execution of a compensation throwing event. Each throw event has a scope, and only activities within this scope can be compensated. An activity is within the scope of a compensation throw event if (1) the activity is contained in the same process as the event; or (2) the event is contained in a compensation event sub-process of the process that contains the activity.

Usually, a compensation throw event contains a reference to the activity to be compensated. However, it is left open in the official BPMN document whether all completed instances of the activity inside the scope will be compensated, or only the last instance is to be compensated. In our translation, all completed instances are compensated. An activity can be compensated only after being completed. If a compensation trigger is thrown when an activity instance is still active, the compensation handler of the activity instance is not triggered and, in this translation, will never be triggered unless another compensation trigger is thrown again in the future.

The following code shows how the shipping activity in Figure 4.3 is compensated. \text{au}_{Retailer\_shipcomp\_insts} records the activity instances which need to be compensated, and \text{au}_{Retailer\_shipcomp\_sync} is used to wait for the completion of the involved compensations before passing tokens to outgoing flows.

\textbf{MACHINE Level\_04\_Retailer\_Data}
VARIABLES
......
au_Retailer_shipcomp_sync
au_Retailer_shipcomp_insts
......

EVENTS
Event \( \text{evt}_\text{Retailer\_shipcomp\_activate} \) \( \equiv \)
extends \( \text{evt}_\text{Retailer\_shipcomp\_activate} \)

\[
\text{any} \quad \text{pid} \quad \text{to\_comp}
\]
where
\[
\text{grd}_1: \text{pid} \in \text{instances\_Retailer} \\
\text{grd}_2: \text{state\_Retailer(pid)} = \text{active} \\
\text{grd}_3: \text{tk\_Retailer\_gate\_shipcomp(pid) > } 0 \\
\text{grd}_4: \text{au\_Retailer\_shipcomp\_sync(pid)} = \text{FALSE} \\
\text{grd}_5: \text{to\_comp} \subseteq \text{instances\_ship} \\
\text{grd}_6: \text{to\_comp} = \text{dom(\text{parent\_ship}(\{\text{pid}\})) \cap \text{dom(\text{state\_ship}(\{\text{completed}\}))}}
\]
then
\[
\text{act}_1: \text{tk\_Retailer\_gate\_shipcomp(pid)} := \text{tk\_Retailer\_gate\_shipcomp(pid)} - 1 \\
\text{act}_2: \text{au\_Retailer\_shipcomp\_sync(pid)} := \text{true} \\
\text{act}_3: \text{au\_Retailer\_shipcomp\_insts(pid)} := \text{to\_comp}
\]
end

Event \( \text{evt}_\text{Retailer\_shipcomp\_complete} \) \( \equiv \)
extends \( \text{evt}_\text{Retailer\_shipcomp\_complete} \)

\[
\text{any} \quad \text{pid}
\]
where
\[
\text{grd}_1: \text{pid} \in \text{instances\_Retailer} \\
\text{grd}_2: \text{state\_Retailer(pid)} = \text{active} \\
\text{grd}_3: \text{au\_Retailer\_shipcomp\_sync(pid)} = \text{true} \\
\text{grd}_4: \text{ran(au\_Retailer\_shipcomp\_insts(pid) ◁ state\_ship)} \subseteq \{\text{compensated}\}
\]
then
\[
\text{act}_1: \text{au\_Retailer\_shipcomp\_sync(pid)} := \text{false} \\
\text{act}_2: \text{tk\_Retailer\_shipcomp\_chargecomp(pid)} := \text{tk\_Retailer\_shipcomp\_chargecomp(pid)} + 1 \\
\text{act}_3: \text{au\_Retailer\_shipcomp\_insts(pid)} := \emptyset
\]
end

Event \( \text{act}_\text{Retailer\_shipcomp} \) \( \equiv \)
refines \( \text{act}_\text{Retailer\_shipcomp} \)

\[
\text{any} \quad \text{pid} \quad \text{child}
\]
where
\[
\text{grd}_1: \text{pid} \in \text{instances\_Retailer} \\
\text{grd}_2: \text{child} \in \text{instances\_ship} \\
\text{grd}_3: \text{state\_ship(child)} = \text{completed} \\
\text{grd}_4: \text{parent\_ship(child)} = \text{pid} \\
\text{grd}_5: \text{child} \in \text{au\_Retailer\_shipcomp\_insts(pid)}
\]
then
\[
\text{act}_1: \text{state\_ship(child)} := \text{compensated} \\
\text{act}_2: \text{db\_order\_status(at\_Retailer\_order(pid))} := \text{returned}
\]
end

END
Consistency of business processes

We can use the Rodin toolset to examine the generated Event-B models for properties such as deadlock and livelock. In this section we show how we may gain further confidence in the correctness of the BPMN model by stating and proving application-level properties as invariants within the Event-B model. We use the online retailer model in Figure 4.3 as an example. The BPMN contains two extra annotations in the top right corner. These are extra application-level consistency conditions on the BPMN model. We anticipate these conditions to be defined by the developer and treated by the implementor as further constraints on the model. We show how we take account of them within the Event-B translation.

The online retailer model starts with the buyer, at which point a new instance of the process is generated. The buyer sends a purchase order to the retailer, which contains order and buyer information. The retailer ships the requested item, and the buyer is then charged. If, within a specified time period, the buyer asks to return the item, and the retailer chooses to accept the return, then both the shipping and charging activities must be compensated – shipping by the return of the item and charging by sending a refund to the buyer. The consistency of the information maintained about the order status and the buyer account must be maintained by this process.

The BPMN compensation event passes control to an associated compensating activity (Return item and Refund in our example.) The purpose of the compensating activity is to “undo” an earlier part of the workflow. A precise specification of the behaviour of this activity is usually left to a later stage in the development process.

The text annotations we investigate here, such as (1) and (2) in Figure 4.3, give the BPMN developer the opportunity to provide a more precise specification of required properties of this subsequent development. Text annotation (1) states that the order status is refunded if and only if the compensation paid is equal to the price of the item. Translating annotation (1) extends the Event-B refinement hierarchy with a new machine containing a new variable compensation and an additional invariant. The variable records the compensation paid in each instance of the retailer process. The consistency invariant introduced is formalized as

\[ \forall pid \cdot pid \in \text{instances}_{\text{Retailer}} \Rightarrow \\
\quad (\text{db_order_status}(\text{at_Retailer_order}(pid)) = \text{refunded} \Leftrightarrow \\
\quad (\text{compensation}(pid) = \text{price}(\text{at_Retailer_order}(pid)))) \]

where the order is marked as refunded only when the compensation paid is equal to the price of the goods ordered. The event generated from the
Figure 4.3: A BPMN model for an online retailer.
refund activity is also extended to record the compensation paid on that order. The new event is shown below with act4 as the additional action.

\[\text{Event } \text{act}_\text{Retailer\_chargecomp} \equiv\]
\[\text{extends } \text{act}_\text{Retailer\_chargecomp}\]
\[\begin{align*}
\text{any} \\
\text{pid} \\
\text{child} \\
\text{where} \\
\text{grd1} & : \text{pid } \in \text{instances}_\text{Retailer} \\
\text{grd2} & : \text{child } \in \text{instances}_\text{charge} \\
\text{grd3} & : \text{state}_\text{charge}(\text{child}) = \text{completed} \\
\text{grd4} & : \text{parent}_\text{charge}(\text{child}) = \text{pid} \\
\text{grd5} & : \text{child } \in \text{au}_\text{Retailer\_chargecomp\_insts}(\text{pid}) \\
\text{then} \\
\text{act1} & : \text{state}_\text{charge}(\text{child}) := \text{compensated} \\
\text{act2} & : \text{db}_\text{buyer\_account}(\text{at}_\text{Retailer\_buyer}(\text{pid})) := \text{db}_\text{buyer\_account}(\text{at}_\text{Retailer\_buyer}(\text{pid})) + \text{price}(\text{at}_\text{Retailer\_order}(\text{pid})) \\
\text{act3} & : \text{db}_\text{order\_status}(\text{at}_\text{Retailer\_order}(\text{pid})) := \text{refunded} \\
\text{act4} & : \text{compensation}(\text{pid}) := \text{price}(\text{at}_\text{Retailer\_order}(\text{pid})) \\
\text{end}\]

The second annotation in Figure 4.3 is a property over all instances of processes. The value in the account of any buyer should be the initial value of the account less any purchased items. Translating annotation (2) again adds a new machine to the model, which includes the invariant

\[\forall b \cdot \left( b \in \text{BUYERS} \Rightarrow \right.\]
\[\left. (\text{db}_\text{buyer\_account}(b) = \text{initial\_buyer\_account}(b) - \right.\]
\[\left. \text{Sum}(\text{ran}(\text{dom}(\text{at}_\text{Buyer\_buyer} \triangleright \{b\}) \triangleleft \text{at}_\text{Buyer\_order}) \right.\]
\[\left. \cap \right.\]
\[\left. \text{dom}(\text{db}_\text{order\_status} \triangleright \{\text{charged, returned}\}))\right)\]

in which the clause \text{ran}(\text{dom}(\text{at}_\text{Buyer\_buyer} \triangleright \{b\}) \triangleleft \text{at}_\text{Buyer\_order}) identifies all orders placed by buyer \(b\). This is restricted to orders with status \text{charged} or \text{returned} by the clause \text{dom}(\text{db}_\text{order\_status} \triangleright \{\text{charged, returned}\}). Order status \text{returned} identifies those orders which have been returned but not yet refunded, and therefore still need to be included in our invariant.

**Proofs.** The first property results in 28 proof obligations, of which 16 are automatically discharged. The other proof obligations require expert human intervention. The second property is considerably more complex and therefore results in 582 proof obligations, of which 300 are automatically discharged. The proving of both properties requires the discovery and use of auxiliary invariants as lemmas. For the second property, a total number of 88 additional invariants are added. Currently, we need to manually discover these lemmas. However, we observe that 30 lemmas express relations
between token quantities on different sequence flows, e.g., if the incoming flow of Charging buyer has tokens then the incoming flow of Shipping cannot have tokens. Such information can be obtained by an automated static analysis on the control flow of the model. Therefore, it is possible to automatically discover these 30 lemmas. In future work we will also investigate the possibility of discovering other kinds of lemmas. Furthermore, we observe a highly repeated pattern in the proofs that involves case splitting to distinguish process instances. Such patterns can be implemented as proof strategies customized for proving a certain class of invariants.

Enhancement of processes models using patterns

When a property is violated by a model, it is possible that the model contains undesired behavior which can be removed by further constraining the model via refinement steps. We may directly perform such steps on the Event-B translation of the model in order to verify whether such refinement steps are valid, before making changes to the original model. Moreover, refinements in Event-B can be done automatically using patterns.

Event-B patterns [BB09, HFA09, Ili08, CMR07] are a means of expressing reusable modeling structures and managing effort by promoting proof re-use. In this example, we use the type of pattern presented in [HFA09], which provides a controlled way of extending an Event-B development with a pre-validated refinement step. Since the refinement step between the abstract and the concrete pattern machines has been proved in advance, any application of the pattern results in a new, fully-proved, refinement step. The approach is automated as a plug-in for the Rodin platform ([Für09]). In the example we present below, a pattern is used to correct a previously discovered omission in a specification.

We use the shift worker scheduling process in Figure 4.1 as an example. The process depicted in Figure 4.1 contains a timing-related fault\(^1\), which can lead to an inconsistency in the data maintained by the business process. It is caused by the use of the timeout at the point where worker responses are received. It arises when a request is sent to a potential worker but no reply is received within the allotted time. Another request is therefore sent to another available candidate. He may accept and be assigned to the shift, after which an accept message is received from the first worker. Now the first worker thinks he is the replacement, but in fact the second has been chosen. We discovered this error using the Rodin model checker ProB: we added an

\(^{1}\)Note that the shift worker scheduling process is a simplified version of a BPMN workflow proposal, and not a part of any real world system.
invariant expressing the property that at most one worker accepts the shift at any given time. ProB found an erroneous execution within a short time.

When translated into Event-B, the flaw present in the described scenario can be corrected using the timed error recovery pattern, shown in Figure 4.4 and presented in full in [BFRR10]. It is designed to be applied to any model in which late messages are not properly processed. When applied, a further refinement level is added to the Event-B development. This new level contains the error recovery behavior which ensures adequate processing of any late messages.

Figure 4.4: Structure of the timed error recovery pattern.

The concrete machine in the pattern separates normal and recovery behavior by distinguishing the receipt of messages before and after the deadline and handling these two cases separately. Late responses are followed with a compensation event, which may be further refined depending on the way in which recovery is implemented.

Applying the pattern requires the identification of the activities in the workflow where the timer is set and the (on time or late) replies are received. These activities are then matched with the sending and receiving events in the pattern abstract machine (snd and rcv in the abstract machine in Figure 4.4). The pattern variables must also be matched to the appropriate variables within the development.

In the Shift Worker Scheduling model in Figure 4.1, the timer is set at the task Select an available worker. The Receive response action is the point at which messages are received. The application of the pattern introduces a new event corresponding to rcv_bad (the arrival of late replies) and given below.

\[
\text{Event} \quad \text{act\_schedule\_response\_late} \equiv \text{any}
\]
\( \text{pat}_m \)

**where**

\[
\begin{align*}
\text{grd1} & : \text{pat}_m \in q_{\text{rcv}} \\
\text{grd2} & : \text{tt}(\text{pat}_m) < \text{now}
\end{align*}
\]

**then**

\[
\begin{align*}
\text{act1} & : q_{\text{rcv}} := q_{\text{rcv}} \setminus \{\text{pat}_m\} \\
\text{act2} & : q_{\text{comp}} := q_{\text{comp}} \cup \{\text{pat}_m\} \\
\text{act3} & : \text{timercvd} := \text{timercvd} \cup \{\text{pat}_m \mapsto \text{now}\}
\end{align*}
\]

**end**

In this event, \( \text{pat}_m \) is the message and \( q_{\text{rcv}} \) and \( q_{\text{comp}} \) are the messages queued for reception and compensation respectively. The second guard requires that the current time (\( \text{now} \)) is later than the target arrival time of the message (\( \text{tt}(\text{pat}_m) \)). On arrival, the message moves to the queue for compensation and the time at which it is received is recorded.

The **recover** event refines the **rcv** event. As well as retaining all the functionality of **rcv**, it places the compensated message in the database of consistent messages. The precise nature of the compensation activity required will vary according to the particular activity it is compensating, so the **recover** event acts as a placeholder for a fuller description of compensation within the workflow, which may be added (perhaps by the application of a more specific pattern) in further refinements.

The ability to automatically add pre-validated refinement steps to generated Event-B models can be used to support BPMN development. In our example, the refinement step made to the Event-B translation can be re-constructed in the original model by introducing a parallel thread to detect and react to late messages. Such reconstruction can be achieved either through a reverse translation procedure from Event-B back to BPMN, or by building up a repository of BPMN refinement patterns corresponding to Event-B patterns. We will explore both possibilities in future work.

### 4.2.2 Event-B Decomposition

Business software systems are often large collections of services and components that work together to fulfill business needs. Therefore, business process models are usually large, complex, but can be broken down to smaller modules and components, while their integration scenarios can be modeled atop smaller individual component models. Hence model decomposition is paramount in mastering complexity of large models. Although there has never been a lack of theoretical research in this area, few modelling platforms provide a sufficiently good level of tool support to evaluate structuring techniques and their impact on industrial application of formal modelling. Moreover, there are no clear guidelines for model designers which often impedes the application of formal verification to large-scale designs.
The basic Event-B language supports model structuring by nothing more than the use of events and refinement inside one model. However, it is highly impractical to construct a large scale formal design as one monolithic model, which results in numerous problems with legibility, maintainability, team work, reuse, and so on. Notably, it also affects proof structuring: autonomous provers are suffocated by a large number of hypotheses and thus anything that makes the context of a proof smaller is extremely beneficial.

Model decomposition comes as a great potential to solve the above problem. Three different decomposition styles exist for Event-B, all providing a guarantee of refinement monotonicity: the model after decomposition is a correct refinement of the one before decomposition, provided all obligated proofs are given. This allows a decomposed part of the model to be treated as an independent artefact so that the modeller can concentrate on this part and does not have to worry about the other parts. The tool supports for these techniques, realized as Rodin plugins, not only provide assistance on how to decompose a model, but also generate explicit constraints and relations between the decomposed model and the resulting sub-components.

Technically, decomposition is a special form of refinement by which a single abstract model is refined by several concrete models and their aggregation. Like refinement, a properly planned decomposition step results in very low proof cost. It requires, however, a good degree of foresight. First, a model is difficult to decompose if the model does not possess a structure that can be easily divided and mapped to independent components to be produced by decomposition. Second, any careless design decision before decomposition may be difficult to correct afterward, which often leads to complete rework on the development of each individual local component. Therefore, a modeller can serve a better job when some guidelines are available so that the modeller knows which decomposition style to choose from, and which design decisions need to be taken care of at each stage.

We provide some insights and propose modelling guidelines for applying model decomposition, drawn from our personal experiences and illustrated by a case study by an industrial user of Event-B. We introduce stages in decomposition and point out which design problems (that can be easily overlooked) need to be addressed in each step.

**Decomposition**

The top-down style of development used in Event-B allows the introduction of new events and data-refinement of variables during refinement steps. A consequence of this development style is an increasing complexity of the refinement process when dealing with many events and state variables. Decomposition
addresses such difficulty by providing a mechanism for splitting a large model into several sub-models (that can be further developed independently). Several decomposition techniques have been proposed by extending the existing Event-B notation. We consider the following three existing approaches: shared-variable [Abr09], shared-event [But09] and modularisation [ITL+10b], all of which are supported by Rodin plug-ins [SPHB11, Mod]. These decomposition techniques differ in that different model elements are shared among sub-components. For shared-variable decomposition, a part of state information (variables) is shared among sub-components. Further refinements then concentrate on how each component processes shared state information. For shared-event decomposition, a set of events are synchronised and shared by sub-components. Hence, it is important to take care of the inputs/outputs of these synchronised events. Modularisation defines a set of interfaces that are shared and accessed by different components. Interfaces provide callable operations and promises that these operations can deliver. The implementation of an operation should guarantee that the promises are fulfilled for any given circumstance.

The primary challenge of applying decomposition is to ensure that the structure of the original model fits the requirements of the chosen decomposition style, leading to helpful sub-models that can be developed separately with a tangible advantage in terms of proof efforts and overall model scale. As with any top-down approach for system development using refinement, the more abstract models are initially, the more useful the decomposition step will be. Here we do not focus on directly justifying the use of a particular decomposition style. Instead we focus on how to proceed when decomposing using one of the suggested decomposition styles.

We define a general top-down guideline for the three decomposition techniques based on the following common template.

**Stage 1** To model the system abstractly, expressing all the relevant global system properties.

**Stage 2** To refine the abstract model to fit the structure expected by a given decomposition technique.

**Stage 3** To apply decomposition.

**Stage 4** To develop the resulting sub-models independently.

Following this guideline, global properties are captured early in the model and guaranteed to hold in the final models by combining refinement and decomposition. The development of each decomposed part is done independently of the others. Consequently, we can have different implementations
for a decomposed model that is guaranteed to work with any implementation of other decomposed models.

In the remainder of this section, we elaborate on the application of different decomposition techniques using our proposed modelling guideline.

**Case Study: A Master Data Updating System**

Fig. 4.5 shows a master data updating system that we use as our case study. The system consists of a **User** process and a **Server** process keeping some master data in sync. When **User** proposes a data change, it first updates its local copy, and then sends a request message to **Server** and waits for the answer. Upon receiving the request, **Server** checks the validity of the proposed change, and updates the master copy only when the change is deemed valid. Then, **Server** sends to **User** a response containing either an approval or a rejection. **User** has to roll back the change if a rejection is received. We are interested in the global property that the local and master databases are always identical before and after each data update procedure.

The Event-B model in Fig. 4.6 serves as the most abstract view of the above system. The initial model is designed to have as few variables and events as sufficient to express the above mentioned property (see invariant `inv0_3`). The model contains variables `udb` and `sdb` denoting **User** and **Server**’s database respectively. The Boolean variable `is` denotes if the global system is in sync. There are three events, namely `u_update`, `s_update` and `u_final`. When the system is in sync, `u_update` changes the local database, which invalidates the insynch status. While the system is out of sync, `s_update` may occur to update the server database (either to be the same as `udb` or unchanged). Finally, `u_final` occurs to put the system back in sync by making the local database to be identical as the server database. Note that `s_update` may be skipped when the system is out of sync. Although this cannot happen in the real system, we permit it in the abstract model to simplify proofs, which does not affect the satisfaction of the global property in consideration. This spurious behavior will be removed by refinement and decomposition in the further developments.

**Shared-Variable Decomposition**

We describe in detail how to develop the example of update master data using shared-variable decomposition. The model in Figure 4.6 represents our abstract model of **Stage 1**. We continue with the subsequent stages of our modelling guideline.
Figure 4.5: The Process Model of Update Master Data
**Stage 2. Shared Channels Between Components.** In this preparation stage, we introduce the channels (the shared elements) acting in between User and Server. The channels are modelled by two variables `creq` and `cres`, corresponding to the set of messages going through the request and response channels respectively.

Moreover, since in Stage 4 the shared variables and external events can be *neither removed nor refined*, the shared elements introduced in this preparation stage must be *concrete*.

Furthermore, we are going to split the variables and events into two groups, corresponding to each site, preparing for the later decomposition step. An important design constraint here is that User’s events can only reference the variables belonging to User and the shared channels, but not the variables of Server. The same for Server’s events.

As a result, variable `is` must be refined away. We replace `is` by `uis`, the local in-synch flag, with a gluing invariant `uis = is`, i.e. the global in synch is consistent with the User’s view. A separated in-synch flag `sis` is introduced for Server. Moreover, in order to separate User and Server completely, we introduce new variables for keeping some information belong to each site. For User, variable `udb_old` is added in order to keep the old value of User’s database for undoing later if necessary. For Server, variable `sc` keeps the user’s change to the database on the server site for updating Server’s database if the change is valid.

Despite of the details that we have to introduce in order to clearly separate the future sub-components, we aim to keep the model at this stage fairly abstract. It should contain only necessary information for maintaining the global properties and specifying the shared elements between future sub-components. Other information, e.g. control/data flows within each sub-component can/should be abstracted away.

For example, we assume for the moment that the update of the database
and sending the request message from User happens simultaneously. This is represented by event \texttt{u_update_and_req}, a refinement of the abstract event \texttt{u_update}.

\begin{verbatim}
u_update_and_req refines u_update
    any ch where
    uis = T \land ch \in CH
    then
    uis, udb, udb_old, creq := F, upd(udb \mapsto ch), udb, \{ch\}
end
\end{verbatim}

In \texttt{u_update_and_req}, the local database \texttt{udb} is updated to be \texttt{upd(udb \mapsto ch)}, the new value obtained by applying changes \texttt{ch}; the old local database is saved in \texttt{udb_old}; and the actual change is send as a request to the server via channel \texttt{creq}. Note that \texttt{u_update_and_req} satisfies our Memo SV3, i.e. reference only variables belonging to User and the shared channel \texttt{creq}.

Other events in this model include: \texttt{s_receive_req} for Server to receive some request; \texttt{s_accept_res} for Server to update its database and send a positive response; \texttt{s_reject_res} for Server to send a negative response without updating its database; \texttt{u_receive_res_acc} and \texttt{u_receive_res_rej} for User to receive some (positive/negative) response and act accordingly.

An important advantage during the model design in this stage is the use of the abstract model from Stage 1. Consistency enforced by refinement guides our design in Stage 2, i.e. constraints on the shared variables will be derived from the need to maintain the global properties introduced in Stage 1 (typically in terms of invariants).

In our example, the following invariants are discovered during the process of discharging proof obligations such as guard strengthening and invariant preservation of the model. They relate the content of the channels and the internal status of User and Server. Invariants \texttt{inv1.7} and \texttt{inv1.8} relate Server’s database \texttt{sdb} with the User’s database (current \texttt{udb} or old \texttt{udb_old}) depending on the content of the channel \texttt{cres}. Invariants \texttt{inv1.9} and \texttt{inv1.10} state that User is out of synch if there are some request or response messages.

\begin{verbatim}
inv1.7 : cres = \{T\} \Rightarrow sdb = udb
inv1.8 : cres = \{F\} \Rightarrow sdb = udb_old
inv1.9 : creq \neq \emptyset \Rightarrow uis = F
inv1.10 : cres \neq \emptyset \Rightarrow uis = F
\end{verbatim}

\textbf{Stage 3. Decomposition Summary.} This stage is semi-automatic: we provide the tool with input on how the events are partitioned into different future sub-models. Intuitively, we separate our events into two groups, corresponding to User and Server accordingly. The variables distribution amongst
these models are calculated according to the information about events distribution. The summary of our decomposed models is as follows.

<table>
<thead>
<tr>
<th></th>
<th>User</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internal events</strong></td>
<td>u_update_and_req, u_receive_res_acc, u_receive_res_rej</td>
<td>s_receive_req, s_accept_res, s_reject_res</td>
</tr>
<tr>
<td><strong>Private variables</strong></td>
<td>udb, udb_old, uis</td>
<td>sdb, sc, sis</td>
</tr>
<tr>
<td><strong>Shared variables</strong></td>
<td>creq, cres</td>
<td>creq, cres</td>
</tr>
</tbody>
</table>

**Stage 4. Developments of Sub-models.** We present a summary of the additional refinement steps for each model. Most invariants in the sub-models are technical and related to the sequentialisation of the actions, reflecting the process flows in Figure 4.5.

**User** The control flow is introduced via means of a program counter \( upc \) to capture the actual sequential steps inside the **User** process. Other internal variables of **User** are introduced accordingly, i.e. **User**’s change \( uch \) and **User**’s stored response type \( ures \).

**Server** Similarly, the control flow of **Server** is introduced via means of a program counter \( spc \). Internal variables of **Server**, such as the check result \( scr \), are introduced.

**Shared-Event Decomposition**

Using the same initial model in Fig. 4.6, we describe the following stages in the application of a shared event decomposition. The system is designed to be decomposed into components **User** and **Server** synchronously communicating by value passing messages.

**Stage 2. The Value Passing Protocol.** The goal of this stage is to have a model where the state variables are partitioned amongst the future sub-models. Typically, this stage involves refinement of events to introduce the shared elements in the form of events’ parameters. Similar to the shared-variable style, a good abstract model is pursued where only necessary information related to the global properties and the shared elements are specified.

In this refinement, we prepare the decomposition by introducing synchronous channels and respective value passing protocol. The content of the protocol is represented by shared parameters (in the resulting sub-events). At this stage, the communication is abstract and occurs in a single event.
The global flag \( is \) is replaced by \( uis \) and \( sis \) for User and Server sync respectively. The gluing invariants between \( uis \), \( sis \) and \( is \) are given by \( inv1_1, inv1_2 \) and \( inv1_3 \): \( uis \) always matches \( is \); while a request is being processed, \( sis \) matches \( is \); otherwise, the server is synchronised (\( sis = T \)).

\[
\text{variables: } udb, sdb, uis, \quad inv1_1 \quad uis = is \\
u_ch, u_rq \quad inv1_2 \quad u_rq = PRC \Rightarrow sis = is \\
sis, s_st, s_ch \quad inv1_3 \quad u_rq \neq PRC \Rightarrow sis = T
\]

Some control variables are added: \( u_ch \) corresponds to the User change; \( u_rq \) holds the request state on the User side where \( PRC \) corresponds to the processing state; \( s_st \) corresponds to the server state and \( s_ch \) holds the change from the Server’s viewpoint. Refined event \( u_update \) models a modification that is stored in \( u_ch \) before being sent to the server by the new event \( rq \). Event \( rq \) simultaneously sends the request from User and receives it in the server. Then the request is stored in \( s_ch \) and User (\( u_rq := PRC \)) and Server (\( s_st := VAL_RQ \)) states are updated. The server is considered out of sync once receives a request (\( sis := F \)).

\[
\text{Note that event } rq \text{ has been designed so that it can be syntactically split into parts concerning only with variables of the User or Server (Memo SE1). The request validation can be deferred until the decomposition because it is irrelevant to the considered global property. The server is updated in the refined event } s_update \text{ for a valid request. Even when the request is deemed invalid, a response is sent back by the new event } rsp. \text{ This event syncs in the Server and updates the User’s request. If the request is valid, } u_ch \text{ is applied}
\]
locally; if the request is invalid, \( udb \) remains the same. In either case, \( udb \) is back in sync with the server. Several gluing invariants are discovered as a result of the generated proof obligations. inv1.4 state that while \( u_rq \) is processed, \textbf{User/Server} changes match; if \( u_rq \) is deemed invalid, \( sdb/udb \) are identical (inv1.5); a valid request results in \( sdb \) matching with \( udb \) updated with \( u_ch \) (inv1.6).

\begin{align*}
\text{inv1.4} : \quad &uis = F \land u_rq = \text{PRC} \Rightarrow s_{ch} = u_{ch} \\
\text{inv1.5} : \quad &u_rq = \text{INVLD} \Rightarrow udb = sdb \\
\text{inv1.6} : \quad &u_rq = \text{VLD} \Rightarrow \text{upd}(udb \mapsto u_{ch}) = sdb
\end{align*}

Our model is synchronous since the messages exchanged by \textbf{User} and \textbf{Server} are sent and received simultaneously. Alternatively, we could also model asynchronous communication by introducing a \textit{buffer} between \( udb \) and \( sdb \) suggesting a three way decomposition.

\textbf{Stage 3. Decomposition Summary.} Sub-models \textbf{User} and \textbf{Server} result from the allocation of the original variables according to their use. The decomposition is summarised in the following table:

<table>
<thead>
<tr>
<th>Variables</th>
<th>User</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>( udb, u_{ch}, uis, u_rq )</td>
<td>( sdb, s_{st}, sis, s_{ch} )</td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td>( u_{update}, u_{final}, rq, rsp )</td>
<td>( s_{update}, rq, rsp )</td>
</tr>
</tbody>
</table>

\textbf{Stage 4. Developments of Sub-models.} The decomposition allows the separation of sending/receiving a request by defining the request as parameter \( msg \) shared by \textbf{User} and \textbf{Server} (similarly applied to the server’s response). The resulting sub-models can be refined independently:

\textbf{User} A program counter is added defining the local states (update \( udb \), send request, receive response, commit/discard change). Two events refined the two possible outcomes: \texttt{l_commit} for valid modifications updating \( udb \) and \texttt{l_discard} to discard the modification. An additional refinement could add a request queue removing the waiting between the server reply and the next modification.

\textbf{Server} The server is refined by modelling the request validation with a new event \( s_{val} \).
Modularisation

Unlike the two other decomposition approaches, modularisation offers a greater degree of flexibility on how to construct a decomposed specification because of lower coherence among sub-components. One consequence is that the modularisation approach applies to both top-down and bottom-up designs, and even blurs the boundary. For top-down development, the previously introduced guideline can be used in modularisation approaches as well. In bottom-up development, sub-component interface may already exist before a global integration scenario is designed. This is useful for many industrial use cases in which service integrations and customizations are formally analyzed.

We take a different approach here that does not follow the above mentioned guideline. Even though we will explain the approach in a top-down fashion, it resembles certain aspects of a bottom-up development in that the design of sub-component interfaces is relatively independent of the global integration, because the interfaces are standard communication interfaces that produce and consume messages.

We start with an abstract machine that only specifies message flows and does not state the global property. Interfaces are defined for the two processes and implemented by separate machines by adding local control and data flow information. A sufficient amount of implementation details, such as local variables and properties, is carefully chosen and exposed in their interfaces to enable the verification of the global property. Finally, the top abstract machine is refined by adding details of operation calls and message buffers. The global property is then verified on a final refinement.

Stage 1. Specifying Abstract Message Flows. The top abstract machine uses flags to indicate whether a certain message has been sent or received. For example, if the flag req_snt is T then a request message has been sent. Several invariants describing the order of message events are added for property verification later. As an example, inv6 specifies that a response message has to be sent before it can be received. Message events are abstract at this level and simply set the flags accordingly.

variables: req_snt, req_rcv, res_snt, res_rcv

\[ \text{inv6 : } \text{res_snt} = \text{F} \Rightarrow \text{res_rcv} = \text{F} \]

\[ \text{send_req} \triangleq \text{when } \text{req_snt} = \text{F} \land \text{res_rcv} = \text{F} \text{ then } \text{req_snt} = \text{T} \text{ end} \]

Stage 2. Process Interfaces. The interface of each process defines a set of message operations. These operations do not consider how messages are transported, but merely specify the types of messages that they provide or expect. For example, interface User provides messages to be sent
(get_request), and take incoming messages fed to them for local consumption (put_response). In particular, get_request produces a message equivalent to the local change \((ch)\) proposed by the user.

\[
\begin{align*}
g_{\text{req}} &\text{ pre} & \neg req_{snt} &\land \neg res_{rcv} &\land \neg res_{rcv} &\land \neg res_{rcv} = \text{F} \\
&\text{return} & msg &\land \neg res_{rcv} &\land \neg res_{rcv} &\land \neg res_{rcv} = \text{F} \\
&\text{post} & msg' = ch &\land req_{snt}' = \text{T} &\land req_{snt}' = \text{T} &\land req_{snt}' = \text{T} \\
&\text{end} &
\end{align*}
\]

\[
\begin{align*}
\text{puter} &\text{ any} & msg &\in &\text{BOOL} &\land \neg req_{snt} &\land \neg req_{snt} &\land \neg req_{snt} = \text{T} \\
&\text{post} & \neg res_{rcv} &\land \neg res_{rcv} &\land \neg res_{rcv} = \text{F} \\
&\text{post} & res_{rcv}' &\land res_{rcv}' &\land res_{rcv}' = \text{F} &\land res_{rcv}' = \text{T} &\land res_{rcv}' = \text{T} \\
&\text{end} &
\end{align*}
\]

**Stage 3. Process Implementations.** Concrete details of sub-components are added in implementations, such as events that describe how control states and local variables are updated. Message-related flags are no longer present, thus we provide a link between those flags and local control states \((u_{cs})\) as gluing invariants like the one below among others.

\[
\text{inv18} \quad \neg req_{snt} = \text{F} \iff u_{cs} \in \{\text{start}, \text{upd}, req\}
\]

An interface implementation is essentially an Event-B refinement step. We need to prove that the postcondition of any interface operation must be fulfilled by the corresponding events that implement the operation. We also need to prove relative deadlock freedom that, whenever an interface operation is enabled, some of its implementing events must be executable.

**Stage 4. Final Global Machine.** The top-level machine is refined at this level to contain operation calls and actual message exchanging behavior. Each process has a buffer to store incoming messages. When a message needs to be sent, the corresponding interface operation of the sender process is called to retrieve the outgoing message, which is then added to the corresponding buffer. When a message is to be received, the message is taken out of the buffer and passed to the receiver process by calling the respective interface operation. In the following code, the prefix \texttt{user} is used in interface variables and operations of the \texttt{user} module.

\[
s_{\text{send req}} = \text{when } u_{\text{req snt}} = \text{F} \land u_{\text{res rcv}} = \text{F} \text{ then } bu_{s} = \text{bu}_{s} \cup \{u_{\text{get request}}\} \text{ end}
\]

51
Stage 5. Property Verification. Unlike the other approaches, the global property is encoded and proved at the final global machine. The proof is based on the symbolic values of the local and master databases delivered as operation post-conditions in the process interfaces.

Discussions

All approaches decompose global machines into two components, one per process\(^2\). Shared-variable and shared-event approaches start with similar abstractions, specifying a minimal set of events reflecting how the local and remote databases are updated while preserving the global property of interest. A series of refinements are introduced with appropriated chosen gluing invariants and proofs of deadlock freedom and convergence. The two approaches are different in that the shared-event version implements a synchronous message passing model. The choice is mostly motivated by the fact that synchronised message passing events can be shared by two processes. However, the system could be modelled in an asynchronous communication by introducing a *buffer* sub-component. These two approaches ensure that the global properties are preserved before decomposition. Afterwards each individual sub-component focus on their specific properties. A system involving a shared object is favoured by a shared variable decomposition where the shared object can be accessed by all the sub-components. On the other hand, communicating system parts can be shared event decomposed possibly introducing a middleware to allow an asynchronous approach.

In contrast to the other approaches, the modularisation version formulates and proves the global property at the final stage, after the module interfaces are designed. This is possible because the global machines and modules are loosely coupled, only linked by interfaces. An advantage that immediately comes to mind is flexibility: changes to existing components and inclusion of new components do not necessarily affect unchanged modules nor their proofs. However, the largest challenge of modularisation is the design of appropriate interfaces as they play a crucial role in linking multiple worlds while preserving the global property. During the design of our model, we go through iterations of “trial and error” to find the appropriate amount of information to be exposed in interfaces. As we learned from our experiences, a good practice is to start with an initial interface containing a minimal amount of information and weak possible post-conditions for operations. When these are insufficient to prove the global property, we can gradually bring more information to the interface and strengthen post-conditions.

\(^2\)The models are available online at [http://eprints.ecs.soton.ac.uk/22164/](http://eprints.ecs.soton.ac.uk/22164/)
4.3 Scenario Testing

4.3.1 Background

While unit testing in SOA applications is usually well researched [BBMP08, TCP+05, OX04, BD07] and consistently deployed in practice, other testing activities pose several new challenges [CP09, BHB+07]. The difficulties to be overcome are due to the heterogeneity, high distributivity, dynamicity, and loose coupling of the service-based systems. These complexity properties are taking their toll on the testing process. A model-driven approach to SOA integration helps to address such challenges, as it allows for a general solution, applying state-of-the-art tools and techniques for formal reasoning about service models [BD07] and model-based testing (MBT) [UL07, BDG+08].

To further exploit our activities on business process modeling, for the enhanced pilot deployment we decided to continue our efforts in model-based testing. While we focused on service integration testing in the previous pilot deployment phase, we now investigated into scenario testing based on business processes. Scenario testing is usually conducted on the user interface. Therefore, we started working on model-based GUI testing, which is an interesting but little covered research subject, especially considering the market relevance of enterprise software. In Section 4.3.1 we describe that the necessary application of the SOA paradigms for enterprise software systems implies that black-box testing techniques have to be used. In Section 4.3.1 we explain the relevance of scenario testing for SOA applications at system level. In Section 4.3.1 we lay out that current test automation in the industry only takes place for the test execution and that any applicable testing approach for enterprise software has to be based on the system’s user interface. In Section 4.3.1 we mention some related work on applying MBT on GUI systems. Section 4.3.1 describes our proceeding regarding the enhanced pilot deployment.

Current SOA Testing Challenges

Since the most modern enterprise systems have SOA as the underlying paradigm, we must first understand the challenges in the development and testing of these systems.

When new programming paradigms, such as the ones associated with SOA, are emerging, it naturally arouses the question whether there is a need for new testing methods or whether existing approaches can be adapted. More concrete, it has to be determined whether approaches and techniques, developed for traditional monolithic systems, distributed systems, component-
based systems, and web applications can be adapted to service-based systems. In order to provide an answer, the particularities and challenges of SOA testing have to be analyzed.

The authors of [CP09] identified the following key distinguishing factors for SOA that generate unique challenges for the testing activities:

- **Lack of code access.** For users, services are just interfaces as they neither have knowledge about the structure of the code nor the possibility to observe its execution. These limitations are preventing any form of white-box testing for users.

- **Dynamicty and adaptiveness.** For traditional systems, one is always able to determine at least the set of possible targets for a call [MRR05]. This is not true for SOA where the work flow of abstract services might be bound to concrete services retrieved from one or more registries during execution.

- **Lack of control.** Services are deployed independently and might be updated without informing the consumers. Therefore service can unexpectedly change their behavior or miss service level agreements.

- **Lack of trust.** As decisions for choosing a certain service solely depend on information that the provider is publishing. Therefore there exists the risk that potential users are negatively influenced by providing them with incorrect or inaccurate information.

- **Cost of testing.** As test invocation by users may cause cost or other undesirable effects (e.g. an experience of a denial-of-service attack) on the provider side, extensive or repeated testing might not be feasible.

A slightly different approach has been taken in [GGN09], where the challenges of SOA testing are clustered to the items **Stakeholder separation, Service integration, Service versioning and migration** and **Service binding and reconfiguration**. However, this regrouping neither leaves out nor introduces additional aspects.

Considering the challenges derived from [CP09], some general implications can be derived. It seems reasonable that the first three items - namely **lack of code access, dynamicty and adaptiveness, and lack of control** - are dealing with technical challenges while the following items - **lack of trust and cost of testing** - may have to be solved on the management level.

For addressing the management challenges group, it may be necessary to provide means of interaction between stakeholders, in order to share information and rights. Whether these means of interaction have to provide
for anonymity, as it is presumed in [GGN09] will be seen. However, the conservatism of major business companies and the consequent request for knowing and trusting business partners will probably lead to different solutions [O’L00].

The technical challenges group clearly requires that black-box testing techniques should be applied, as code access in a SOA environment is limited. Dynamicity and adaptiveness itself can only be achieved by providing detailed information about interfaces. Otherwise (in-)compatibilities cannot be detected and handled automatically. Therefore it can be assumed that MBT will be have a much greater impact in the testing process than in traditional industrial development setups, where modeling is carried out rather sporadically. The lack of control over parts of the system implies that tests will have to be carried out not only in the development of a service-based application, but also regularly after deployment. Therefore having automatic regression tests in place is a natural conclusion [AKK+05]. Moreover, the GUI testing might be in many case the only possibility of testing the functional and non-functional properties of a service-based system.

The Enterprise SOA Testing Stack

After having discussed the general challenges of SOA testing, a closer look will be taken on the particular testing activities, in order to see which of them is affected in what way. In [UL07], the commonly used layered testing approach for component-based systems (CBS) has been described. As the general idea of partitioning applications into logical units is somehow similar to the SOA approach of encapsulating related functional units in a service, the definition of SOA testing layers can be done analogous to the one for CBSs. Consequently four distinct testing layers, illustrated in Figure 4.7, can be distinguished [WS09]. In the following each layer is shortly introduced.

Unit Testing Unit testing is the best understood testing layer in research and practice. In contrast to all other testing layers, unit testing focuses on getting confidence in the functional correctness and hence in the correct implementation of the algorithms. As mentioned above, it deals with single software units in isolation. During unit testing the execution context of the software unit under test is mocked. Therefore, it can be carried out in SOA systems just like in any CBS implementation or any other software architecture, using all available tools and techniques.

Service Testing Also service testing for SOA is analog to the testing of components in the CBS world to some extend. The general focus of ser-
vice testing is less on the correct implementation of algorithms but on the integration of the functional units inside the (service) component and on the fulfillment of the contractual obligations of the component’s interfaces. This is also conforming to the definition of testing layers [ABR+07], were it is argued that everything apart from unit testing is a form of integration testing.

**Integration Testing**  As mentioned before, the loose coupling of service components is one of the distinguishing factors of SOA. In contrast to the CBS approach, integration testing cannot rely on homogeneous components with tightly connected interfaces. Instead, the adaptability and distribution of SOA demands additional considerations for integration testing. Especially the effects of message racing and its implications have to be considered during system development and should be tested thoroughly [WS09]. Message racing in this context refers to situations where messages are not received in the same order as they are sent.

**System Testing**  In the SOA world, system testing can be defined analog to the classical definition for CBS, as comprising the fully integrated application, usually using its externally exposed interfaces. As the faultless interplay of the services can be assured on the integration testing level, in practice system testing is based on high-level usage scenarios and business requirements that have been defined by business analysts or customers. UI-based testing is therefore most appropriate to carry out the tests, as the system should be validated as a whole and only using access points that are available to the

---

**Figure 4.7: SOA testing layers as presented in [WS09]**
A Glimpse in the State of the Practice

In the past, a significant effort has been put by the software industry to increase the efficiency of testing. Test automation in this context has been regarded as the most promising way. State of the practice of test automation for enterprise software is the automation of the test execution process, while the test design (the definition of abstract test cases and their concretization) is in general still a manual task.

At SAP, the transformation from abstract test cases to executable test scripts usually follows the keyword-driven testing principles. 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].

The introduced keywords are for example realized on top of SAP’s eCATT test script language [HLT07]. To allow the highest possible reuse, they are oriented on the structure of the enterprise system itself, which is organized in so-called transactions. A transaction offers a set of methods to alter an enterprise’s internal data in a consistent way. As most of the offered data modifications do not demand extensive computation, the transaction’s functional logic is usually realized by the user interface and therefore can not be
tested separately.

Therefore, implementing testing keywords is mostly done by utilizing capture/replay functionality, which is provided by most of the test automation tools. These tools are monitoring user interactions on the interface and producing a test script that can reproduce the execution of the recorded sequence of events. SAP’s Test Workbench can be used to capture eCATT scripts, which further enables to give them more flexibility by exchanging concrete values with variables that can be initialized independently through the interface of the script. For complex transactions, a captured script can further be broken down to multiple scripts with lower complexity.

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 [WSS08]. Therefore, the current practice is to leverage the experience of the testers, who are asked to provide appropriate test data.

SAP further provides a tool called Test Data Migration Server (TDMS), which is able to derive consistent reference data from existing systems. It is also quite common that reference test data is provided by customers or internal departments, as additional information to the requirement specification. If these data samples are available, testers are able to choose the appropriate input for each test case from that source.

At SAP the test execution is controlled by the Test Workbench, where test plans (consisting of multiple test suites) are executed automatically and periodically in the case of regression tests. The results of the test runs are centrally reported, including different coverage criteria based on source code, model elements or requirements. Figure 4.8 shows, how eCATT automates the test execution, by having a global test scripts calling the referenced keyword scripts of each test step. The results of each test step are transferred to the next script using exporting and importing functions.

A Glimpse in the State of the Art on MBT for GUIs

Model-based testing has recently received a lot of attention both in the academic as well as the industrial communities, with a couple of books published [UL07, BDG+08, JVCS08] and several dedicated workshops (A-MOST, MBT, MOTES, MOTIP etc.). Moreover, there is already a market with several MBT commercial tools (like Qtronic from Conformiq, TestDesigner from Smartesting, and SpecExplorer from Microsoft). However, most of the tools and methods work for functional testing during the software development phase and very few methods are available for the system level testing and especially UI testing. Some reasons for that are the lack of precise models
at the UI level based on the business processes, the large number of possible states and user events in modern GUIs and the UI dynamicity and runtime adaptation. One of the most complete reference on the research on MBT for GUI systems is [Mem07]. This paper proposes a general event-flow model for GUI systems that can be used on the existing work of testcase generation, test oracle creation and regression testing. The work of [VLH+06] proposes annotated UML activity diagrams as modeling environment for GUIs. On the industrial side of research, [CSH03] showed the difficulties of using the AGEDIS tools on a real GUI testing systems, whereas [PFTV05] shows how to use the Microsoft tool SpecExplorer for GUI testing.

Planned Activities

As we have seen in the previous two subsections, there is a need for research in order to successfully apply MBT to GUIs in an industrial setting. Figure 4.10 depicts the envisioned system testing approach that defines the research plan. As explained in Section 4.3.1, system testing is carried out when the whole system (or at least major parts of it) is developed and test ready. In the following the individual steps of such approach are described. The development of the system or the adjustment of an already existing solution is left out of this description. It should happen previous or in parallel with the second step.

1. Members of consulting, key customers and development architects are deriving business process models for a new product or feature or customer implementation according to the market’s or customer requirements and based on SAP’s expertise on industrial best practices. Such processes can be modeled using the Business Process Modeling (BPM) product of SAP NetWeaver and then the UI elements and their connections can be modeled with the Visual Composer tool - see Figure 4.9 for two screenshots.

2. The created content, which effectively describes the usage scenarios of the new functionality is used to generate test model skeletons. By utilizing model transformation techniques this step should be automatic.

3. The test models are afterwards enhanced by test engineers, such that they reflect previously defined test goals and acknowledge the specifics of the concrete software architecture.

4. From the test models, abstract test suites are derived automatically, using model-based testing techniques.
5. The abstract test suites are optimized according to industrial best practices (e.g. minimizing test case length while preserving test coverage). After further concretizations, the optimized suite is executed automatically on the user interface of the system under test.

4.3.2 Approach

In the previous subsection we tried to motivate and draw some guiding lines of research in the area of system testing for enterprise systems, with a focus on GUI testing. As a consequence we focussed at improving the state of the art and state of the practice by using MBT techniques. This work leveraged the existing work in GUI testing and MBT and tried to find solutions that overcome the new challenges deriving from the new SOA paradigm. As described above, a key element in this endeavor was the business process modeling that has been described in Section 4.2. In the following we will describe how this model content can be leveraged for deriving scenario tests
For enterprise applications the most important objective of testing is to gain confidence over the executability of the supported business processes. While functional errors may only affect a specific department of a customer (even a workaround may be possible), the general inability to run a business process will most likely have severe impact. Therefore system testing is the predominant quality assurance activity in enterprise application development. In practice, system testing is based on high-level usage scenarios and business requirements that have been defined by business analysts or customers. UI-based testing is most appropriate to carry out the tests, as the system should be validated as a whole, only using access points that are available to the prospect user.

Keyword-based testing for UI is mostly done by utilizing capture/replay functionality, which is provided by standard test automation tools. These tools are monitoring user interactions on the interface that can reproduce the execution of the recorded sequence of events. These captured scripts commonly allow data flexibility by exchanging the concrete values (used during capturing) with variables that can be initialized independently. Further, the recorded scripts can be combined in so-called scenarios. A scenario is a sequence of recorded scripts working on a predefined set of data. Usually global variables are used in scenarios to organize the data flow. Their function is to store output values of a captured script and make it available as input for another. In Figure 4.11 an example of the described data flow is
given. The value of the local output variable \textit{A.out} of \textit{Script A} is written to the global variable \textit{X} and later mapped to the local input variable \textit{B.in} of \textit{Script B}.

![Figure 4.11: Data flow in a scenario.](image)

The testing approach we introduce here leverages a keyword-driven testing framework as described above. For details please refer to [WS10]. On top we implemented a model editor to facilitate automated test generation. According to the taxonomy of Utting et.al. [UPL06], the model which is in the center of the MBT approach describes environment behavior, i.e. the users interaction with the system in contrast to the systems implemented routines. In more detail, the Test Model (TM) is an activity-based model, containing start and end points and a number of activities. An activity corresponds to a step in the business process that should be supported by the SUT. Each activity is linked to a recorded script and contains additional information necessary for test generation:

- data-flow information as described above
- a guard condition to specify constraints for the execution based on global variables
- a side effect for modifying global variables in case of activity execution

Various other information, which is necessary for the execution of generated test cases (e.g. user roles) can be defined as well. However, as they are not relevant to the test generation itself, they will not be discussed here. For the convenience of the test designers, our model editor further supports conditional branching elements and hubs. In Section 4.3.4 they will be described in more details.

In order to be independent of a specific tool vendor, we tried to integrate a general test generation approach. In contrast to our activity-based
modeling language, most test generators rely on abstract state machine notations similar to ASM, B or Z [Bör10, Abr96, SA89]. In more detail, test generators like SpecExplorer, Conformiq, and the ProB-Test-Generator [VCG+08, Hui07, WKR+09] rely on input languages that include a general definition of the state space and actions, consisting of preconditions and effects. Preconditions restrict the state space in which an action is enabled. Effects describe how the current state is modified when the action is executed.

The following subsections assemble the main building blocks of our solution. Section 4.3.3 gives insights into the design decisions that we took in order to support a generic testing approach as sketched above. Section 4.3.4 describes different kinds of transformations from an activity diagram representing the behavior of the SUT into an (abstract) state transition machine (STM), which is used as input for a common test generator. Following the test generation, a unified test suite reduction is applied to the output, as described in Section 4.3.5. This is necessary in order to get similar result regardless of the chosen test generator, which do not necessarily have similar optimization criteria. Afterwards the test cases are mapped back to the activity-based TM. This is necessary to guide the generation of executable scenarios (one per test case) and further allows for visualization of the test cases in the TM.

An aspect, which we will not consider in detail is test data provisioning for enterprise systems [WSS08]. As mentioned, a scenario uses a predefined set of data. In the case of TM we usually consider a pool of possible data sets and utilize the test generator to find a suitable data configuration for each generated test case. The identified data will be used to concretize the generated scenarios in order to enable direct execution without further test concretization effort.

4.3.3 Architecture

In this section we give details on our architecture which is supporting the utilization of common MBT tools in a proprietary test environment. In Figure 4.12 the main blocks of this architecture are presented. As described in the previous sections the Test Model Editor allows the creation and editing of activity-based test models, triggering test generation and visualization of the resulting test suite. It is embedded into a proprietary Test Environment. The Test Environment offers UI-based keyword-driven testing capabilities through a Scenario Editor, which allows to assemble captured test scripts and to visualize the generated executable scenarios (obtained from test cases). The scripts can be recorded through the Script Recorder component, which is connected to the SUT for this purpose. Besides the capturing of user
interactions on the SUT, the *Script Recorder* offers replay functionality, which is also utilized for the stepwise execution of scenarios.

Figure 4.12: Architecture of the MBT environment.

TM can be converted into an STM according to given coverage criteria as described in Section 4.3.4. These model transformations are implemented on the client side, i.e. in the *Test Environment* and utilizes an intermediate format for STMs. This intermediate format was defined general enough, such that it can be transformed to the concrete STM input languages in the web-services hosting the different test generators. A proxy is set up for routing the test generation requests in order to obtain a single communication partner, which allows to add and update generator components without additional configuration of the test environment.

General-purpose MBT tools rely on varying strategies to reduce the large initial test suites they produce during test generation. Therefore we decided to offer unified test suite optimization independent of the chosen test generator. This further allows us to consider custom requirements for the enterprise software domain, as described in [WKR+09]. The different optimization procedures are wrapped in another set of web-services and can be used in the following way. After the test generation succeeded, the resulting traces are transformed into an intermediate test suite format and sent to the proxy component, which forwards it to an appropriate *Suite Optimizer*.

The test reduction is implemented on the intermediate format for test suites. Therefore a further transformation of the results in the *Suite Optimizer* is not necessary. The proxy takes the reduced test suite and routes it back to the *Test Model Editor* where it will be used to create the concrete
test suite, containing executable scenarios.

Because the test environment is relying on a detached backend repository for the storage of test artifacts, we were able to implement the Test Model Editor as a web-based tool, running in standard browsers. Having such a browser-based modeling environment and a service-based test generation brings the following advantages:

- **Usability**: Utilizing a generic input and output format, we are able to hide the complexity of the specific model transformations into the input format of concrete test generators, thus making MBT accessible as a service to other test environments.

- **Re-usability**: Having a browser-based front end also allows promises a light-weight integration of the Test Model Editor into other testing environments as long as it can be connected via back-end services.

- **Performance**: Decoupling computational expensive functionality like test generation and test suite reduction promises better system performance and does not block front-end users. Replication of the web-services and the introduction of load balancing to the proxy further increases scalability.

- **Maintainance**: The service-based decoupling in combination with a proxy further allows to maintain and upgrade test generation components in a non-persuasive way.

### 4.3.4 Test Model Transformation

As explained in the previous subsection, the main motivation for transforming an activity-based TM into a state-based STM is to enable the usage of common MBT tools for test generation. In this section we describe different transformation strategies from a TM into a STM. This becomes necessary, because we want to leverage standard action coverage of STM-based test generators for various coverage goals on activity-based TM. As defined in Section 4.3.2 the term action refers to abstract transitions, i.e. defined on abstract states. Our concrete implementation is based on SAP’s proprietary modeling languages [KP09]. Because we cannot introduce them here, we will describe the core concepts of our transformations utilizing a pseudo modeling language instead.

The example TM depicted in Figure 4.13 will be used for illustrating the various transformation descriptions in this section. The activity diagram describes a sales order process. The diagram contains start and end
Figure 4.13: Activity diagram of the Sales Order example

points, activities (rounded rectangles) and a hub element (triangle) as well as connecting flows (arrows). The hub element is a graphical shortcut for a connection of all input elements with all output elements. The given example is therefore equivalent to a TM without hub element but with 4 flows that are pairwise connecting the Create Sales Order and Create Sales Order from Template activities with the Release Sales Order and Approve Sales Order activities. Further, diamond-shaped conditional elements are supported by our editor, which can be used similar to hubs. However they allow to specify guards on each outgoing flow. Their transformation will be discussed in a separate example.

The behavior described in the example should be interpreted in the following way: The SUT should enable two ways of creating a sales order, either from scratch or by utilizing a template. After the sales order is created, the user should be able to choose either releasing the sales order or explicitly approving it. While released sales order with a quantity > 10 must be approved after releasing it, for the others this activity is optional.

When generating tests based on TM we speak about activity-coverage of the TM, if the resulting test suite covers every activity of the diagram at least in one scenario. Analogous, flow-coverage means that the test suite covers all flows of the diagram at least once. As the hub element is a graphical shortcut only, we attempted to resolve it during test transformation, but the resulting suite appeared to contain redundancy for users. Therefore we speak about extended-flow-coverage, if each flow of the TM with resolved hub elements is covered at least once. For the given example, the minimal test suite for activity-coverage contains 2 tests, while the suite for extended-flow-coverage contains at least 4 tests.

As described above, our approach assumes that each test generator supports action-coverage of an STM, which means that generated test suites visit each reachable action in at least one test case. In order to utilize such a test generation algorithm for the 3 defined coverage criteria for TM, we illustrate three different translations from activity diagrams to STMs using
the example from Figure 4.13.

**Activity-Coverage** The transformation, which is necessary to utilize action-coverage on the resulting STM to simulate activity-coverage on the TM works as follows. We define a state type \( t_s \) containing the elements \( \text{start} \) and \( \text{end} \) as well as \( s_1..s_n \), where \( n \) is the number of activities in TM. Further we define a variable \( \text{state} \) of type \( t_s \) and initialize it with \( \text{start} \). Each activity is translated into one action, which sets \( \text{state} \) to one distinct value of \( t_s \). The precondition of each action is expressed as a guard over \( \text{state} \) and enforces that the action is only executable, if \( \text{state} \) holds the value of an associated enabling activity. Further an \( \text{End} \) action is introduced, which is enabled in states related to final activities in TM. The actions \text{Approve Sales Order} and \text{End} have additional guard conditions: \( \text{Quantity} \leq 10 \) and \( \text{Quantity} < 10 \).

![Diagram of activity-coverage](image)

Figure 4.14: Translation of Sales Order example in order to use the activity-coverage criteria

Figure 4.14 illustrates the result of this transformation for the given TM. For example the \text{Release Sales Order} activity has been translated to an action with identical label, which is enabled when \( \text{state} \) is either assigned to \( s_1 \) (indicating that \text{Create Sales Order} has been executed) or \( s_2 \) (indicating preceding execution of \text{Create Sales Order from Template}) and results in assigning \( \text{state} \) to \( s_3 \).

As each activity of the TM is translated to exactly one action in STM, action-coverage of for STM results into the desired activity-coverage of TM. For the STM in Figure 4.14 the obtained test suite is given below:

2. Create Sales Order from Template, Approve Sales Order.

**Flow-Coverage** Figure 4.15 illustrates the result of the transformation of the example using the flow-coverage criteria. Like for activity-coverage,
we define a state type $t_s$ containing the elements $start$, $end$, $s_1$..$s_n$, but in this case the actions correspond to the arrows (flows) and the states to the modeling elements of the diagram in Figure 4.13. For example, the action $Start$ To $Create$ corresponds to the arrow from $start$ to $Create$ Sales Order and the states $s_1$ and $s_3$ correspond to the activity $Create$ Sales Order and the $Hub$-element.

Figure 4.15: Translation of Sales Order example in order to use the flow-coverage criteria

The obtained action-coverage in STM therefore simulates the flow-coverage of the original TM. The test cases can be translated back to the TM using the fact that each action relates to a flow in TM connecting two activities or an activity and $start$ or $end$. For the STM in Figure 4.15 the obtained test suite is given below:

2. Create Sales Order from Template, Approve Sales Order.
3. Create Sales Order, Release Sales Order, Approve Sales Order.

Extended-Flow-Coverage Figure 4.16 illustrates the result of the transformation of the example using the extended-flow-coverage criteria. The translation is similar to the one proposed for the flow-coverage criteria. The actions in STM also correspond to the arrows (flows) and the states to the modeling elements of TM, but in this case the hub-elements are resolved and the obtained arrows are translated instead. As shown in Figure 4.16 this results in six states instead of the seven states for flow-coverage in Figure 4.15.

The hub-element is not relevant for the test suite of TM, because it does not relate to any executable activity. This means that the translation of
Figure 4.16: Translation of Sales Order example in order to use the extended-flow-coverage criteria

generated test cases back to the TM is similar to the one used in the case of flow-coverage. For the STM in Figure 4.16 the obtained test suite is given below:

2. Create Sales Order, Approve Sales Order.
3. Create Sales Order from Template, Approve Sales Order.
4. Create Sales Order from Template, Release Sales Order.
5. Create Sales Order, Release Sales Order, Approve Sales Order.

Note that it is possible to use flow-coverage or even extended-flow-coverage transformations in order to obtain a test suite which realizes activity-coverage. However, for minimizing test effort the suite must then be reduced after test case generation in order to eliminate residual test cases. As described before, test suite optimization usually has to be performed anyway (e.g. when lots of test cases are generated because of TMs containing loops in combination with guards) but tend to introduce performance issues in larger models. Further, also the test generation itself is slower for flow-coverage compared to action-coverage.

Conditional Element. In the remainder of this section we will discuss the translation of the conditional element. Let us therefore consider another example, depicted in Figure 4.17. It contains two conditional elements, C1 and
Figure 4.17: Conditional elements of activity diagrams

Figure 4.18: Resolving conditional elements

$C_2$, both having two output flows with the corresponding guard conditions $g_i$.

For calculating activity-coverage the conditional elements could be resolved similar to hub elements. The resulting STM fragment for the given example can be seen in Figure 4.18. However we do not advice such an approach, because deriving traces of generated test cases on the original TM (e.g. for highlighting) is much easier when preserving the original structure of the model. Further, the debugging of the resulting STM becomes much more convenient. From a theoretical standpoint, also for extended-flow-coverage the conditional elements should be resolved similar to hub elements. However in practice conditional elements are seldomly used as graphical shortcuts but for explicitly visualizing a conditional choice after the execution of an activity. Therefore our users do not expect that extended-flow-coverage might result in a smaller test suite than flow-coverage.

As a consequence we decided that our transformations preserve each conditional element. This means that for conditional elements the translation rule is the same for activity-coverage, flow-coverage and extended-flow-coverage. Basically, for each branch of the conditional element a separate action and an additional element for the state type is created in the STM.
Figure 4.19: Translation of conditional elements

The guard of the action contains the condition of the associated branch in addition to the description of enabling states. As depicted in Figure 4.19, the branch to activity $B$ at the conditional element $C_2$ in Figure 4.17 is translated to action $C_2a$. It is enabled, if $g_3$ is evaluated true and state is assigned to $c_1b$, the state element assigned with the branch of conditional element $C_1$, which leads to $C_2$. After execution of action $C_2a$, state is set to the associated state element $c_2a$.

### 4.3.5 Test Suite Reduction

Test suite reduction is an activity which reduces test suites while maintaining its coverage properties. This problem is equivalent to the set covering problem and therefore NP-complete. Many strategies for solving the problem are known, e.g. the Greedy or Branch and Bound approaches. These algorithms focus on the minimization of the number of action calls within a given test suite. Balancing the test case distribution is another objective which should be considered when choosing an efficient test suite. This work introduces and evaluates different extensions of the standard techniques which incorporate test case distribution. We show that these adjusted strategies compute a reduced test suite with a smoother distribution within an acceptable amount of additional time in comparison to the classic algorithms.

Most MBT approaches are running in two phases. In the first phase vast amount of test cases are generated for an inserted model until a coverage of model entities is achieved. In the second phase a subset of these test cases is selected with the aim to preserve the targeted coverage and therefore the assumed fault-uncovering capabilities [RHvRH02]. This activity is called test suite reduction. The problem of test suite reduction can be formulated like follows [HUGS93]:

71
Given: A test suite $TS$, a set of test case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired testing coverage of the program, and subsets of $TS$, $T_1, T_2, \ldots, T_n$, one associated with each of the $r_i$’s such that any one of the test cases $tc_j$ belonging to $T_i$ can be used to test $r_i$.

Problem: Find a representative set of test cases from $TS$ that satisfies all of the $r_i$’s.

The test suite reduction problem can be considered as a hitting set problem — the problem of finding the hitting set having minimum cardinality, which is equivalent to the set cover problem and is known to be NP-complete \cite{GJ79}. The standard way of solving the hitting set problem is a restatement into a 0/1-Integer linear program. Afterwards this can either be exactly solved by using a Branch and Bound algorithm or approximately by applying different variations of Greedy heuristics \cite{Vaz01, Chv79}.

The problem of test suite reduction is largely discussed in the literature. There are papers, where the general test suite reduction activity is described \cite{CL96, OPV95}. In \cite{BMK04, YH07} there are approaches using multi-objective optimization functions, whereas in \cite{MEF99} an approach based on genetic algorithms is introduced. Some empirical results for test suite reductions have been reported in \cite{RHvRH02}.

In this section we propose modifications of the standard algorithms Branch and Bound and Greedy in order to achieve a smoother test case distribution over the test suite, which is grounded on concrete requirements of SAP MBT users. The experimental results presented in this subsection demonstrate the applicability and efficiency of the proposed approaches.

Test Suite Reduction

Most MBT-tools are using abstract transition state machines as input \cite{UL07}. Generally they can be described as tuples $(\mathcal{S}, \mathcal{A}, s_0, s_e)$, where $\mathcal{S}$ is a space of states with some initial state $s_0$ and some end state $s_e$ and $\mathcal{A}$ is a finite set of actions. Each action $a$ can only be run if the current state $s$ satisfies some condition $\text{cond}_a(s)$ and modifies the current state if run, i.e. $s \mapsto \text{mod}_a(s)$. In this context a test case $tc$ is a sequence of actions $(a_1, \ldots, a_k)$, such that the following conditions hold: If we define

$$s_i = \text{mod}_{a_i}(s_{i-1}) \quad \text{for } i = 1, \ldots, k; \quad (4.1)$$

with $s_0$ being the initial state, then $\text{cond}_{a_i}(s_{i-1})$ is true for all $i$. Additionally $s_k$ shall be the end state $s_e$. Obviously the \textit{length} of $tc$ is $k$. We will denote it as $|tc|$.

Additionally there is a finite set of requirements $\mathcal{R} = \{r_1, \ldots, r_n\}$. All requirements have to be met by a complete test suite. Each test case $tc$ either
satisfies a given requirement \((r_i(tc))\) or does not, that is \(r_i(tc) \in \{true, false\}\). For convenience we also define

\[
cov(tc) = \{ r_i: r_i(tc) = T, 1 \leq i \leq n \}.
\]  

(4.2)

Now a test suite \(TS = \{tc_1, \ldots, tc_m\}\) is complete, if \(R = \bigcup_{i=1}^m \text{cov}(tc_i)\) and the test suite reduction problem can be reformulated as follows:

**Given:** A test suite \(TS\) and a set of requirements \(R\), such that \(TS\) is complete with respect to these requirements.

**Problem:** Find a complete test suite \(TS_0 \subseteq TS\) that is minimal with respect to

\[
\text{value}(TS_0) = \sum_{tc \in TS_0} |tc|.
\]  

(4.3)

In the remainder of this section we present two classical approaches, the *Greedy-* and the *Branch and Bound-*algorithm, to solve this problem. Later we will describe how these algorithms can be modified in order to obtain a better test case distribution.

**Greedy Algorithm**  
Even though the Greedy algorithm computes an approximation, [Chv79] showed that the result cannot become arbitrarily bad. In fact the upper bound for the error only depends on the number of requirements.

The algorithm stores two objects: An iteratively constructed subset \(TS_0\) of \(TS\), which will be a complete test suite after termination of the algorithm and a set \(R_0\) of all those requirements that are already met by this subset (l. 1–2). While not all requirements are met (l. 3), the algorithm does the following: It computes the set of all test cases \(tc\), for which the ratio \(w_{tc}\) of the number of additionally satisfied requirements and the number of additional action calls is maximal (l. 4–6). Then it picks one of these at random (l. 7). This test case is afterwards added to \(TS_0\) and \(R_0\) is updated appropriately (l. 8–9).

**Input:** test suite \(TS = \{tc_1, \ldots, tc_m\}\), set of requirements \(R\)

**Output:** approximated minimal test suite \(TS_0 \subseteq TS\) which is complete

1: \(TS_0 = \emptyset\)
2: \(R_0 = \emptyset\)
3: **while** \(|R_0| < |R|\) **do**
4: **for** \(tc \in TS \setminus TS_0\) **do**
5: \(w_{tc} = \frac{1}{|tc|} : |\text{cov}(tc) \setminus R_0|\)
6: \(TS_0' = \{tc \in TS \setminus TS_0 : w_{tc} \text{ is maximal}\}\)
7: Pick \(tc \in TS_0'\)
Algorithm 1. Greedy Branch and Bound — Algorithm

We use the Branch and Bound variation (Balas-algorithm) described in [J.W03] to compute an optimal result. The algorithm identifies all possible subsets of $TS = \{tc_1, \ldots, tc_m\}$ with arrays $(n_1, \ldots, n_m)$. Here $n_i = 1$ means, that $tc_i$ is part of the subset, while $n_i = 0$ means, that it is not. To check these arrays systematically, they are organized as a binary tree. At the root node no decisions have been made, as any node on level $i$ represents a certain choice of the first $i$ bits. For simplicity it is also denoted as array $(n_1, \ldots, n_i)$ and identified with the test suite $\{tc_j: n_j = 1\}$. We denote the level of a node by level$(n_1, \ldots, n_i) = i$.

Now (4.3) can be extended to nodes by

$$\text{value}(n_1, \ldots, n_i) = \sum_{j=1}^{i} |tc_j| \cdot n_j. \quad (4.4)$$

The Branch and Bound algorithm stores two objects: The best solution found so far, $n_{\text{res}}$, and a stack $S$ of nodes that has to be checked. Obviously $n_{\text{res}}$ is initialized with the array that represents whole $TS$ and $S$ with the stack that only contains the root node (l. 1–2). As long as additional nodes have to be checked, one of them is popped from $S$ (l. 3–4). Then child nodes $n_0$ and $n_1$ are generated, where $n_0$ rejects the next test case and $n_1$ includes it. To decide, whether it is necessary to check these as well, the following rules are applied:

- $n_0$ is expanded if it has a successor which represents a complete test suite (l. 7) and if appending the smallest remaining test case results in a suite that is smaller than the one represented by $n_{\text{res}}$ (l. 8). (To be able to efficiently evaluate the second condition, we presume TS to be sorted ascending by length.)

- $n_1$ is expanded if it is smaller than $n_{\text{res}}$ (l. 11), but not if it is complete (l. 12, 15).

Finally $n_{\text{res}}$ is updated, if $n_1$ is smaller than $n_{\text{res}}$ and complete. In this case all nodes bigger than $n_1$ are removed from $S$ afterwards (l. 11–14).

As already stated earlier, the complexity of this algorithm is exponential in test suite size. Therefore it is reasonable to reduce this size by removing
redundant test cases in advance. Hereby we call a test case \( tc \) redundant, if there exists a smaller test case \( tc' \) as well, that satisfies all requirements fulfilled by \( tc \). Those redundant test cases usually occur due to loops in the test model. The removal can be done by pairwise comparison with time complexity \( O(|TS|^2) \).

**Input:** test suite \( TS = \{tc_1, \ldots, tc_m\} \) sorted ascending by length, redundant test cases eliminated

**Output:** exact minimal test suite \( TS_0 \subseteq TS \)

1: \( n_{res} = (1, \ldots, 1) \)
2: Stack \( S = \{()\} \)
3: while \( S \neq \emptyset \) do
   4:   \( n = S.pop \)
   5:   if \( \text{level}(n) < m - 1 \) then
   6:      \( n_0 = (n, 0) \)
   7:      if \{\( tc_i : (n_0)_i = 1 \) or \( i > \text{level}(n_0) \)\} is complete test suite then
   8:         if \( \text{value}(n_0) + |tc_{i+1}| < \text{value}(n_{res}) \) then
   9:            \( S.push(n_0) \)
 10:   end
   11: else if \( \text{level}(n_1) < m \) then
 12:       \( n_1 = (n, 1) \)
 13:       if \( \text{value}(n_1) < \text{value}(n_{res}) \) then
 14:          if \{\( tc_i : (n_1)_i = 1 \)\} is complete test suite then
 15:             \( n_{res} = n_1 \)
 16:             \( S = \{n' \in S : \text{value}(n') < \text{value}(n_1)\} \)
 17:          end
 18:       else
 19:          \( S.push(n_1) \)
 20:       end
 21: end
 22: return \{\( tc_i : (n_{res})_i = 1 \}\)

**Algorithm 2.** Branch and Bound

**Test Case Distribution**

In this section we formulate the test case distribution problem and describe it mathematically by introducing a sequence of functions that measures distribution quality in terms of variances. Using these functions we show how to modify both Greedy and Branch and Bound algorithms in order to improve test case distribution.

Let us start with the simple example depicted in Figure 4.20 together with two test suites \( TS_1 \) and \( TS_2 \). Both test suites are minimal with respect to action coverage and vary only in one action, which is denoted in bold font. Actions A and B are used an equal number of times in \( TS_2 \), while they are not in \( TS_1 \). From our industrial experience, the second test suite is more desirable as it increases confidence of MBT users. Therefore, a smoother test case distribution or a better distribution quality is a practical requirement.
Let us now move on to the more complex example depicted in Figure 4.21, which again comes with two test suites $\text{TS}_3$ and $\text{TS}_4$. As before, both test suites are minimal and their difference is marked in bold font. However this time all actions are called the same number of times in $\text{TS}_3$ as in $\text{TS}_4$. Nevertheless the second suite can be regarded as having a smoother test case distribution since it includes four different variations for the first two actions, while the first one only includes two. In a similar fashion one could obtain larger examples, where distribution quality only depends on the number of occurrences of even larger test case subsequences.

**Formalization** We now want to give a mathematical description of the problem. Therefore we propose a way how to measure the distribution quality of a given test suite $\text{TS}_0 \subseteq \text{TS}$.

A simple approach is to evaluate the variance of action calls: For any test case $\text{tc} = (a_1, \ldots, a_k) \in \text{TS}$ we define a counting function $d^{(1)}_\text{tc}$, which assigns to each action $a \in \mathcal{A}$ the number of times it is called by $\text{tc}$:

$$d^{(1)}_\text{tc} : \mathcal{A} \rightarrow \mathbb{N}, \quad a \mapsto |\{i \in \{1, \ldots, k\} : a_i = a\}|.$$  

(4.5)

Equivalently for any test suite $\text{TS}_0 = \{\text{tc}_1, \ldots, \text{tc}_m\} \subseteq \text{TS}$ we define a counting function $d^{(1)}_{\text{TS}_0}$, which assigns to each action $a \in \mathcal{A}$ the number of times it is called by whole $\text{TS}_0$. This can be formalized as

$$d^{(1)}_{\text{TS}_0} : \mathcal{A} \rightarrow \mathbb{N}, \quad a \mapsto \sum_{i=1}^{m} d^{(1)}_{\text{tc}_i}(a).$$  

(4.6)
Example 1

<table>
<thead>
<tr>
<th>Suite</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS₁</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TS₂</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example 2

<table>
<thead>
<tr>
<th>Suite</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS₃</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TS₄</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Values of \( d^{(1)}_{TS_i} \)

Now the mean number of calls per action in TS₀ is given as

\[
\overline{d}^{(1)}_{TS_0} = \frac{1}{|A|} \sum_{a \in A} d^{(1)}_{TS_0}(a) = \frac{1}{|A|} \sum_{i=1}^{m} |tc_i|.
\]  (4.7)

If TS₀ is well distributed, we would expect \( d^{(1)}_{TS_0}(a) \) not to vary much, but to stay near its mean value for all \( a \in A \). Hence, the variance of \( d^{(1)}_{TS_0} \) or the variance of action calls in TS₀ will be our first measure for distribution quality:

\[
\text{Var}(TS_0) = \frac{1}{|A|} \sum_{a \in A} \left( d^{(1)}_{TS_0}(a) - \overline{d}^{(1)}_{TS_0} \right)^2.
\]  (4.8)

Let us apply this definition on the examples from above. The values obtained for \( d^{(1)}_{TS_i} \) are denoted in table 4.1. Further calculations show, that for the first example \( \text{Var}(TS_1) = 1.35 \), while \( \text{Var}(TS_2) = 1.06 \). As expected TS₂ has a better distribution quality than TS₁. Nevertheless for the second example we get \( \text{Var}(TS_3) = \text{Var}(TS_4) = 0.84 \), i.e. we cannot determine the difference between TS₃ and TS₄ by \( \text{Var} \). This is not surprising, since the counting functions for both test suites are identical.

To circumvent this problem we generalize our ideas in order to construct a variance of action-sequence calls for sequences of a fixed length \( p \): Let \( A^{(p)} \) be the set of all action-sequences of length \( p \) that are part of at least one test case in TS. To ease notation we will use the symbol \( a \) to denote such sequences. First for any test case \( tc = (a_1, \ldots, a_k) \in TS \) we again define a counting function \( d^{(p)}_{tc} \), which assigns to each action-sequence \( a \in A^{(p)} \) the number of times it is called by \( tc \):

\[
d^{(p)}_{tc} : A^{(p)} \to \mathbb{N}, \quad a \mapsto \left| \{ i \in \{ p, \ldots, k \} : (a_{i-p+1}, \ldots, a_i) = a \} \right|.
\]  (4.9)

As before we can use these functions to define a counting function \( d^{(p)}_{TS_0} \) for a
Table 4.2: Values of $d^{(2)}_{T_{S_i}}$

<table>
<thead>
<tr>
<th>Suite</th>
<th>(A,C)</th>
<th>(A,D)</th>
<th>(B,C)</th>
<th>(B,D)</th>
<th>(C,E)</th>
<th>(D,E)</th>
<th>(E,F)</th>
<th>(E,G)</th>
<th>(E,H)</th>
<th>(E,I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{S_1}$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$T_{S_2}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2: Values of $d^{(2)}_{T_{S_i}}$

whole test suite $T_{S_0} = \{ tc_1, \ldots, tc_m \} \subseteq TS$, i.e.

$$d^{(p)}_{T_{S_0}} : A^{(p)} \to \mathbb{N}, \ a \mapsto \sum_{i=1}^{m} d^{(p)}_{tc_i}(a).$$  \hfill (4.10)

The mean amount of calls per $p$-action-sequence is given by

$$\overline{d^{(p)}_{T_{S_0}}} = \frac{1}{|A^{(p)}|} \sum_{a \in A^{(p)}} d^{(p)}_{T_{S_0}}(a)$$ \hfill (4.11)

and we can finally define the variance of $d^{(p)}_{T_{S_0}}$ or the variance of $p$-action-sequences by

$$\text{Var}_p(T_{S_0}) = \frac{1}{|A^{(p)}|} \sum_{a \in A^{(p)}} \left( d^{(p)}_{T_{S_0}}(a) - \overline{d^{(p)}_{T_{S_0}}} \right)^2.$$ \hfill (4.12)

Let us apply this definition with $p = 2$ to example 2 from above. The values obtained for $d^{(2)}_{T_{S_3}}$ are denoted in table 4.2. Further calculation shows, that $\text{Var}_2(T_{S_3}) = 0.56$, while $\text{Var}_2(T_{S_4}) = 0.16$. This means that $\text{Var}_2$ rates the second test suite better than the first one, as demanded by the motivation from the beginning of this section.

**Lexicographical approaches**

**Greedy** We present concrete implementations that respect distribution quality as described in section 4.3.5. The approaches proposed here optimize with respect to the total number of action calls first and with respect to distribution quality afterwards.

In order to modify the Greedy algorithm, we have to rate distribution quality not of a complete test suite, but of a partially constructed one. More precisely, we have to determine how well an additional test case $tc = (a_1, \ldots, a_k)$ would fit in a non-complete test suite $T_{S_0}$. This is done by the quality functions

$$d^{(p)}_{T_{S_0}}(tc) = \sum_{i=p}^{k} d^{(p)}_{T_{S_0}}(a_{i-p+1}, \ldots, a_i).$$ \hfill (4.13)
Using them we can modify the Greedy algorithm in order to achieve better distribution quality by exchanging line 7 with the following steps:

\[
\text{for } i = 1 \rightarrow p \text{ do} \\
\quad T_S'_{i} = \left\{ tc \in T_S'_{(i-1)} : q^{(i)}_{T_{S_0}}(tc) \text{ is minimal} \right\} \\
\quad \text{Pick } tc \in T_S'_{p}
\]

**Branch and Bound** To modify the Branch and Bound algorithm from section 4.3.5, just some simple modifications have to be made in lines 8, 11 and 13–14, so that the set of all optimal nodes \( N^{(0)} \) is returned instead of just one optimal node \( n_{res} \). For example we have to exchange the sharp inequality \((<)\) in line 8 with a weak one \((\leq)\). We will come back to this particular replacement when discussing experimental results.

More importantly, we have to choose a single node from \( N^{(0)} \) when the main loop is finished. This is done such that the according test suite has optimal distribution quality. Therefore we extend definition (4.12) to nodes \( n = (n_1, \ldots, n_i) \) by

\[
\text{Var}_p(n) = \text{Var}_p\left(\{tc_j : n_j = 1\}\right)
\]

and insert the following steps between lines 16 and 17:

1: for \( i = 1 \rightarrow p \) do
2: \( N^{(i)} = \{n \in N^{(i-1)} : \text{Var}_i(n) \text{ is minimal}\} \)
3: Pick \( n_{res} \in N^{(p)} \)

**Multi-objective Greedy approach** In this subsection we present another approach to the problem, which is based on the Greedy algorithm from section 4.3.5. The strategy proposed here optimizes with respect to both objectives (test suite size and distribution quality) simultaneously and allows arbitrary weighting between them.

As discussed in 4.3.5, the requirement of \( T_{S_0} \subseteq T_S \) having a good distribution quality is equivalent to the one that its variances are minimal. To simplify matters we will only consider \( \text{Var}_1 \) here. Therefore this objective can be stated as

\[
\text{Var}_1(T_{S_0}) = \frac{1}{|A|} \sum_{a \in A} \left( d^{(1)}_{T_{S_0}}(a) - \overline{d^{(1)}_{T_{S_0}}} \right)^2 \rightarrow \min!
\]

To combine it with the goal of minimizing the number of action calls we use a probabilistic approach. The probability of being contained in an
optimal test suite with respect to distribution quality is greater for some \( tc \in TS \setminus TS_0 \), if its distribution variance contribution is small. This contribution can be expressed by the value of \( \text{Var}_1 \) that would result when including \( tc \) into \( TS_0 \), \( \text{Var}_1(TS_0 \cup \{tc\}) \), in relation to the sum of variances for all elements in \( TS \setminus TS_0 \), namely \( \sum_{\tilde{tc} \in TS \setminus TS_0} \text{Var}_1(TS_0 \cup \{\tilde{tc}\}) \).

Thus, we start with an initially empty set of test cases \( TS_0 \) and iteratively add test cases \( tc \in TS \) to \( TS_0 \) such that distribution variance increase is minimized. This leads to the following definition of probabilities for each \( tc \in TS \setminus TS_0 \):

\[
p_{\text{var}}(tc) := \frac{1 - \frac{\text{Var}_1(TS_0 \cup \{tc\})}{\sum_{\tilde{tc} \in TS \setminus TS_0} \text{Var}_1(TS_0 \cup \{\tilde{tc}\})}}{|\{tc \in TS \setminus TS_0\}| - 1}.
\]  

(4.16)

Obviously, \( 0 \leq p_{\text{var}}(tc) \leq 1 \) for all \( tc \in TS \setminus TS_0 \) by construction and the expressions defined in equation (4.16) can be interpreted as probabilities.

For our primary objective of minimizing the number of action calls we will construct probabilities in an analogous way by using the same decision criterion as for the Greedy algorithm from section 2.1. Referring to line 5 of the algorithm we denote \( w_{tc} = \frac{1}{|tc|} \cdot |\text{cov}(tc) \setminus \mathcal{R}_0| \) as the weight for each \( tc \in TS \setminus TS_0 \). The weight of a test case \( tc \) is the total number of requirements \( r \in \mathcal{R} \) that are satisfied by \( tc \) but not yet covered by any test case in \( TS_0 \). This value is normalized by the length of \( tc \). It is clear that test cases with a high weight will more probably be contained in a test suite that is optimal with respect to test suite size than test cases with a lower weight. See [Chv79] for more details. We can thus define probabilities for each \( tc \in TS \setminus TS_0 \) as follows:

\[
p_{\text{rate}}(tc) := \frac{w_{tc}}{\sum_{\tilde{tc} \in TS \setminus TS_0} w_{\tilde{tc}}}.
\]  

(4.17)

So, given a rate proportion coefficient \( \delta \in [0, 1] \) we can construct a weighted probability distribution by defining

\[
p(tc) := \delta \cdot p_{\text{rate}}(tc) + (1 - \delta) \cdot p_{\text{var}}(tc)
\]  

(4.18)

for each \( tc \in TS \setminus TS_0 \).

A resulting Greedy strategy would be to iteratively sort test cases by their combined probability descending and add the test case with highest probability value to the final solution set. These preliminary considerations yield to a Greedy algorithm like the one presented in section 2.1, except that lines 4 to 6 have to be replaced by the following steps\(^3\):

\(^3\)Note that for \( \delta = 1 \) this algorithm is equivalent to the one proposed by [Chv79].
Table 4.3: Use cases

| #  | \(|TS|\) | AC | AC\(^{(2)}\) | \(|A|\) | \(|A^{(2)}|\) |
|-----|---------|----|-----------|-------|----------|
| I   | 15      | 84 | 69        | 13    | 26       |
| II  | 21      | 123| 102       | 10    | 15       |
| III | 32      | 128| 96        | 13    | 28       |
| IV  | 41      | 164| 123       | 15    | 31       |
| V   | 30      | 189| 159       | 25    | 35       |
| VI  | 36      | 190| 154       | 31    | 40       |
| VII | 46      | 317| 271       | 40    | 52       |
| VIII| 45      | 374| 329       | 23    | 36       |
| IX  | 120     | 600| 480       | 15    | 33       |
| X   | 132     | 1306| 1174    | 26    | 40       |
| XI  | 512     | 6656| 6144   | 30    | 54       |
| XII | 284     | 1600| 1316   | 140   | 422      |
| XIII| 625     | 4375| 3750   | 23    | 86       |

Experimental Results

In this subsection we present experimental results for the optimization techniques described above. Due to computational constraints we only considered \(p = 2\) for the lexicographical algorithms. All computations were performed on an AMD Opteron (tm) Quad Core with 2.60 GHz and 32 Gigabytes of RAM. As input we derived 13 different transition state machines which were designed on the basis of industrial case studies. To get realistic statements for the context of our work, most of our use cases are small- or intermediate-sized (I-IX). Nevertheless we included some larger models as well (X-XIII). Actually (XII) and (XIII) have proven to be too large to be optimized with algorithms of the Branch and Bound-type.

For detailed information about the use cases consider table 4.3. It contains the number of test cases (\(|TS|\)), the number of action calls (AC), the number of action-pair calls (AC\(^{(2)}\)), the number of actions (\(|A|\)) and the number of action-pairs (\(|A^{(2)}|\)) for each of our use cases. Nevertheless the full model definitions may not be published due to legal reasons.

Greedy Based Approaches In the following we compare experimental results for the original Greedy algorithm from [Chv79] with our proposed
Table 4.4: Results for Greedy algorithms

<table>
<thead>
<tr>
<th>#</th>
<th>AC</th>
<th>Time</th>
<th>AC0</th>
<th>Var1</th>
<th>Var2</th>
<th>Time</th>
<th>AC0</th>
<th>Var1</th>
<th>Var2</th>
<th>Time</th>
<th>AC0</th>
<th>Var1</th>
<th>Var2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>84</td>
<td>0.06</td>
<td>16</td>
<td>0.33</td>
<td>0.00</td>
<td>0.08</td>
<td>16</td>
<td>0.33</td>
<td>0.00</td>
<td>0.09</td>
<td>16</td>
<td>0.33</td>
<td>0.00</td>
</tr>
<tr>
<td>II</td>
<td>123</td>
<td>0.06</td>
<td>25</td>
<td>2.65</td>
<td>0.86</td>
<td>0.08</td>
<td>25</td>
<td>2.05</td>
<td>0.56</td>
<td>0.09</td>
<td>25</td>
<td>2.05</td>
<td>0.56</td>
</tr>
<tr>
<td>III</td>
<td>128</td>
<td>0.08</td>
<td>32</td>
<td>7.17</td>
<td>1.89</td>
<td>0.09</td>
<td>32</td>
<td>4.40</td>
<td>0.16</td>
<td>0.08</td>
<td>32</td>
<td>4.40</td>
<td>0.89</td>
</tr>
<tr>
<td>IV</td>
<td>164</td>
<td>0.08</td>
<td>32</td>
<td>4.78</td>
<td>0.76</td>
<td>0.08</td>
<td>32</td>
<td>3.45</td>
<td>0.08</td>
<td>0.08</td>
<td>32</td>
<td>3.45</td>
<td>0.12</td>
</tr>
<tr>
<td>V</td>
<td>189</td>
<td>0.08</td>
<td>41</td>
<td>1.83</td>
<td>0.18</td>
<td>0.08</td>
<td>41</td>
<td>1.83</td>
<td>0.18</td>
<td>0.08</td>
<td>41</td>
<td>1.83</td>
<td>0.18</td>
</tr>
<tr>
<td>VI</td>
<td>190</td>
<td>0.08</td>
<td>47</td>
<td>0.64</td>
<td>0.15</td>
<td>0.09</td>
<td>48</td>
<td>0.64</td>
<td>0.21</td>
<td>0.09</td>
<td>48</td>
<td>0.64</td>
<td>0.21</td>
</tr>
<tr>
<td>VII</td>
<td>317</td>
<td>0.08</td>
<td>69</td>
<td>1.30</td>
<td>0.40</td>
<td>0.09</td>
<td>69</td>
<td>1.30</td>
<td>0.40</td>
<td>0.09</td>
<td>69</td>
<td>1.30</td>
<td>0.40</td>
</tr>
<tr>
<td>VIII</td>
<td>374</td>
<td>0.08</td>
<td>47</td>
<td>2.22</td>
<td>0.45</td>
<td>0.09</td>
<td>47</td>
<td>2.22</td>
<td>0.45</td>
<td>0.08</td>
<td>47</td>
<td>2.22</td>
<td>0.45</td>
</tr>
<tr>
<td>IX</td>
<td>600</td>
<td>0.09</td>
<td>25</td>
<td>1.56</td>
<td>0.15</td>
<td>0.13</td>
<td>25</td>
<td>1.46</td>
<td>0.19</td>
<td>0.11</td>
<td>25</td>
<td>1.16</td>
<td>0.19</td>
</tr>
<tr>
<td>X</td>
<td>1306</td>
<td>0.14</td>
<td>44</td>
<td>1.37</td>
<td>0.30</td>
<td>0.16</td>
<td>44</td>
<td>1.14</td>
<td>0.17</td>
<td>0.17</td>
<td>44</td>
<td>1.14</td>
<td>0.15</td>
</tr>
<tr>
<td>XI</td>
<td>6656</td>
<td>0.50</td>
<td>65</td>
<td>1.87</td>
<td>1.00</td>
<td>0.52</td>
<td>65</td>
<td>1.61</td>
<td>0.72</td>
<td>0.52</td>
<td>65</td>
<td>1.61</td>
<td>0.72</td>
</tr>
<tr>
<td>XII</td>
<td>1600</td>
<td>0.28</td>
<td>429</td>
<td>92.66</td>
<td>4.63</td>
<td>0.31</td>
<td>425</td>
<td>90.09</td>
<td>4.37</td>
<td>0.30</td>
<td>425</td>
<td>89.95</td>
<td>4.28</td>
</tr>
<tr>
<td>XIII</td>
<td>4375</td>
<td>0.36</td>
<td>35</td>
<td>1.82</td>
<td>0.59</td>
<td>0.39</td>
<td>35</td>
<td>1.82</td>
<td>0.59</td>
<td>0.39</td>
<td>35</td>
<td>1.82</td>
<td>0.59</td>
</tr>
</tbody>
</table>

Table 4.4: Results for Greedy algorithms

extensions. As parameters we choose $p = 2$ for the lexicographical variation and $\delta = 0.5$ for the multi-objective one. Our results are displayed in Table 4.4.

The second column (AC) contains the number of action calls in the unmodified test suite TS, while columns 4, 8, and 12 (AC0) in each case contain the number of action calls in the resulting test suite TS0. Columns 3, 7, and 11 (Time) denote the computation time in seconds needed to run the specific algorithm and columns 5–6, 9–10, and 13–14 present the variance values as defined by equations (4.8) and (4.12) for the corresponding reduced suite TS0.

We can see that the computation time for the lexicographical as well as for the multi-objective approach is always higher than the one for the original Greedy algorithm. This is just as expected, since computation and consideration of action call distribution takes additional time. Nevertheless we also note, that the additional time consumption is usually not very significant.

The number of action calls is always equal for all algorithms except for the use cases VI and XII. The variances are usually decreased when running a modified algorithm, although for Var2 this does not always hold. This is reasonable as well, since the lexicographical Greedy optimizes with respect to Var1 first and the multi-objective Greedy does not consider Var2 at all.

Another conclusion one can draw from the results is, that the two modifications of Greedy behave quite similarly except for use cases III and IV, where a significant difference in Var2 can be noticed.

To sum up this discussion, we present the average values over the eleven

82
first use cases in table 4.5⁴. Here we can see, that the variance values obtained by the original Greedy algorithm can be reduced by almost about 22% on average when using the lexicographical or the multi-objective approach. The values for $\text{Var}_2$ can be even be improved by almost 50% or 37% on average when using the lexicographical or the multi-objective approach, respectively.

### Branch and Bound Based Approaches

Now let us compare the standard Branch and Bound approach from section 4.3.5 with our extension from 4.3.5. As parameter we again choose $p = 2$. The use cases are similar to the ones from the last section except that XII and XIII are excluded since the algorithms did not terminate within a reasonable time constraint. Additionally we have to remark that (in difference to the Greedy algorithms) redundant test cases as described in 4.3.5 were removed from the test suite prior to running the algorithms.

Our results are displayed in table 4.6. Here the second column (AC) contains the number of action calls in test suite $TS$ after performing the removal of redundant test cases, but before running the Branch and Bound algorithms. The third column (AC₀) contains the number of action calls in the resulting test suite $TS₀$, i.e after performing Branch and Bound. By construction of the algorithms these numbers are always minimal and thus equal. Columns 4–6 contain further results for the unmodified standard algorithm, while columns 7–9 contain the further results for the modification discussed in section 4.3.5.

Considering use cases IX and XI it is evident, that the lexicographical approach is totally outperformed by the original algorithm in time. Analyzing this problem leads to the conclusion, that most of the additional time consumption yields from the change in line 8 of the algorithm, where a sharp inequality ($<$) is replaced with a weak one ($\leq$) in order to return all minimal solutions. Hence, we also tried to use another modified version, which uses sharp inequality ($<$) and therefore does not return all minimal solutions, but only a subset of these. Afterwards, the best suite with respect to

---

⁴The last two examples have been excluded to ensure comparability with Branch and Bound results, see below.
distribution quality is chosen out of this subset just as in the lexicographical approach. The results for this modified lexicographical algorithm are displayed in columns 10-12 of table 4.6.

To compare the algorithms with each other we again computed average values, which are presented in table 4.7. One can see that standard Branch and Bound and the modified lexicographical version have almost equal computation time on average (about 50 seconds). Nevertheless the average variance values for the latter one are considerably better than those for standard Branch and Bound (about 20% for Var$_1$ and almost 60% for Var$_2$). The average variance values for the “exact” lexicographical version are of course even smaller, but do not advance very much (only about 2% for Var$_1$ and 10% for Var$_2$). However, this benefit comes with the cost of a significant increase in computation time (about 400%).

**Comparison of Results** We can see that for the Greedy as well as for the Branch and Bound approaches taking distribution of action calls into account can yield to considerably smaller variances than for the standard versions. Nevertheless except for the lexicographical Branch and Bound algorithm computational effort for the extended approaches is not significantly higher.

<table>
<thead>
<tr>
<th>#</th>
<th>AC</th>
<th>AC$_0$</th>
<th>Time</th>
<th>Var$_1$</th>
<th>Var$_2$</th>
<th>Time</th>
<th>Var$_1$</th>
<th>Var$_2$</th>
<th>Time</th>
<th>Var$_1$</th>
<th>Var$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>78</td>
<td>16</td>
<td>0.06</td>
<td>0.33</td>
<td>0.00</td>
<td>0.09</td>
<td>0.33</td>
<td>0.00</td>
<td>0.09</td>
<td>0.33</td>
<td>0.00</td>
</tr>
<tr>
<td>II</td>
<td>123</td>
<td>17</td>
<td>0.06</td>
<td>1.01</td>
<td>0.14</td>
<td>0.09</td>
<td>1.01</td>
<td>0.20</td>
<td>0.11</td>
<td>1.01</td>
<td>0.14</td>
</tr>
<tr>
<td>III</td>
<td>128</td>
<td>32</td>
<td>3.66</td>
<td>7.17</td>
<td>1.89</td>
<td>7.55</td>
<td>4.40</td>
<td>0.16</td>
<td>3.78</td>
<td>4.40</td>
<td>0.16</td>
</tr>
<tr>
<td>IV</td>
<td>164</td>
<td>32</td>
<td>9.05</td>
<td>5.32</td>
<td>1.25</td>
<td>24.06</td>
<td>3.45</td>
<td>0.08</td>
<td>9.56</td>
<td>3.45</td>
<td>0.08</td>
</tr>
<tr>
<td>V</td>
<td>189</td>
<td>38</td>
<td>0.39</td>
<td>1.45</td>
<td>0.16</td>
<td>0.47</td>
<td>1.45</td>
<td>0.16</td>
<td>0.42</td>
<td>1.45</td>
<td>0.16</td>
</tr>
<tr>
<td>VI</td>
<td>190</td>
<td>44</td>
<td>10.25</td>
<td>1.08</td>
<td>0.44</td>
<td>11.28</td>
<td>1.08</td>
<td>0.44</td>
<td>10.28</td>
<td>1.08</td>
<td>0.44</td>
</tr>
<tr>
<td>VII</td>
<td>317</td>
<td>64</td>
<td>239.61</td>
<td>0.89</td>
<td>0.28</td>
<td>271.66</td>
<td>0.89</td>
<td>0.27</td>
<td>234.02</td>
<td>0.89</td>
<td>0.27</td>
</tr>
<tr>
<td>VIII</td>
<td>185</td>
<td>47</td>
<td>0.06</td>
<td>2.22</td>
<td>0.31</td>
<td>0.11</td>
<td>2.22</td>
<td>0.28</td>
<td>0.11</td>
<td>2.22</td>
<td>0.28</td>
</tr>
<tr>
<td>IX</td>
<td>600</td>
<td>25</td>
<td>77.49</td>
<td>1.56</td>
<td>0.31</td>
<td>1755.37</td>
<td>1.16</td>
<td>0.05</td>
<td>81.72</td>
<td>1.16</td>
<td>0.10</td>
</tr>
<tr>
<td>X</td>
<td>1089</td>
<td>43</td>
<td>198.68</td>
<td>1.38</td>
<td>0.40</td>
<td>230.98</td>
<td>1.15</td>
<td>0.15</td>
<td>198.94</td>
<td>1.38</td>
<td>0.40</td>
</tr>
<tr>
<td>XI</td>
<td>4368</td>
<td>52</td>
<td>19.00</td>
<td>1.13</td>
<td>0.52</td>
<td>484.06</td>
<td>1.00</td>
<td>0.37</td>
<td>19.38</td>
<td>1.00</td>
<td>0.37</td>
</tr>
</tbody>
</table>

**Table 4.6: Results for Branch and Bound algorithms**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg Time</th>
<th>Avg AC$_0$</th>
<th>Avg Var$_1$</th>
<th>Avg Var$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>50.755</td>
<td>37.273</td>
<td>2.139</td>
<td>0.518</td>
</tr>
<tr>
<td>Lexicographical</td>
<td>253.248</td>
<td>37.273</td>
<td>1.648</td>
<td>0.197</td>
</tr>
<tr>
<td>Lex. Mod.</td>
<td>50.765</td>
<td>37.273</td>
<td>1.669</td>
<td>0.218</td>
</tr>
</tbody>
</table>

**Table 4.7: Comparison of Branch and Bound algorithms**
When comparing the results for the extended Branch and Bound with the extended Greedy approaches we see that the number of action calls for all Branch and Bound approaches is about 7.5\% less on average than the corresponding numbers for the Greedy strategies. Furthermore, the variance of action calls can be decreased by about 10\% and the variance of action-pair calls even by 22\% up to 43\% on average when using an extended Branch and Bound algorithm instead of lexicographical or multi-objective Greedy. However, for the most examples this is dearly bought with a dramatically higher computation time in comparison to the Greedy strategies.

**Conclusion** We presented two classical solutions for the test suite reduction problem, namely the Branch and Bound algorithm, which computes an exact solution in exponential time, and the Greedy heuristic, which yields the best approximation possible in polynomial time. Based on these algorithms we introduced modifications to advance distribution quality.

A main contribution of this paper is the formalization of the term “distribution quality” itself. With the variances $\text{Var}_p$ at hand a mathematical description of the problem can easily be given. Our modifications of the algorithms introduce simple but effective ways to use this description in order to solve the problem.

Experimental results support these approaches. At no time the result of a modified algorithm was outperformed by the result of its unmodified counterpart in distribution quality. Conversely the variances shrunk in most use cases, at times tremendously. Both variations of the Greedy algorithm performed almost equally and were only marginally slower than the unmodified version. On the other hand it showed that our first modification of Branch and Bound was significantly slower, such that we would not advice to use it. Nevertheless we also introduced a variation that comes with nearly the full advantage of a better distribution quality, but computes insignificant longer compared to standard Branch and Bound.

**Multi-objective test suite reductions**

As seen before, large test suites need to be optimised according to different criteria. We have seen how the greedy or branch-and-bound algorithms can be adapted to achieve such test suite optimisation. In the following, we follow a more general approach of defining the reduction criteria as multi-objective test suite optimization problems. They are solved using two modern Multi-Objective Evolutionary Algorithms, namely: NSGA-II [DAPM00] and SPEA-2 [ZLT01]. The experiments have been conducted using five test suites generated from two industrial inspired Event-B models.
The rest of this section is structured as follows. We introduce the test suite optimization problem for Event-B models, then we mathematically define the six different test suite optimizations, and we finally describe the experiment setup and results.

The presented results are based on [Din11] and have used publicly available Event-B models rather than the SAP internal models.

**Multi-Objective Test Suite Optimization for Event-B Models**

Given an Event-B machine $M$ with $E = \{e_1, e_2, ..., e_m\}$ the set of its events, a test case can be defined as a sequence of events in $E$ that can be executed in the machine $M$ (an execution path). Each test case begins with a special event called *INITIALISATION* which serves to initialize the global variables of the machine before starting the execution of a test case. A test suite is by definition a collection of test cases.

We introduce the multi-objective test suite minimization problem. We adopt here the definitions from [YH07]. Generally, a multi-objective optimization problem can be defined as to find a vector of decision variables $x$, which optimizes a vector of $M$ objective functions $f_i(x), 1 \leq i \leq M$. The objective functions are the mathematical formulations of the optimization criteria. Usually, these functions are conflicting, which means that improvements with respect to one function can only be achieved when impairing the solution quality with respect to another objective function. Solutions that can not be improved with respect to any functions without impairing another one are called *Pareto-optimal solutions*.

Formally, let us assume that, without loss of generality, the goal is to minimize the functions $f_i(x), 1 \leq i \leq M$. A decision vector $x$ is said to dominate a decision vector $y$ (we write $x \succ y$) if and only if the following property is satisfied by their objective vectors:

$$f_i(x) \leq f_i(y), \forall i \in \{1, 2, ..., M\} \text{ and } \exists i_0 \in \{1, 2, ..., M\}, f_{i_0}(x) < f_{i_0}(y).$$

The dominance relations states that a solution $x$ is preferable to another solution $y$ if $x$ is at least as good as $y$ in all objectives and better with respect to at least one objective. The *Pareto-optimal set* is the set of all decision vectors that are not dominated by any other decision vectors. The corresponding objective vectors are said to from *Pareto frontier*. Therefore, the multi-objective optimization problem can be defined in the following manner:

*Given:* a vector of decision variables, $x$, and a set of objective functions, $f_i(x), 1 \leq i \leq M$,

*Problem:* minimize $\{f_1(x), f_2(x), ..., f_M(x)\}$ by finding the Pareto-optimal set over the feasible set of solutions.
With respect to multi-criteria test suite optimization, the objective functions $f_i$ are the mathematical descriptions of the testing criteria that must be satisfied to provide desired adequate testing of the model. In real industrial testing problems, there exist multiple test criteria, because a single ideal criterion is simply impossible to be achieved. For example, a frequently optimization problem is to produce a minimal test suite which achieves maximal coverage of the model entities with a minimal execution cost. Therefore, this is a bi-objective minimization test suite problem.

Formally, multi-objective test suite optimization problem can be defined in the following manner [YH10]:

**Multi-Objective Test Suite Optimization.**

*Given:* a test suite $TS$, a vector of $M$ objective functions $f_i, 1 \leq i \leq M$

*Problem:* to produce a subset $T \subset TS$, such that $T$ is a Pareto-optimal set with respect to the set of the above objective functions.

In the following, we instantiate this general multi-objective test suite optimization problem with respect to our Event-B models.

We assume an Event-B machine $M$ for which we have generated a test suite $TS$. Of course, $TS$ satisfies a set of test requirements which are expressed as a level of coverage of the model. For the moment, we only consider that the test suite $TS$ achieves the following simple coverage criterion:

**Event Coverage Criterion:** A test suite $TS = \{t_1, ..., t_m\}$ of $m$ test cases for an Event-B model $M$ is said to achieve event coverage criterion if and only if for each event $e$ of the model $M$ there exists a test case $t_i \in TS$ which covers $e$.

Having the above criterion in mind, we can formulate the following optimization problem:

**Test Suite Minimization Problem.**

*Given:* A test suite $TS$ generated for a machine $M$ with $E = \{e_1, e_2, ..., e_n\}$ the set of events, and subsets of $TS$, $T_i$s, one associated with each of the $e_i$s such that any one of the test cases $t_j$ belonging to $T_i$ can be used to cover $e_i$.

*Problem:* Find minimal test suite $T$ from $TS$ which covers all $e_i$.

As also mentioned in the previous section, this problem is NP-complete because it can be reduced to the minimum set-cover problem [CSRL01] as follows. We recall that for us a test case $tc \in TS$ is an execution path which consists in a sequence of events from $E$. Let be $cov(tc) = \{e \in E | tc$ covers $e\}$ the set of events covered by test case $tc$. By definition, $cov(tc)$ is a subset of $E$. Therefore the solution $T$ of the above test suite minimization problem is exactly a minimum set cover for $E$, because

$$\bigcup_{t \in T} cov(t) = E$$
and $T$ is the minimal subset of $TS$ which covers $E$.

Many solutions have been proposed to solve this test suite minimization problem [Chv79, HGS93, Agr99, MB03, BMK04]. Due to its exponential complexity, we use Multi-Objective Evolutionary Algorithms for solving it. For that, we mathematically reformulate it as a constraint bi-objective test suite optimization problem (see TSO1 problem below).

**Optimization Criteria** Based on practical experience at SAP, we propose here different test suite optimization criteria.

**TSO1-Minimizing the size of the test suite.** Due to the restrictions of time, obtaining a minimal test suite which achieves maximal level of coverage is of particular interest among testers. Therefore the goal of this problem is to produce a test suite that contains the smallest possible number of test cases that achieve the same coverage (in our case, the event coverage) as the complete test suite. We formulate this problem as a constraint bi-objective optimization problem: maximize event coverage (the first objective) by a minimum number of test cases (the second objective) under the constraint that at least a test case has been selected. The problem can be mathematically described in the following manner.

Let be $TS = \{t_1, t_2, ..., t_m\}$ the initial set of $m$ test cases and $E = \{e_1, e_2, ..., e_n\}$ the set of the events to be covered. We recall that $\text{cov}(tc)$ is the set of events covered by the test case $tc$. Given an order between the elements of a set, a subset $T \subset TS$ can be mathematically represented by a binary vector $x = (x_1, x_2, ..., x_m) \in \{0, 1\}^m$ with

$$
x_i = \begin{cases} 
1, & t_i \in T \\
0, & t_i \notin T
\end{cases}, 1 \leq i \leq m.
$$

Therefore the constraint bi-objective test suite optimization problem to be solved is the following:

Minimize $(f_1(x), f_2(x))$

Subject to:

$$
\sum_{i=1}^{m} x_i \geq 1 \quad (T \neq \emptyset)
$$

Where:

$$
f_1(x) = 1 - \sum_{i=1}^{m} (x_i \cdot \frac{|\text{cov}(t_i)|}{n}) \quad \text{(maximize the coverage)}
$$

$$
f_2(x) = \frac{\sum_{i=1}^{m} x_i}{m} \quad \text{(minimize the size of test suite)}.
$$
A Pareto-optimal solution of the above problem corresponds to a minimal subset of the test suite \( TS \) which achieves a maximal level of coverage. More, we can see that \( f_1 : \{0, 1\}^m \rightarrow [0, 1] \) and \( f_2 : \{0, 1\}^m \rightarrow (0, 1] \). Therefore we avoid to select the empty set as a solution.

**TSO2-Minimizing the number of the executed events.** In order to reduce the effort of the testing process, the number of executed events from the whole test suite should be minimized. Therefore we want to obtain test suites which achieve the event coverage criterion with a minimum number of executed events. The first objective function \( f_1 \) and the constraint from the problem TSO1 remain valid. Let be \( len(tc) \) the length of the test case \( tc \in TS \). The second objective function \( f_2 \) which can be used to minimize the number of executed events by the subset \( T \subset TS \) is

\[
f_2(x) = \frac{1}{\sum_{k=1}^{m} len(t_k)} \sum_{i=1}^{m} (x_i \cdot len(t_i)).
\]

**TSO3-Minimizing the length of the longest execution path.** The longer execution paths are harder to maintain. In this problem we control the lengths of the execution paths by minimizing the length of the longest test case. The mathematical formulation is the following:

Minimize \((f_1(x), f_2(x))\)

Where \( f_1(x) \) is the same as for TSO1 problem and

\[
f_2(x) = \max\{len(t_i)|x_i = 1 \text{ and } 1 \leq i \leq m\}.
\]

The second objective function \( f_2 \) is used for minimizing the length of the longest test case.

**TSO4-Minimizing the execution time.** We measure the execution time for each test case \( tc \) from the initial test suite \( TS \). Let us denote by \( time(tc) \) the execution time of \( tc \). Then the execution time of a test suite \( T \subset TS \) is \( \sum_{tc \in T} time(tc) \). In this problem the goal is to minimize the execution time of the test suites. The first objective and the constraint are the same as for TSO1 problem. The second objective function \( f_2 \) to be minimized is

\[
f_2(x) = \sum_{i=1}^{m} (x_i \cdot time(t_i)) \text{ (minimize the execution time)}.
\]

**TSO5-Maximizing the distribution quality.** In order to understand the problem proposed here, let us consider a simple example. Let be \( T_1 =
\{e_1 e_3 e_4, e_1 e_2, e_3 e_2 e_5\} and \(T_2 = \{e_2 e_2 e_4, e_1 e_2, e_3 e_3\}\) two test suites which cover the set of events \(E = \{e_1, e_2, ..., e_5\}\). The events \(e_1\) and \(e_2\) are executed an equal number of times in \(T_1\), while they are not in \(T_2\). We say that \(T_1\) has a better distribution quality. Therefore the goal is to obtain test suites with a good distribution of the events. This property is a practical requirement of users.

In the following, we propose an objective function which measures the distribution quality of a given test suite \(T \subset TS\). Let be \(TS = \{t_1, t_2, ..., t_m\}\) the initial test suite and \(E = \{e_1, e_2, ..., e_n\}\) the set of the events. Let be a matrix \(A\) which captures the events covered by each test case \(tc\) in \(TS\); the number of rows of \(A\) equals the number of events to be covered, \(n\), and the number of columns equals the number of test cases in the initial test suite, \(m\). Therefore the entries \((a_{ij})_{1 \leq i \leq n, 1 \leq j \leq m}\) of \(A\) are

\[
a_{ij} = \begin{cases} k, & \text{if } t_j \text{ covers } e_i \text{ by } k \text{ times} \\ 0, & \text{if } e_i \text{ is not covered by } t_j \end{cases}, 1 \leq i \leq n, 1 \leq j \leq m. 
\]

Let be \(x = (x_1, x_2, ..., x_m) \in \{0, 1\}^m\) the mathematical representation of the test suite \(T \subset TS\). We define the matrix \(D(x)\) to be

\[
D(x) = A \times \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_m \end{pmatrix}
\]

More exactly, \(D(x)\) is a vector of \(n\) components \(d_i(x), 1 \leq i \leq n\). From the definition, the entry \(d_i(x) = \sum_{k=1}^{m} (a_{ik} \cdot x_k)\) of \(D\) denotes the number of times the event \(e_i\) was covered by the test suite \(T\).

Now the mean amount of executions per event in \(T\) is exactly

\[
m_T(x) = \frac{1}{n} \sum_{i=1}^{n} d_i(x).
\]

If the test suite \(T\) has a good distribution of the events, we would expect \(d_i(x), 1 \leq i \leq n\) values to stay near the mean value \(m_T(x)\). Therefore in order to obtain a good distribution of the events we define the objective function to be minimized in the following manner:

\[
 f(x) = \frac{1}{n} \sum_{i=1}^{n} (d_i(x) - m_T(x))^2.
\]

Let us illustrate this definition on our simple example. We consider that \(TS = T_1 \cup T_2 = \{e_1 e_3 e_4, e_1 e_2, e_3 e_2 e_5, e_2 e_2 e_4, e_1 e_2, e_3 e_5\}\). Then, \(x_1 = 90\)
(1, 1, 1, 0, 0, 0) and $x_2 = (0, 0, 0, 1, 1, 1)$ are the mathematical descriptions of $T_1$ and $T_2$ respectively. Given that, the matrix $A$ will be

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 2 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

and

$$D(x_1) = A \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 1 \\ 1 \end{pmatrix}, \quad D(x_2) = A \times \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Further calculation shows that $f(x_1) = 0.24$ and $f(x_2) = 0.64$. Therefore the test suite $T_1$ has a better distribution of the events.

We formulate this problem as a constraint single-objective optimization problem and search for solutions which minimize $f(x)$ subject to

$$d_i(x) \geq 1, \; 1 \leq i \leq n \text{ (each event is covered at least one time)}.$$

**TSO6-Balancing the lengths while minimizing the longest path.**

Finally, we propose here to balance the lengths of the execution paths while we keep valid the two objectives of TSO3 problem (achieve event coverage while minimize the length of the longest path). Therefore this problem is a 3-objective test suite optimization problem. We search here for test suites which achieve event coverage by short and balanced execution paths. The third objective function can be mathematically formulated as below.

We remember that $\text{len}(tc)$ denotes the length of the test case $tc$. Let be $T \subset TS$ a test suite and $x$ its mathematical description. First, we define the mean of the lengths as

$$m_T^{\text{len}}(x) = \frac{1}{|T|} \sum_{i=1}^{m} (x_i \cdot \text{len}(x_i)).$$

If the test suite $T$ contains balanced execution paths, the $\text{len}(tc), tc \in T$ values will stay near the mean value $m_T^{\text{len}}(x)$. Given that, the third objective function to be minimized can be defined as
Table 4.8: Summarize the six test suite optimization problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Type</th>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO1</td>
<td>bi-objective</td>
<td>yes</td>
<td>Minimizing the size of the test suite</td>
</tr>
<tr>
<td>TSO2</td>
<td>bi-objective</td>
<td>yes</td>
<td>Minimizing the no. of the executed events</td>
</tr>
<tr>
<td>TSO3</td>
<td>bi-objective</td>
<td>no</td>
<td>Minimizing the longest execution path</td>
</tr>
<tr>
<td>TSO4</td>
<td>bi-objective</td>
<td>yes</td>
<td>Minimizing the execution time</td>
</tr>
<tr>
<td>TSO5</td>
<td>single-obj.</td>
<td>yes</td>
<td>Maximizing the distribution quality</td>
</tr>
<tr>
<td>TSO6</td>
<td>3-objective</td>
<td>no</td>
<td>Balancing the lengths + TSO3 problem</td>
</tr>
</tbody>
</table>

$$f_3(x) = \frac{1}{|T|} \sum_{i=1}^{m} (x_i \cdot (\text{len}(t_i) - m_T^{\text{len}}(x))^2)$$

We solve all these six test suite optimization problems using multi-objective evolutionary algorithms. In Table 4.8 we summarize the properties of our problems.

**Experiments** We provide now the results of a couple of experiments to verify the efficiency and effectiveness of the presented methods.

**Solution Encodings.** We chose two modern and widely used Pareto efficient genetic algorithms, NSGA-II and SPEA-2 [ZLT01, Din11]. When using evolutionary algorithms for solving a multi-objective test suite optimization problem, we must properly encode the possible solutions of the problem. Let be $T \subset TS$ a subset of the initial test suite $TS = \{t_1, t_2, \ldots, t_m\}$. We use the mathematical representation $x \in \{0, 1\}^m$ of $T$ (see Section 4.3.5) to encode the possible solutions. Therefore binary encoding is considered to be a natural representation for the possible solutions. The inclusion and exclusion of a test case within a subset of the initial test suite are represented by 1 and 0 respectively in a binary string (chromosome string).

**Subjects.** We conducted the experiments with a total of 5 test suite subjects of varying sizes and complexity levels. The test suites were generated from two industrial inspired Event-B models: the BepiColombo and SSFPilot models which are publicly available DEPLOY model repository\(^5\). The first 4 machines are different levels of refinements of BepiColombo project and the last machine is the high level of abstraction of SSFPilot model. The sizes of the machines are listed in Table 4.9. Moreover, the test suite generated from these Event-B models were obtained using the MBT plugin\(^6\) available on the web.

\(^5\)http://deploy-eprints.ecs.soton.ac.uk

\(^6\)http://wiki.event-b.org/index.php/MBT_plugin
Table 4.9: Sizes of five test suite subjects generated from two industrial inspired models (number of events, size of test suites and maximum length of test cases).

<table>
<thead>
<tr>
<th>Subject</th>
<th>No. of events</th>
<th>Size of TS</th>
<th>Max. size of tcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BepiColombo_M0</td>
<td>5</td>
<td>40</td>
<td>7</td>
</tr>
<tr>
<td>BepiColombo_M1</td>
<td>10</td>
<td>170</td>
<td>7</td>
</tr>
<tr>
<td>BepiColombo_M2</td>
<td>12</td>
<td>256</td>
<td>7</td>
</tr>
<tr>
<td>BepiColombo_M3</td>
<td>16</td>
<td>240</td>
<td>7</td>
</tr>
<tr>
<td>SSFPilot_TCTM</td>
<td>13</td>
<td>786</td>
<td>8</td>
</tr>
</tbody>
</table>

for Rodin. The test generation algorithm is based on [Ipa11].

The two Event-B models are summarized below:

• **BepiColombo**: This is an abstract model\(^7\) of two communication modules in the embedded software on a space craft. The Event-B model was proposed for formal validation of software parts of BepiColombo mission to Mars\(^8\). The model has different levels of refinements. In the abstraction, \(M_0\), the main goal of the system is modeled. The details of the system are added through three refinement levels, \(M_1\), \(M_2\) and \(M_3\). The modeling approach starts on the first level with 5 set-type variables and 5 events and ends up with 18 variables and 16 events.

• **SSFPilot**: This is an Event-B model\(^9\) of a pilot for a complex onboard satellite mode-rich system: Attitude and Orbit Control System (AOCS). In [ITL\(^{+}10a\)] the authors present a formal development of an AOCS in Event-B modeling language. They show that refinement in Event B provides the engineers with a scalable formal technique that enables both development of mode-rich systems and proof-based verification of their mode consistency.

**Results.** The test suite optimization techniques attempt to reduce the test suite cost w.r.t. a given coverage criterion (event coverage in our case). Given that, the percentage reduction will be used as a measure for comparative analysis. To increase the confidence, we compare the results produced by the two algorithms: NSGA-II and SPEA-2.

We have used the multi-objective evolutionary algorithm framework jMetal [DNA10] for our experiments. The two algorithms were configured with pop-

\(^7\)http://eprints.ecs.soton.ac.uk/22048/5/Rodin_Space_Craft.zip


\(^9\)http://deploy-eprints.ecs.soton.ac.uk/58/
Table 4.10: TSO1. Average reduced sizes for optimized test suite $T$.

<table>
<thead>
<tr>
<th>Subject</th>
<th>NSGA-II</th>
<th>SPEA-2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f_2(x_{TS})$</td>
<td>Avg $f_2(x_T)$</td>
</tr>
<tr>
<td>BepiColombo_M0</td>
<td>40</td>
<td>1.03</td>
</tr>
<tr>
<td>BepiColombo_M1</td>
<td>170</td>
<td>7.59</td>
</tr>
<tr>
<td>BepiColombo_M2</td>
<td>256</td>
<td>28.87</td>
</tr>
<tr>
<td>BepiColombo_M3</td>
<td>240</td>
<td>26.14</td>
</tr>
<tr>
<td>SSFPilot_TCTM</td>
<td>786</td>
<td>228.42</td>
</tr>
</tbody>
</table>

Table 4.11: TSO2. Average reduced number of executed events for optimized test suite $T$.

<table>
<thead>
<tr>
<th>Subject</th>
<th>NSGA-II</th>
<th>SPEA-2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f_2(x_{TS})$</td>
<td>Avg $f_2(x_T)$</td>
</tr>
<tr>
<td>BepiColombo_M0</td>
<td>252</td>
<td>8.02</td>
</tr>
<tr>
<td>BepiColombo_M1</td>
<td>1300</td>
<td>65.09</td>
</tr>
<tr>
<td>BepiColombo_M2</td>
<td>1977</td>
<td>224.42</td>
</tr>
<tr>
<td>BepiColombo_M3</td>
<td>1873</td>
<td>204.77</td>
</tr>
<tr>
<td>SSFPilot_TCTM</td>
<td>6554</td>
<td>1897.79</td>
</tr>
</tbody>
</table>

ulation size of 100. The archive size of SPEA-2 was set to the same value, 100. The stopping criterion is to reach the maximum number of generation which was set to 100. The both algorithms use the following genetic operators: the binary tournament selection operator, the single point crossover operator with probability of 0.9 and the single bit-flip mutation operator with the mutation rate of $1/m$ where $m$ is the length of the bit-string (i.e. the size of the initial test suite).

For each test suite subject, each optimization problem and each algorithm, 100 independent runs were performed. The results are presented in Tables 4.10-4.15. To compare the results, we computed for each problem the specific objective function values for the initial test suite. For example, the column $f_3(x_{TS})$ from the Table 4.15 indicates the values of the third objective function of the problem TSO6 when computed for the initial test suite $TS$. Otherwise, in each table, the average values of specific objective functions of the solutions are indicated. As shown in the tables, the results of the two algorithms are comparable. We obtained high values for the percentage reduction of test suite because of the simplicity of the event coverage criterion.
Table 4.12: TSO3. Average length of the longest path of optimized test suite \( T \).

<table>
<thead>
<tr>
<th>Subject</th>
<th>NSGA-II</th>
<th>SPEA-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BepiColombo_M0</td>
<td>4.69</td>
<td>4.84</td>
</tr>
<tr>
<td>BepiColombo_M1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>BepiColombo_M2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>BepiColombo_M3</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>SSFPilot_TCTM</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.13: TSO4. Average execution time (in seconds) of optimized test suite \( T \).

<table>
<thead>
<tr>
<th>Subject</th>
<th>( f_2(x_{TS}) )</th>
<th>Avg ( f_2(x_T) )</th>
<th>Avg%</th>
<th>Avg ( f_2(x_T) )</th>
<th>Avg%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BepiColombo_M0</td>
<td>4.6</td>
<td>0.13</td>
<td>97.07</td>
<td>0.14</td>
<td>96.95</td>
</tr>
<tr>
<td>BepiColombo_M1</td>
<td>48.43</td>
<td>1.88</td>
<td>96.11</td>
<td>2.16</td>
<td>95.54</td>
</tr>
<tr>
<td>BepiColombo_M2</td>
<td>130.16</td>
<td>12.39</td>
<td>90.48</td>
<td>13.40</td>
<td>89.70</td>
</tr>
<tr>
<td>BepiColombo_M3</td>
<td>204.28</td>
<td>20.43</td>
<td>89.99</td>
<td>22.13</td>
<td>89.16</td>
</tr>
<tr>
<td>SSFPilot_TCTM</td>
<td>197.80</td>
<td>50.78</td>
<td>74.32</td>
<td>51.38</td>
<td>74.02</td>
</tr>
</tbody>
</table>

Table 4.14: TSO5. Average distribution quality of optimized test suite \( T \).

<table>
<thead>
<tr>
<th>Subject</th>
<th>( f(x_{TS}) )</th>
<th>Avg ( f(x_T) )</th>
<th>Avg%</th>
<th>Avg ( f(x_T) )</th>
<th>Avg%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BepiColombo_M0</td>
<td>520.24</td>
<td>0.16</td>
<td>99.96</td>
<td>0.16</td>
<td>99.96</td>
</tr>
<tr>
<td>BepiColombo_M1</td>
<td>8771.4</td>
<td>17.03</td>
<td>99.80</td>
<td>22.45</td>
<td>99.74</td>
</tr>
<tr>
<td>BepiColombo_M2</td>
<td>19840.90</td>
<td>238.98</td>
<td>98.79</td>
<td>270.26</td>
<td>98.63</td>
</tr>
<tr>
<td>BepiColombo_M3</td>
<td>14432.43</td>
<td>169.14</td>
<td>98.82</td>
<td>191.42</td>
<td>98.67</td>
</tr>
<tr>
<td>SSFPilot_TCTM</td>
<td>166187.40</td>
<td>13251.76</td>
<td>92.02</td>
<td>13667.67</td>
<td>91.77</td>
</tr>
</tbody>
</table>

Table 4.15: TSO6. Average balancing values of the lengths of optimized test suite \( T \).

<table>
<thead>
<tr>
<th>Subject</th>
<th>( f_3(x_{TS}) )</th>
<th>Avg ( f_3(x_T) )</th>
<th>Avg%</th>
<th>Avg ( f_3(x_T) )</th>
<th>Avg%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BepiColombo_M0</td>
<td>2.16</td>
<td>0.00</td>
<td>100</td>
<td>0.00</td>
<td>100</td>
</tr>
<tr>
<td>BepiColombo_M1</td>
<td>1.81</td>
<td>0.21</td>
<td>88.27</td>
<td>0.22</td>
<td>87.52</td>
</tr>
<tr>
<td>BepiColombo_M2</td>
<td>1.57</td>
<td>0.33</td>
<td>78.41</td>
<td>0.34</td>
<td>77.87</td>
</tr>
<tr>
<td>BepiColombo_M3</td>
<td>1.62</td>
<td>0.36</td>
<td>77.76</td>
<td>0.37</td>
<td>77.04</td>
</tr>
<tr>
<td>SSFPilot_TCTM</td>
<td>2.21</td>
<td>1.15</td>
<td>47.96</td>
<td>1.17</td>
<td>47.05</td>
</tr>
</tbody>
</table>
Conclusions In this subsection the multi-objective test suite optimization problem for Event-B testing was introduced. Different optimization criteria were proposed and the resulted problems were solved using two modern multi-objective evolutionary algorithms. For all optimization problems the considered test adequacy criterion was the event coverage. All our optimization problems can be easily formulated in a more general framework: a test suite $T$ must meet a set of $n$ requirements $\{r_1, r_2, ..., r_n\}$ to provide the desired 'adequate' testing of the model. We will consider in the future more complex coverage criteria.

4.4 Results

4.4.1 Appropriateness of Enhanced Deployment Strategy

For this pilot we adopted the deployment strategy reported in deliverable D4.2: not making use of Event-B “as is”, but to base all modelling on existing domain-specific model types and translations to Event-B. This imposes additional effort when mapping back results of the model checker or theorem prover to the original modelling language.

However our deployment strategy allows developers to create and maintain modelling content without a deep understanding of formal methods. The Event-B language as well as the utilized concepts and tools embedded in Rodin are completely hidden from our users. Focussing on scenario testing as the main business case has further been a beneficial decision, as this is the development activity where software vendor’s investment steadily grows.

In order to increase the chance of productive adoption of the project results we continuously partnered with internal development units and prioritized our effort according to their requirements. Even though this meant that improving usability of the prototypes became as important for us as investigating advanced concepts and delivering missing functionality, we were able to convince our stakeholders that the created tools reached the necessary technical maturity for a productive use.

4.4.2 Suitability of Business Process Modelling as Enhanced Pilot Application

Choosing business process modelling for the enhanced pilot application has been suitable. Even though the Business Process Modeling Language BPMN, tends to have quite complex constructs (e.g. a huge number of gateways
with different semantics) we decided to focus on realistic use cases from the scenario testing domain. These are of medium-size complexity, utilizing a subset of elements only, so that we could transform, check, verify, and utilize them for test generation with realistic efforts.

A disadvantage however is that our developed solution does not fully comply with BPMN and therefore its application is limited to SAP’s context. However, the concepts and tools have been created such that they can be generalized with reasonable effort.

4.4.3 Suitability of Event-B for the Enhanced Pilot

As for the previous pilot deployment, 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 especially the process modelling for testing purposes, where the test steps also have precondition and actions.

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.

As the absence of records in the language is solved, the necessity of modelling records by total or partial functions, which increases the complexity and decrease the readability of obtained models could be avoided. Further, the work on flow seems to improve the description of sequential behavior, which is a common characteristic of BPMN, and consequently its validation.

4.4.4 Suitability of Rodin for the Enhanced Pilot

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

During the first piloting, a major problem in Rodin was the maturity of the automated provers. The addition of the relevance filter, which uses heuristics to find useful hypothesis, increases the number automatically discharged proof obligations dramatically.

Even though Rodin is a well-developed, stable and simple-to-use application, the large footprint of the Eclipse framework hinders its deployment inside of a web-service. Fortunately the ProB plugin was also made available as a standalone tool, so that we could realize our envisioned client-server architecture for a seamless integration into the currently used test framework, as described in Section 4.3.
4.4.5 Probability of Adoption for Enhanced Pilot

During the development of our model-based scenario testing solution we were able to integrate our prototypes with productively used systems. With the support of senior management we were therefore able to evaluate our work on realistic use cases, and utilizing development resources. Since the evaluation activities have not been finalized and estimated yet, it is not possible for us to draw final conclusions.

However we can report that in the recently finished stage of piloting, 6 use cases (i.e. business scenarios) have been investigated by professional developers. For each case, a process model incorporating various process variants has been created, enriched with test specific information, and successfully utilized for scenario test generation. In oral experience reports to senior management all users evaluated our solutions “mature enough for productive use”. This was further confirmed by the absence of high priority requirements and feature requests. On the contrary a large number of low priority requests, mostly concerning the enhancement of usability, indicated the strong engagement of the users.

As a result, the decision for a phased roll-out starting in November was made. It is further planned to hand over the responsibility for the tool development and maintainence to the development units during this period. All this indicates that the deployment of the research results of DEPLOY has been successful and will sustain the projects termination.
Chapter 5

Evidence Consolidation

During the enhanced deployment new contributions were identified to enrich industrial FAQ answers (Frequently Asked Questions) with additional material further showing evidence on the usefulness of formal method in the e-Business sector. This new evidence material was jointly discussed between CETIC and SAP. This chapter summarised the main contributions from SAP’s work related to training/technology transfer, reuse, controlling the impact of formalism through domain specific notations and exploiting models for testing.

5.1 Training, transfer and human resources

TSP-HM-1 - What is the cost or effort needed to train engineers/analysts to use a new formalism, taking into account their previous experience with formal engineering methods? Most of the effort of transferring a formal method to new staff can be avoided when hiding the formal method behind tools already used by the targeted users. At SAP, the formal aspects are hidden behind the tool already in use by the target team. Users don’t really need to know that a formal method is used in the background. Consequently, the effort for training can be saved. On the other hand, there is a need to hide the formal methods completely so that no guidance is required from the user or no error related to the formal method is reported at the error level. The effort to hide the formal method can be quite significant. For example, for SAP it is required to achieve high level of proof automation for the checks and to produce user level explanations from the output of the formal tools used. However in the long term if a significant amount of staff members use the tool, the additional effort of hiding the formal method is largely compensated by savings on training.
New FAQ: MF-HM-2 - What strategy increases the chance of adoption of formal methods? We sketch here the approach used by SAP research to increase the chance of internal adoption of the project results. It relies on a partnership with internal development units throughout the integration of a formal method (e.g. collecting requirements, early feedback, conducting validation experiments). In other words, the use of a formal method was motivated by a need from the development units. Internal instruments for this kind of partnership are available internally at SAP so has resources can be devoted to it. The prioritization of the R&D effort was then driven by the development team's requirements priority. A consequence was that improving usability of the prototypes became as important as investigating advanced concepts and delivering missing functionality. Using this strategy, SAP research was able to convince staff from development units that the resulting tools had reached the necessary technical maturity for a productive use.

5.2 Reuse

R-EA-1 - When using a formal method efficiently, does it become more natural to design generic, reusable components than when using non formal methods? SAP encountered this effect in both pilot deployments. For instance, SAP was able to leverage not only their experience from the first piloting phase, but also concrete concepts and implementations where applicable. As described earlier in this deliverable, for the modelling of business processes, SAP targeted an automated model transformation and verification approach in order to hide complexity related to Event-B from the user. SAP was then able to reuse their previous work on formal property checking (e.g. deadlock freedom) as reported in deliverable D4.1 [FD] and to further enhance it (e.g. checking for data consistency).

New FAQ: Is the formalism providing efficient reuse mechanisms? At domain level, SAP is using BPMN-like notation for which a number of patterns have been published (e.g. Business Patterns in UML, Event Processing patterns). At formal level, SAP is using Event-B has underlying formalism. Event-B patterns are a means of expressing reusable modelling structures and managing effort by promoting proof re-use. Pattern mechanisms have been developed during the DEPLOY project (WP8). In Section 4.2 SAP presents how such patterns provide a controlled way of extending an Event-B development with a pre-validated refinement step. Given a proven refinement, any application of the pattern results in a new, fully-proven refinement step, in
other words, following a refinement pattern makes it possible to reuse a proof tactic.

**New FAQ: Are the tool efficiently supporting reuse?** During the pilot deployment, reusing Event-B proofs was criticised as requiring a lot of manual replay work that could be automated. During the SAP enhanced deployment, a specific Rodin plugin supporting automated proof when reusing refinement patterns has been developed and a release is now available.

**G-HM-1 - What are the main benefits and risks when using formal methods in product development?** When one introduces a new language and paradigm, there is always a risk of non- adoption. In particular, switching from a specialised and established domain-specific language to a generic purpose language built on mathematical syntax is typically considered not to be feasible in Industry. SAP’s approach to decrease the risk of non-adoption was to hide Event-B from developers while still benefiting from Event-B’s proofing capabilities in their development of business applications. As demonstrated by the transfer of the tool from SAP research to SAP development units, the hiding of a formal method behind existing tools proposes an efficient approach for decreasing the risk related to transferring formal methods to Industry. This transfer was possible at SAP because the domain specific language already used at SAP was semantically rich enough to make the transformation to Event-B models possible.

### 5.3 Control Impact of Formalism: DSL and hidden formal methods

**CIF-PQAM-2 - What are the risks of hiding the use of formal methods and what are the strategies to mitigate them?** According to SAP’s experience, the main risk is that people don’t understand how responses of their tools are computed. SAP mitigated this risk by investigating a lot into understanding user’s expectations and tailoring the formal methods accordingly (see training and technology transfer FAQ)

**CIF-EA-1 - How much of the use of a formal method can be automated, given the available tooling in 2011?** During the first DEPLOY pilot, SAP could reach high score of 70% of proof obligations that could be discharged automatically. For applying automated checking tool, this figure means that a significant number of properties cannot be checked. During
the enhanced pilot, automation improved thanks to progress made by the provers (availability of the relevance filter for selecting adequate hypotheses) and the development of optimisations. A set of real-life software models from the SAP was used as benchmark. For consistency checks, 90% of the proofs could be automatically discharged. Application specific properties could also be checked but they require the expert knowledge to introduce auxiliary invariants. In addition to proof-based approach, model-checking using ProB could also be applied to the same set of real-life software models. ProB could complete the checking in a few seconds. However this required to set bounds (e.g. to integer) and thus restrict the portion of the state space being searched, thus not providing the total assurance that the considered property holds.

5.4 Exploiting Models: Model-based Testing

EM-EA-1 - Is it possible to take advantage of formal models beyond using them to guarantee certain properties of a system, for example, to automate or simplify certain development and QA tasks? SAP used the same set of models for proving properties and for generating test cases. Test generation required the model to be annotated with additional test-specific information. However, it was not necessary to write totally different kinds of models, meaning that the effort of building models could be shared between verification and test generation activities. Evolving the system will also not require to maintain two different artefacts. This means productivity is positively impacted.

EM-QAP-1 - Does the use of formal engineering methods help in the design of tests? SAP developed model-based testing suites aiming at generating test cases from formal models, enabling various coverage criteria (activity coverage, flow coverage, extended flow coverage). In addition test reduction techniques were also developed to reduce the size of the test suite while preserving its coverage. The models input by the test case generators were used for proof-related activities beforehand. Expert estimations implied that automatic test generation reduces the overall test effort significantly.

5.5 Other relevant FAQ items

In this section we summarise more punctual, yet useful, contributions to various questions.
TOOL-HM-3 - Are there qualification/certification constraint related to the introduction of a tool in my development chain? SAP reported that no such constraint was encountered during the DEPLOY project for business information sector. There is no specific certification scheme for business information software, however SAP has its own internal quality standards.

ExFac-HM-2 - In what sectors or in what industry is the culture strongly pushing or enforcing the use of formal methods? According to SAP, the business sector is moving towards on-demand products, which demand instant quality. Formal methods are perceived as beneficial towards this goal, so that SAP devoted research and development time in this direction at their R&D centre in Darmstadt. It is also aligned with the model-driven approach of their “ByDesign” development environment.

G-HM-2 - Questions: Why have formal methods failed to breakthrough on the market for such a long time? Customer demand is the general driver. In the business sector, customers are not requiring this kind of techniques. Sector trend might influence on customer demand, and it seems that there is an increasing trend to believe that formal methods are able to provide high quality solutions.
Chapter 6

Conclusion

6.1 Summary

In this document we reported on our activities towards the enhanced deployment of Formal Methods developed in the EU-FP7 project DEPLOY in the business sector. Our approach throughout the project was to make business application developers benefit from formal verification techniques by hiding 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.

The deliverable D4.2 summarized the previous deployment activities and described the remaining activities on the first pilot on service choreography that were conducted after publishing deliverable D4.1 [FD].

Reviewing and analysing the lessons learnt form this first pilot, we decided to center our efforts towards the enhanced pilot deployment on business process modeling. The work conducted in this context was again devided into activities supporting the modeling of domain-specific languages and consequent translation into Event-B, as well as the utilization of the Rodin toolset for quality assurance during software development. This strategy of enabling users to benefit from the hughe variety of formal methods and concepts without forcing them to learn another language or tool, paid off nicely and convinced our pilot users from the benefits for a productive use.

6.2 Remaining Activities

We plan to devote most of the remaining project time on finalizing the piloting and roll-out activities. This includes the training of users, capturing of
usage data, bug-fixing and improvement of documentation. Furthermore, we are involved in hand-over activities, which have the motivation to transfer knowledge and code to the development units in order to assure maintainence and support once the project terminated. Besides these activities, we plan to continue publishing and demonstrating the project results in scientific conferences and industrial fares.
Bibliography


[BBMP08] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Towards automated WSDL-based testing of web services.


[OMGb] OMG. Omg unified modeling language (omg uml), superstructure version 2.2.


