Applying Model Checking to Generate Model-Based Integration Tests from Choreography Models

Sebastian Wieczorek[1], Vitaly Kozyura[1], Andreas Roth[1], Michael Leuschel[2], Jens Bendisposto[2], Daniel Plagge[2], and Ina Schieferdecker[3]

[1] SAP Research, CEC Darmstadt,
Bleichstr. 8, 64283 Darmstadt, Germany
firstname.lastname@sap.com
[2] University of Düsseldorf,
Universitätsstrasse 1, 40225 Düsseldorf, Germany
lastname@cs.uni-duesseldorf.de
[3] Fraunhofer Institute for Open Communication Systems (FOKUS),
Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
ina.schieferdecker@fokus.fraunhofer.de

Abstract. Choreography models describe the communication protocols between services. Testing of service choreographies is an important task for the quality assurance of service-based systems as used e.g. in the context of service-oriented architectures (SOA). The formal modeling of service choreographies enables a model-based integration testing (MBIT) approach. We present MBIT methods for our service choreography modeling approach called Message Choreography Models (MCM). For the model-based testing of service choreographies, MCMs are translated into Event-B models and used as input for our test generator which uses the model checker ProB.

Keywords: Model Checking, Model-based Testing, Formal Methods, Integration Testing, Service Choreography Models.

1 Introduction

Service choreography models play an important role in SOA development and can provide a basis for ensuring quality at several levels, e.g., through verification and testing. In previous work [25], we defined precise requirements on choreography modeling languages that would allow supporting three software quality related development methods: design, verification, and testing. However, we observed that state of the art choreography languages such as WS-CDL [15] or BPMN [1] do not fulfill all these requirements simultaneously, mainly due to high abstraction level, imprecise semantics, assumption of ideal channels, lack of termination symbols, etc. Therefore SAP Research developed a language for modeling service choreographies called Message Choreography Modeling (MCM) and an Eclipse-based editor for it. In [26], we introduced MCM and provided an overview of the implemented editor and of its verification and testing plugins. In this paper, we present the model based testing (MBT) approach for service integration testing utilizing MCMs in detail.

As MCMs are based on communicating extended finite state machine (EFSM) semantics, constraint solving techniques have to be applied for the automatic test generation. Therefore, we translate the models to Event-B [4] which can be processed by the model checker ProB [17]. Using ProB, we are able to generate test suites for service integration that are not only covering all transitions of the communication protocols described in the MCMs, but also optimize the test generation towards minimizing the effort of test concretization (e.g. test data provisioning) and execution. We use CSP [12] process algebra expressions, synchronized with the Event-B models, to encode concurrent aspects of the test case generation algorithm. In this paper we aim to show that MBT and formal methods can be applied in an industrial context and explain the practical considerations that have to be made (e.g. model coverage, test suite optimization criteria).

The remainder of this paper is structured as follows. Section 2 briefly introduces the running example for this paper and explains the necessary steps of our MBIT approach. In Section 3, the formal MCM syntax is given as a basis for the translation of MCM to Event-B in Section 4. Section 5 describes the implementation details of the test generation algorithm and Section 6 discusses related test generation approaches. Section 7 concludes the paper and gives future work directions.

2 Overview

Our approach to model-based integration testing (MBIT) comprises the modeling of the conversation between SOA components, the translation of the obtained models and the subsequent generation of test suites. In order to illustrate the approach, we first introduce the following running example, which will be referred to throughout the paper.

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* by the seller. The buyer afterwards has the choice either to send a *Cancel* that rolls back the previous communication and allows to restart the negotiation or to send an *Order* that successfully concludes the ordering process. Because we assume a (reliable) communication channel that is not necessarily preserving the message order, it might be observed that a *Cancel* is delivered after a new negotiation process already started.

2.1 MCM Modeling

The service choreography modeling language MCM complements the structural information of the communicating components (e.g. service interface descriptions and message types) with information on the message exchange between them. A detailed discussion of the underlying concepts of MCM and how they support service development can be found in [26]. MCM consists of different model types each defining different aspects of service choreographies.

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

- **Local Partner Model.** The local partner models (LPMs) specify the communication-relevant behavior for exactly one participating service component. Due to the design process of MCM, each LPM is a structural copy of the GCM with extra constraints on some of the local transitions, usually leading to the affected sending actions being deactivated.
- **Channel Model.** The channel model (CM) describes the characteristics of the communication channel on which messages are exchanged between the service components. These characteristics determine for example whether messages sent by one component preserve their order during transmission and are formalized by the WS-RM standard [23].

Figure 1 shows how the example described above can be described using the MCM artifacts. In the GCM at the top of Figure 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 *Ordered* and *Start* are so-called target states (thus connected with the filled circle). Only in these states, the communication between the partners is allowed to terminate.

➡ Buyer ⬅ Seller ⬆ Send ⬇ Receive

Fig. 1. GCM (top) of the choreography and LPMs of the buyer (left) and the seller (right)

[1] Deprecated here means that the message is out-dated and no-longer relevant as the negotiation has been restarted.

To keep the model deterministic, a set variable called *ID_SET* is declared and initialized with Ø. It stores the transaction ids from the header of *Request* messages that have not yet been addressed by *Cancel*, *Cancel (deprecated)* or *Order* messages (the headers of these messages also store the ids). Whenever a *Request* interaction takes place, an assignment *ID_SET := ID_SET* $\cup$ *{msg.Header.ID}* is executed referring to the *ID* stored in the header of the *Request* message. This assignment is needed to distinguish between a deprecated and an actual *Cancel* in state *Reserved*. Thus for the interaction *Cancel* an additional necessary precondition *ID_SET \ {msg.Header.ID})* = $\emptyset \wedge$ *msg.Header.ID* $\in$ *ID_SET* can be modeled in MCM while for *Cancel(deprecated)* we add the precondition *(ID_SET \ {msg.Header.ID})* $\neq \emptyset \wedge$ *msg.Header.ID* $\in$ *ID_SET*. In Section 3 the formal syntax and the complete set of preconditions and assignments for our example is described.

The LPM of the buyer partner of our example is depicted in the lower left part of Figure 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 (for details see Section 3 and 4). The LPM of the seller is depicted in the lower right part of Figure 1.

2.2 Transformation

Our goal is to generate test cases automatically from MCMs. FSM-based approaches [5,6,11] would be applicable for the test generation if the annotated constraints of the model have no impact on the communication behavior. In our example however this is not the case (e.g. the message *Cancel(deprecated)* is active only if at least two IDs have been stored in the variable *ID_SET*). As the example incorporates a quite common pattern of the enterprise software domain, approaches that are able to compute constraint-compliant paths have to be used for MCM based test generation.

An analysis showed that implementing a tool for the test generation that directly runs on MCM models from scratch would be inefficient and hence infeasible. Therefore, we decided to transform our models to the formal modeling language Event-B. Event-B [4] is an evolution of the B-Method [1] that puts emphasis on a lean design. In particular, the core language of Event-B is (with a few exceptions) a subset of the language used in its predecessor.

Event-B fits quite naturally to MCM: interactions can be seamlessly expressed as events and the relationship between GCM and LPMs can be formulated as Event-B refinement (although we use this technique in our transformation, it is not substantial to understand the test generation and therefore left out of scope for this paper). Also other formalisms, such as UML [20] have also been successfully translated into Event-B, so that we were able to utilize past experiences and practices.

Another distinguishing aspect is the tool support in form of the Eclipse-based Rodin tool [3]. Due to the extensible architecture, various plugins for Rodin exist. The tool can be integrated with other Eclipse-based tools such as the MCM editor. With ProB, a flexible model checker for Event-B models exists that can be utilized for the test generation and enables us to build on the previous experiences with B and model checking in the area of MBT.

Apart from deriving tests, a transformation into Event-B opens a variety of possibilities to analyze the model: e.g. checking the refinement relation ensures the local enforceability of the service choreography. Though being an important part of the overall MCM approach, also formal model analysis is not in the scope of this paper.

2.3 Test Generation

After having obtained a formal representation of the MCM model, we can employ a model checker to derive a test suite for integration testing. Similar to [24], we define integration testing as testing of an assembly of individually already tested components. Because of the confidence about the correctness of the participating components (which results from quality ensuring techniques on the component level, e.g. unit tests), our testing approach focuses on showing that each sent message is interpreted in the correct way by the receiver. This can be determined by checking for each interaction, that the intended message effect has been caused. Consequently, a test suite should cover all receive events modeled in the LPMs.

For automatic test generation, a local model that incorporates information from both LPMs and the CM (to connect the send and receive events) can be used. Because various cases studies (e.g. [10]) show that state space explosion is the major stumbling point when applying automatic test generation to industrial settings, we decided to use the GCM to drive the test generation instead of the much more complex local model. While transition coverage of the GCM is equivalent to receive event coverage of the LPMs in most cases, the state space that needs to be explored is significantly lower.

In [27], we discussed possible coverage criteria that can be used to drive service integration testing and how to choose them accordingly depending on effort and fault assumptions. For this approach, we decided to use transition coverage, i.e. that all interactions are contained in the test suite, because it already uncovers a significant amount of integration faults with relatively small efforts [22]. For example in the MBIT approach of [6], transition coverage of a global communication model was able to detect about 90% of integration related faults.

Important from an industrial perspective is, that our approach further aims to be optimal regarding the minimization of the effort in the subsequent test concretization (e.g. provisioning of test data), test execution and test analysis phases. Based on practical experience of the testing process at SAP [28], we concluded that optimal corresponds to the following list of objectives which is sorted from highest to lowest priority:

1. *Each path should start in the initial state and end in a target state:* As described in [28] setting system states in test preambles is complicated and time consuming. Stopping a test while the system is not in a target state leads to problems with inconsistent data that might hamper consequent test executions.
2. *The length of the longest generated path should be minimal:* The longer a test case gets, the harder it is to maintain. Therefore especially for generated tests a top priority is to carefully control path lengths.

3. *Message racing should be minimal:* Testing the effects that message racing has on the interaction is an important part of each test suite. Tests are mostly carried out in rather idealistic environments where messages are received in the same order they have been sent. Therefore, during test execution, message racing has to be emulated on the channel in a controlled way, usually leading to much higher effort.
4. *The number of test steps should be minimal:* As the effort increases with the overall length of all test cases, the sum of test steps should be minimized.

Section 5 describes the resulting steps of the test generation and their implementation, namely the generation of global test cases, the mapping to local test cases and the test suite optimization.

3 MCM Syntax

In this section, we present the abstract syntax of MCM, which is the basis for the translation into Event-B and the subsequent test generation.

For a simplified presentation, we assume that all choreographies consist of exactly two participating components. Then, a message choreography model $MCM=(GCM, LPM_1, LPM_2, CM)$ consists of a global choreography model (*GCM*), two local partner models (*LPM_1 and LPM_2*) and a channel model *CM*.

Global Choreography Model. The *GCM* is based on a finite state machine $L=(S, I, \Rightarrow)$, where *S* is a finite set of states, *I* is a finite set of interactions and $\Rightarrow \subseteq \mathbb{P}(S) \times I \times S$. The system has an initial state $init \in S$ and target states $\{e_1,\ldots,e_n\}$, where $e_i \in S$.

Below we define the language used for additional guards and actions of the GCM. Since the additional guards and actions refer to entries in the exchanged (XML) messages, we define record types representing the schemas the messages comply with. A finite set *ET* of elementary types (e.g. including the natural numbers) and a finite set of labels *F* are given. For these, the set *T* of record *types* is inductively defined to be the smallest superset of the *elementary types ET*, the set CT of *complex types* $\{(f, t) \mid t \in T, f \in F\} \in CT$, and the set of *set types Set(t)* with $t \in T$. Further each $t \in T$ has a unique assigned name *name(t)* from a set of data type names.

Each interaction $i \in I$ is then assigned to a type $itype(i) \in T$. Further we assume a set V_t of variables of type $t \in T$. For each interaction $i \in I$ there is a special variable $msg_i \in V_{itype(i)}$ referring to the message exchanged during an interaction. Furthermore we define a set C_t of constants (including e.g. $0,1,2,\ldots$, or $\emptyset$) of type $t \in T$.

The set $Term_t$ of terms for $t \in T$ is defined as the smallest set with

- $V_t \cup C_t \subseteq Term_t$ and
- $s.f \in Term_t$ with $(f, t) \in ct$ for some complex data type $ct \in CT$ and $s \in Term_{ct}$
- $s_1 + s_2 \in Term_t$ with $t=\mathbb{N}$, $s_1, s_2 \in Term_{\mathbb{N}}$ (analogous for other arithmetic operations)
- $s_1 \cup s_2 \in Term_t$ with $t=Set(T)$, $s_1, s_2 \in Term_{Set(T)}$ (analogous for other set operations)

The set *Term* consists of the union of $Term_t$ over all $t \in T$. The set *Form* of formulae is the set of first order formulae over *Term,* the predicates $\{=,<,>,\in, \setminus, \cup\}$ and the variables V (respecting typing in an obvious manner).

A global choreography model for a set of data types T is a tuple $GCM=(L, V, C, itype, pre, act)$ with preconditions $pre: I\rightarrow Form$ and actions $act: I \rightarrow (V \nrightarrow Term)$, where $V \nrightarrow Term$ is a partial function with $act(i)(v) \in Term_t$ and $v \in V_t$. The formulae of a precondition and the terms of actions of an interaction i must not contain variables $msg_{i'}$ with $i \neq i'$. If clear from the context we thus just write *msg* instead of msg_i.

Example. As explained in Section 2, the GCM of our example has the following variables, preconditions, and actions:

$V = \{ID\_SET\}$

$pre(Request) = msg.Header.ID \notin ID\_SET$	$act(Request)\ (ID\_SET)=ID\_SET\ \cup \{msg.Header.ID\}$
$pre(Order) = msg.Header.ID \in ID\_SET$	$act(Order)\ (ID\_SET)=ID\_SET \setminus \{msg.Header.ID\};$
$pre(Cancel) = ID\_SET \setminus \{msg.Header.ID\}) = \emptyset \wedge msg.Header.ID \in ID\_SET$	$act(Cancel)\ (ID\_SET) = \emptyset;$
$pre(Cancel(deprecated)) = (ID\_SET \setminus \{msg.Header.ID\}) \neq \emptyset \wedge msg.Header.ID \in ID\_SET$	$act(Cancel(deprecated))\ (ID\_SET) = (ID\_SET \setminus \{msg.Header.ID\})$

Local Partner Model. LPM_1 and LPM_2 are obtained from the *GCM* by duplicating, for each of them, the states and the global variables. Moreover each interaction $i \in I$ is transformed into the corresponding element from $PI=\{send\_i, receive\_i \mid for\ all\ i \in I\}$. The elements from *PI* inherit types, states, preconditions and actions from elements from *I*. *LPMs* can be further extended with an additional inhibitor function $inhib: I\rightarrow \mathbb{P}(S)$ which describes that the partner must not *send* a message associated with *I* if it is in one of the states *inhib(i).*

Example. From the interaction *Request* in GCM, we obtain *send_Request* in LPM_1 and *receive_Request* in LPM_2. LPM_1 contains a set $V_1 =\{ID\_SET_1\}$ and *pre* and *act* of the LPMs are adapted accordingly (w.r.t. GCM), e.g.:

$pre(send\_Request) = msg\_1.Header.ID \notin ID\_SET_1$
$act(send\_Request)\ (ID\_SET_1) = ID\_SET_1\ \cup \{msg.Header.ID\}$

In order to disallow for *send_Request* in the state *Reserved*, we set *inhib(send_Request)={Reserved}.*

Channel Model. Let us consider a set of message types $MT \subseteq ET$, which are root elements of *itype(I).* The channel model *CM* is a total function from a sequence of messages (of types *MT*) to a sequence of messages (of types *MT*). With $MT' \subseteq MT$ and a message sequence *s*, $\pi_{IT'}(s)$ denotes the projection of *s* to sequences of messages of types *MT'*. Let $\pi_{IT'}$ be canonically extended on the channel model. The channel model CM is then based on assignments of disjoint subsets *MT'* of *MT* to channel reliability guarantees[2] which enforce that $\pi_{IT'}(CM)$ satisfies certain properties. Reliability guarantees such as those from WS-RM standard [16] can be modeled:

[2] In the context of SAP applications, it is common to assign reliability guarantees per message type for the communication between two components.

- exactly once in order (EOIO) where $\pi_{IT'}(CM)$ is the identity function on interaction sequences and
- exactly once (EO) where $\pi_{IT'}(CM)$ is a permutation on an interaction sequence.

4 Translating MCM to Event-B

We chose Event-B for the purpose of obtaining a formally analyzable representation of MCM, which serves as basis for test derivation. In the following, we give a brief overview on Event-B, and sketch our translation from MCM.

Event-B is, as mentioned in Section 2.2, an evolution of the B-Method. It distinguishes between static and dynamic properties of a system; while static properties are specified in a context, the dynamic properties are specified in a so-called machine. A context contains definitions of carrier sets, constants as well as a number of axioms. A machine basically consists of a finite set of variables and events. The variables form the state of the machine and can be restricted by invariants. The events describe transitions from one state into another state. An event has the form `EVENT ≙` **ANY** `t` **WHERE** `G(t,x)` **THEN** `S(x,t)` **END**. It consists of a set of local variables `t`, a predicate `G`, called the guard and a substitution `S(x,t)`. The guard restricts possible values for `t` and `x`. If the guard of an event is false, the event cannot occur and is called disabled. The substitution `S` modifies the variables `x`. It can use the old values of `x` and the local variables `t`. E.g., an event that takes two natural number a, b and adds the product ab to the state variable x could be written as `EVENT ≙` **ANY** `a,b` **WHERE** `a∈ℕ ∧ b∈ℕ` **THEN** `x:=x+a*b` **END**. For events that do not require local variables, the abbreviated form `EVENT ≙` **WHEN** `G(x)` **THEN** `S(x)` **END** can be used. The primary way to structure a development in Event-B is through incremental refinement preserving the system's safety and termination properties.

Design Considerations of Translation. We are interested in a formal representation of both, the GCM for a global test generation and the two local LPMs with a connecting channel model. The latter is necessary to map the generated global test cases to local test cases that can be executed on the implemented components. Therefore the subsequently described translation generates two Event-B machines which use a common context: the Global Model describing the *GCM* and the Local Model, describing the composition (defined as in [8]) of the two *LPMs* and the *CM*. Both machines describe the exchange of messages – the first in terms of observing a message, and the latter in terms of sending and receiving messages.

As messages with the same type and content may occur more than once, to each message a unique natural number is assigned, which is incremented when a new message is sent. Further to each message a type is assigned while it is possible to specify the content of the message as functions on the message.

Because we aim at the use of a model checking technique the translation result is designed to be as deterministic as possible. We experimented with an assignment of types to messages which is non-deterministically initialized upfront; however this resulted in an indigestible state space for the model checker.

Translation Description. By defining a translation from the global and from the local MCM models into the two Event-B machines we obtain a precise semantics of MCM, which we present in the following. The translation is implemented and can thus be applied completely automatically.

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

```
i ≙
WHEN
guard1: status=s1 ∨ … ∨ status=sk
THEN
act1: status ≔ sm
END
```

This basic translation must be augmented with preconditions and actions associated with that interaction. Therefore we have to represent data types, constants, variables, terms and formulae used in *MCM* in terms of Event-B. This is done as follows. For each data type $t \in T$ we define a set in the Event-B context without explicit characterization of elements. These sets are named in Event-B according to their type name *name(t)*. For each complex data type $t=\{(f, t')\}$ we define a partial function `f`: $name(t) \mapsto name(t')$. `f` is initialized with `f≔∅`.

The constants and global variables are defined in a standard way. For each constant $c \in C_t$ an element is added to the set *name(t)*. For the interactions $I=\{i_1,\ldots i_n\}$ we additionally define a set `MESSAGES={`$name(itype(i_1,)),\ldots, name(itype(i_n))$`}`.

Example. Consider the interaction *Request* with $pre(Request) = msg.Header.ID \notin ID\_SET$ and $act(Request)\ (ID\_SET) = ID\_SET \cup \{msg.Header.ID\}$ of our running example. For it, we define the functions `Header: ℕ ⇸ MessageHeader` and `ID: MessageHeader⇸InstanceID` (*MessageHeader* and *InstanceID* here are the corresponding names from *name(T)*), and the local variables `t1` and `t2` in order to choose appropriate values to be assigned in the functions. Because $ID\_SET \in T_{Set(InstanceId)}$ we define an Event-B variable `ID_SET` of type `ℙ(InstanceID)`.

```
Request  ≙
ANY t1 t2
WHERE
grd1: status=Reserved ∨ status=Start
grd2: t1 ∈ MessageHeader
grd3: t2 ∈ InstanceID
grd4: t3 ∉ ID_SET
grd5: t1∈dom(ID)⇒ID(t1)=t2
THEN
act1: status ≔ Requested
act2: Header (msg)≔t1
act3: ID(t1)≔t2
act4: type(msg) ≔ Request
act5: ID_SET≔ID_SET ∪ {t3}
act6: msg ≔ msg + 1
END
```

The guard `grd5` describes a consistency property: if the function is already defined on an element, then the value must be the corresponding term.

For the target state $e_i \subseteq S$ we define a special event `terminate` with a guard `status=c`$_1$ `∨ ... ∨ status=c`$_1$ *(for all $c_i \in e_i$)* and an action `targetstate:=true`, where `targetstate` is a global variable. In each event from the translation of *GCM* we additionally add an action `targetstate:=false`. As a result, `targetstate` equals *true* iff the system state is a target state.

Local Model. In the local model we generate events representing sending and receiving of messages. Depending on the viewpoint either the send or the receive event can be defined to be a refinement of the corresponding interaction in GCM.

By definition of *LPMs,* the variables from *V* and the status variable are duplicated (one for each partner). The variable *msg* is translated as for the GCM in order to keep the unique message enumeration. It is only used by send events, where it is set in the same way as in the GCM. In receive events, local variables (parameters) are used in order to obtain some message from a channel.

A channel is defined as a global variable of type $\mathbb{P}(\mathbb{N})$ denoting the set of messages on the being exchanged. It is initialized with $\emptyset$. Typically, we have two partners P_1 and P_2 and two sequencing contexts (EO and EOIO). In that case we obtain four possible channels in the model (two in each direction).

Example. Below we show a translation of the interaction *Request* from the *LPMs* for the partners *buyer (B)* and *seller (S)* of the example. The duplicated variables can be distinguished by the corresponding prefixes. The channel from *buyer* to *selle*r having the sequencing EO is denoted by `channel_BS_EO`.

```
send_Request   ≙
ANY t1 t2
WHERE
grd1: B_status=Reserved ∨
      B_status=Startgrd2: t1 ∈
MessageHeader
grd3: t2 ∈ InstanceID
grd4: t3 ∉ B_ID_SET
grd5: t1∈dom(ID)⇒ID(t1)=t2
THEN
act1: B_status:=Requested
act2: Header(msg):=t1
act3: ID(t1):=t2
act4: type(msg):=Request
act5: B_ID_SET:=B_ID_SET ∪ {t3}
act6: channel_BS_EO:=channel_BS_EO∪{msg}
act7: msg := msg + 1
END
```

```
receive_Request   ≙
ANY m
WHERE
grd1: S_status=Reserved ∨
      S_status=Start
grd2: m ∈ channel_BS_EO
grd3: type(m) = Request
grd4: m ∈ dom(Header)
grd5: Header(m)∈dom(ID)grd6:
ID(Header(m))∉ S_ID_SET
THEN
act1: S_status := Requested
act2: S_ID_SET := S_ID_SET ∪
       {ID(Header(m))}
act3: channel_BS_EO :=
       channel_BS_EO \ {m}
END
```

The translation of a send event is very similar to the translation of the corresponding event in *GCM*. In receive events all function values are already set so the purpose is to find a suiting message *m* in the channel and "receive" it (delete from the channel). If a sequencing context is EOIO then we need an additional guard that checks, that the message m has a smallest number in the channel.

For inhibitor conditions *inhib(i)*=C *(with i∈I*) we add a guard `status∉C` to the event `send_i`. In our example, we add the guard `grd6: B_status∉{Reserved}` to `send_Request`. It remains future work to optimize the translation by simplifying this and `grd1` to `B_status=Initial`.

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

5 Test Generation

In this section we describe how we utilize ProB to obtain an optimized test suite (regarding the objectives explained in Section 2.3) from the translated MCM models.

ProB [15] is a validation toolset originally written for the B method. Its automated animation facilities allow users to animate and model-check their specifications which are valuable capabilities in the development of formal specifications. While consistency can be proven within tools such as Rodin or AtelierB, they are not capable of validating whether the model matches the specification that the modeler intended. Using the ProB animator, confidence in the models can be gained while using the model checker allows (at least for a part of the model's state space) to verify that a certain property holds. ProB has been adapted to support a number of formalisms such as Z, CSP, and CSP||B [9]. Recently a ProB plug-in for the Rodin Platform has been developed, that can be used to animate and model check an Event-B specification within Rodin and to export Event-B models for using it in the ProB application. In the MCM editor the animation of the generated models is used but a detailed description in this paper is out of scope.

The test generation algorithm we developed for the MBIT approach based on MCM is separated into three steps. In the following we describe each step, give details about the implementation and show the computed results when applying it to the example from Section 2.

Step 1: Generation of the Initial Global Test Suite. As explained, our aim is to cover each transition of the global communication model, i.e. each interaction of the GCM. As each interaction is translated into a separate Event-B event, we have to ensure that every event is covered by at least one concrete transition in the state space of the global Event-B model, from which a valid end state can be reached. Note that the same event is typically covered by many different transitions, as its parameters can be valued in many different ways. In our particular example, the full state space is actually infinite, due to the use of integers as message identifiers. In order to reduce the state space, we have to configure ProB to compute only a few possible ways to enable any event.[3]

To satisfy the first and second objective given in Section 2.3, we have extended ProB to detect when full transition coverage is obtained[4]. This is gained by exploring the state space of the model breadth first, stopping when full coverage is achieved. Note that we also need to secure that for every operation we can reach a valid end state.

[3] This approach has proven to be sufficient so far, but in future, we will consider using ProB's symmetry reduction instead.

[4] Note that this is a property that cannot be expressed as an LTL formula, as it is not a property of individual paths but of the entire state space explored so far.

This has been ensured by refining the Event-B translation described in Section 4, by adding a `history` variable, storing the set of executed events, and adding a corresponding end-event for every original event `e` which can be triggered if we are in a valid end state and if `e∈history`. Afterwards all traces that end in a target state are extracted from the explored state space to form the initial test suite. From the example in this paper, we obtain the following initial test suite:

```
[Request, Confirm, Order], [Request, Confirm, Cancel],
[Request, Confirm, Cancel, Request, Confirm, Order],
[Request, Confirm, Cancel, Request, Confirm, Cancel],
[Request, Confirm, Request, Confirm, Order],
[Request, Confirm, Request, Confirm, Order, Cancel(depr.)],
[Request, Confirm, Request, Confirm, Cancel],
[Request, Confirm, Request, Confirm, Cancel(depr.), Order],
[Request, Confirm, Request, Confirm, Cancel(depr.), Cancel],
[Request, Confirm, Request, Cancel(depr.), Confirm, Order],
[Request, Confirm, Request, Cancel(depr.), Confirm, Cancel]
```

The computation takes 0.32 seconds on a 2.33 GHz Core2 Duo laptop and should scale up to much larger examples.

Step 2: Mapping of Global to Local Paths. In order to obtain executable test cases the global sequence of message observations for each path has to be mapped to the corresponding send and receive events of partners. As the GCM uses receive semantics, the global observe sequences can be directly translated to sequences of receive events. In the case of the path

```
[Request, Confirm, Request, Confirm, Cancel(depr.), Cancel]
```

the resulting sequence is (? reads "receives"):

```
[ Seller?Request, Buyer?Confirm, Seller?Request, Buyer?Confirm,
  Seller?Cancel(depr.), Seller?Cancel]
```

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

```
[ Buyer!Request, Seller?Request, Seller!Confirm, Buyer?Confirm,
  Buyer!Cancel, Buyer!Request, Seller?Request, Seller!Confirm,
  Buyer?Confirm, Seller?Cancel(depr.), Buyer!Cancel, Seller?Cancel]
```

The message racing in the illustrated local path is underlined. While the *Cancel* message is sent by the buyer before the *Request* message, the seller receives the *Request* message first.

Similar to Step 1, it is again infeasible to exhaustively explore the full state space (as the state space of the local model is actually even considerably bigger) to find a suitable mapping from global to local traces. One could encode the problem as an LTL formula, but this formula will be very big with ensuing consequences for the complexity of model checking. The solution we have come up with, is to encode the desired LCM scenarios into a CSP [12] process. This process is synchronized with the Event-B model, using the technology of [9], suitably guiding the model checker. The CSP Process is divided into two components.

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

```
RECEIVER =  Seller?Request -> Buyer?Confirm -> Seller?Request ->
            Buyer?Confirm -> Seller?Cancel(depr.) ->
            Seller?Cancel -> goal -> STOP
```

The second process encodes the sender events. We know how many send events of each type must occur, but the order of these is unknown.

```
SENDER(n1,n2,n3,n4) =
          n1>0 & Buyer!Request -> SENDER(n1-1,n2,n3,n4) []
          n2>0 & Seller!Confirm -> SENDER(n1,n2-1,n3,n4) []
          n3>0 & Buyer!Cancel -> SENDER(n1,n2,n3-1,n4) []
          n4>0 & Buyer!Order -> SENDER(n1,n2,n3,n4-1)
```

The sender process is now simply interleaved with the receiver process.[5]

```
MAIN = SENDER(2,2,2,0) ||| RECEIVER
```

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

Step 3: Test Suite Reduction. The resulting test suite incorporating the local traces is now ready to be optimized according to the third and fourth objective from Section 2.3. The optimization of the test suite and the test suite reduction has been implemented in Java. In the first prototypical version we use a brute force algorithm that computes every possible combination of test cases and selects the optimal one according to the given objectives. The computed optimal test suite incorporates the local equivalents of the following global paths:

```
[Request, Confirm, Request, Cancel(depr.), Confirm, Order],
[Request, Confirm, Request, Confirm, Cancel(depr.), Cancel],
[Request, Confirm, Request, Confirm, Order, Cancel(depr.)]
```

For the given example the test suite is produced in less than a millisecond, implying that it is applicable in practice. However as the algorithms computational complexity is exponential in the number of test cases of the extended suite, we are planning to apply the following more sophisticated approach that reduces the number of computations: First it is analyzed which of the global interactions can only be covered by paths incorporating message racing. In our example these are the three interactions called *Cancel (deprecated)*. For these a minimal set of covering paths is determined

[5] Note that we could have additionally encoded that every receive event must be preceded by a corresponding send event in the CSP process, but this will be automatically checked by the Event-B model anyway.

using a greedy algorithm. If more than one possibility exists, the one that has the highest overall interaction coverage is chosen. The resulting test suite is filled with the minimum set of paths (not incorporating message racing) that covers the remaining interactions.

6 Related Work

The academic test generators TorX [21] and TGV [14] utilize model checkers to generate test cases from labeled transition systems (e.g. EFSM). However, problems with scalability have been identified as the major weakness of their approaches in case studies of the AGEDIS project [10]. Our work is based on a different abstraction level and formalism, which we hope will overcome those issues. For example, symmetry can be detected and exploited very easily in B. Also, the use of a higher-level formalism can significantly reduce the blowup of the associated state space [16].

There are various MBT approaches that generate test cases from classical B models, upon which we build. One is the commercial LEIROS tool [13], based on the former BZ-testing tool [7], which is rooted in constraint logic programming to find boundary values. The other approach [18,19] uses ProB [17] – itself also rooted in constraint logic programming – and is based on adding tautologies (e.g., $x=\emptyset$ or $x\neq\emptyset$) to guards and the invariant and then uses the disjunctive normal form (DNF) to partition the executed operations according to the particular disjuncts covered. Traces are generated which try to cover every operation in every reachable partition. An expensive part of [18,19] is the generation of the DNF, which is effectively used to compute boundary cases. In our approach we overcome the need for the DNF and the need to find boundary cases by using Event-B, where events are more fine-grained than in classical B (e.g., due to the absence of complicated substitutions such as CASE or IF-THEN-ELSE). As such, events are already "partitioned" into individual cases by construction. Also, the above approaches do not address the problem of optimizing the test suite or test generation for decomposed systems, which are both a major consideration in our article.

7 Conclusion

In this paper we presented an approach to generate test suites for service choreographies, modeled in MCM, by using model checking. We described how choreography models are translated to Event-B models, which are a suitable input format for ProB, the model checker we used for the test generation. We have extended ProB to detect transition coverage, and have made use of the possibility to guide an Event-B model by a CSP process in order to translate high-level traces into low-level ones. The flexibility of ProB was crucial in addressing the various aspects of choreography models. We further explained the overall integration testing approach including the test goals and introduced the according test generation algorithm as well as its implementation. The test suite for the running example of this paper, has been computed automatically by our implementation. As MCM explicitly considers asynchronous communication, the generation of test suites incorporating message racing is a direct contribution to the research community, as is the utilization of a higher level of abstraction (the global model) to compute an integration test suite, thus avoiding the well known problem of state explosion.

As explained our test generation approach was designed such that the resulting test suite causes a minimal effort during later test concretization and execution. However we see some potential optimizations that could be applied to the test generation steps without sacrificing our goal of minimal test effort. We will also evaluate the fault uncovering capabilities of transition coverage compared to other applicable criteria and therefore will continue to work on suitable test generation algorithms. In order to assess our approach we are currently conducting additional experiments using typical case studies at SAP.

Acknowledgments. This work was partially supported by the EC-funded projects Modelplex[6] and Deploy[7] (grants no. 034081 and 214158).

References

1. Business Process Modeling Notation (BPMN) Specification 2.0, Submitted Draft Proposal V0.9, `http://www.omg.org/cgi-bin/doc?bmi/08-11-01`
2. Abrial, J.-R.: The B–Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: A roadmap for the Rodin toolset. In: Abstract State Machines, B and Z (2008)
4. Abrial, J.-R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundam. Inform. 77(1-2), 1–28 (2007)
5. Aho, A.V., Dahbura, A.T., Lee, D., Uyar, M.Ü.: An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. IEEE Trans. Commun. 39, 1604–1615 (1991)
6. Ali, S., Briand, L., Jaffar-Ur Rehman, M., Asghar, H., Iqbal, M.Z., Nadeem, A.: A State-Based Approach to Integration Testing Based on UML Models. Information & Software Technology 49(11–12), 1087–1106 (2007)
7. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Utting, M., Vacelet, N.: BZ-Testing-Tools: A Tool-Set for Test Generation from Z and Busing Constraint Logic Programming. In: Proc. of FATES 2002, pp. 105–120 (2002)
8. Butler, M.: Decomposition Structures for Event-B. In: Integrated Formal Methods (2009)
9. Butler, M., Leuschel, M.: Combining CSP and B for specification and property verification. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
10. Craggs, I., Sardis, M., Heuillard, T.: AGEDIS Case Studies: Model-Based Testing in Industry. In: Proc. of ECMDA 2003, pp. 129–132 (2003)
11. Gallagher, L., Offutt, A.J., Cincotta, A.: Integration Testing of Object-oriented Components using Finite State Machines. J. Software Testing, Verification, and Reliability (2006)
12. Hoare, C.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
13. Jaffuel, E., Legeard, B.: LEIRIOS Test Generator: Automated Test Generation from B Models. B 2007, 277–280 (2007)

[6] http://www.modelplex-ist.org

[7] http://www.deploy-project.eu

14. Jard, C., Jeron, T.: TGV: theory, principles and algorithms. J. Software Tools for Technology Transfer 7(4), 297–315 (2005)
15. Kavantzas, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web services choreography description language. W3C candidate recomm (2005), `http://www.w3.org/TR/ws-cdl-10`
16. Leuschel, M.: The High Road to Formal Validation. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 4–23. Springer, Heidelberg (2008)
17. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
18. Satpathy, M., Leuschel, M., Butler, M.: ProTest: An Automatic Test Environment for B Specifications. In: Proc. ENTCS 111, pp. 113–136 (2005)
19. Satpathy, M., Butler, M., Leuschel, M., Ramesh, S.: Automatic Testing from Formal Specifications. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 95–113. Springer, Heidelberg (2007)
20. Snook, F., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (2006)
21. Tretmans, J., Brinksma, E.: TorX: Automated Model Based Testing. In: EMSOFT (2002)
22. Utting, U., Legeard, B.: Practical Model-Based Testing – A Tools Approach. Morgan Kaufmann Publ., San Francisco (2007)
23. Web Services Reliable Messaging (WS-ReliableMessaging), Version 1.1. OASIS Consortiom, `http://docs.oasis-open.org/ws-rx/wsrm/v1.1/wsrm.pdf`
24. Weyuker, E.: Testing component-based software – a cautionary tale. IEEE Software 15(5), 54–59 (1998)
25. Wieczorek, S., Roth, A., Stefanescu, A., Charfi, A.: Precise Steps for Choreography Modeling for SOA Validation and Verification. In: SOSE, pp. 148–153. IEEE, Los Alamitos (2008)
26. Wieczorek, S., Roth, A., Stefanescu, A., Kozyura, V., Charfi, A., Kraft, F.M., Schieferdecker, I.: Viewpoints for Modeling Choreographies in Service-Oriented Architectures. In: Proc. of WICSA 2009. IEEE Computer Society, Los Alamitos (to appear, 2009)
27. Wieczorek, S., Stefanescu, A., Großmann, J.: Enabling Model-Based Testing for SOA Integration Testing. In: MOTIP, pp. 77–82. Fraunhofer IRB Verlag (2008)
28. Wieczorek, S., Stefanescu, A., Schieferdecker, I.: Model-based Integration Testing of Enterprise Services. In: Proc. TAICPART. IEEE Computer Society, Los Alamitos (2009)
29. Wieczorek, S., Stefanescu, A., Schieferdecker, I.: Test Data Provision for ERP Systems. In: Proc. of ICST, pp. 396–403. IEEE Computer Society, Los Alamitos (2008)