Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)

Checking Consistency Between Message Choreographies And Their Implementation Models

Vitaly Kozyura, Andreas Roth, Sebastian Wieczorek and Wei Wei

15 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Checking Consistency Between Message Choreographies And Their Implementation Models

Vitaly Kozyura, Andreas Roth, Sebastian Wieczorek and Wei Wei

SAP Research CEC Darmstadt, SAP AG, Bleichstr. 8, 64283 Darmstadt, Germany
[v.kozyura|andreas.roth|sebastian.wieczorek|wei01.wei]@sap.com

Abstract: Applying the concepts of Service-Oriented Architectures (SOA) has already become a mainstream in industry. The development of business applications according to these principles implies a layered design and implementation. This paper describes an industrial approach to the verification of a consistency relation between such layers. In our case service choreographies defined by Message Choreography Models (MCM) and their corresponding implementation models represented as Business Objects are examined. By translating both into Event-B specifications we are able to prove the consistency relation between them. A number of case studies with realistic industrial software models were carried out which showed the solidness of our verification technique. Apart from giving details about our concrete realization, this paper also discusses general challenges that have to be faced when developing a verification approach applicable for the real-world systems.

Keywords: SOA, Modeling, Choreography, Industrial

1 Introduction

Service-oriented architectures (SOA) provide frameworks and methods to compose single services in order to realize complex business scenarios. At the lower end, a single service is described as a set of operations and message types. Its functions usually rely on a simple request-response pattern, which can be specified using standards like XML, SOAP, and WSDL. At the service integration level, more complicated specifications are needed to capture not only the formats of messages, but also the orders and dependencies among exchanged messages, in particular, both control-flow and data-flow dependencies. Thus, the challenge of the SOA-based development lies in the integration of different services according to the defined business processes. Choreography languages were introduced to describe the interaction protocols between service components communicating over message channels from the perspective of a global observer. Further, choreography languages also allow the specification of the communication behavior of local components.

SOA adoption is gaining pace toward becoming a mainstream [3]. Being a leader in the area of business software, SAP delivers SOA via its service-enabled software (e.g., SAP Business ByDesign[1], SAP Business Suite[2]) and its open technology platform SAP

[1] http://www.sap.com/sme/solutions/businessmanagement/businessbydesign
[2] http://www.sap.com/solutions/business-suite

NetWeaver[3]. In a model-driven fashion, the enterprise SOA developed at SAP utilizes many different types of models for business objects, deployment units, service components, service interfaces, integration scenarios, business process variants, and service choreographies [4].

The development of business applications according to the 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 [6]. The work presented in this paper 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 [6], we check consistencies through a translation into Event-B [1], a formal specification language supported by the Rodin platform [9]. This work has been carried out in the context of SAP software developments, and we will report on the experience we gained from it.

Section 2 gives an overview of layered development and explains the motivation of ensuring consistency between different layers from an industrial perspective. Section 3 briefly discusses the choreography modeling language MCM and its verification. Section 4 introduces the basic concepts of implementation models used at SAP. Section 5 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 6. Section 7 concludes the paper and discusses the lessons learnt.

2 Industrial Context

Our modeling approach is based on a three-layer architecture. In this paper we focus on the consistency relation between the last two layers. (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

[3] http://www.sap.com/platform/netweaver

models [10]. Considering our application domain, we define consistency in terms of trace inclusion. This paper presents 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 [2]: 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 [6] we described a mixed approach that takes best of both worlds.

The **consistency between LPMs and IMs** is the main concern of this paper. We use the Event-B specification language [1] and the Rodin platform [9] 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 [7]).

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 Global Choreography and Local Partner Models

A message choreography model (MCM) [11, 13, 12] 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 [11] 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 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 1, the arrows labeled with an envelope depict the interactions *Request*, *Confirm*, *Cancel*, *Order*, and *Cancel(deprecated)*[4] which are ordered with the help of the states *Start*, *Request*, *Re-*

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

served, 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.

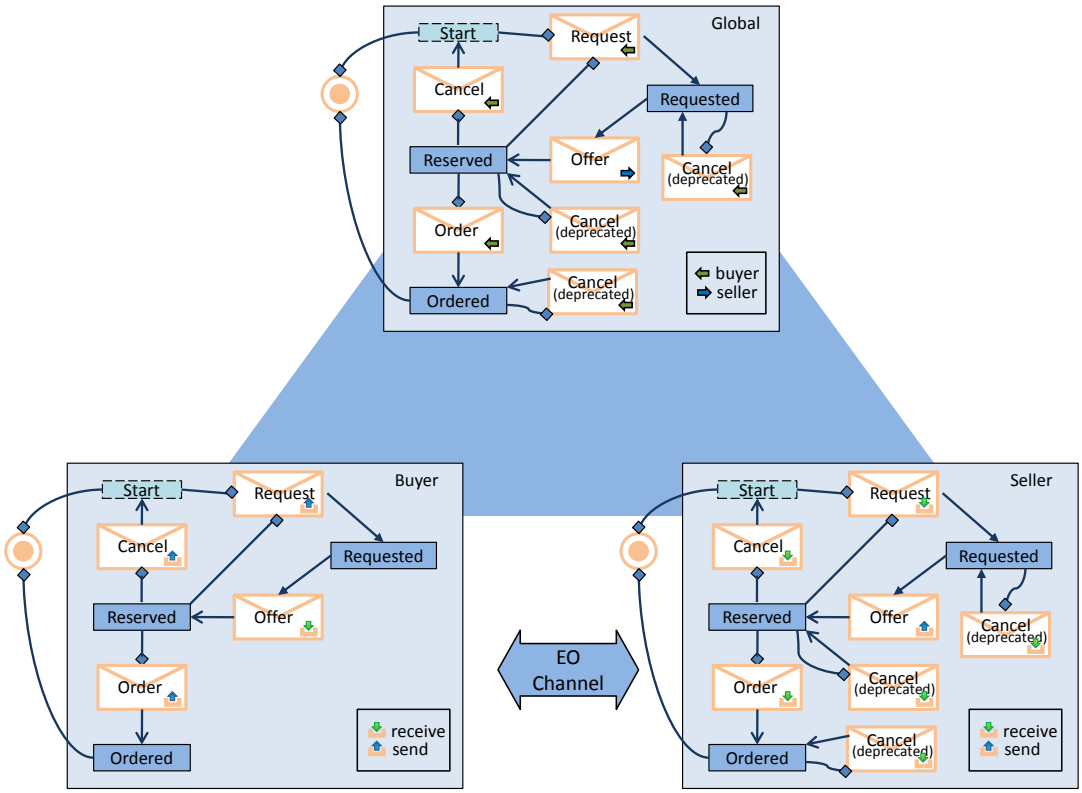

Figure 1: GCM (top) of the choreography and LPMs of the buyer (left) and the seller (right)

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. The LPM of the seller is depicted in the lower right part of Figure 1 with the exact structure as the GCM.

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 [11]. 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 $\mathscr{P}(T)$, where T is a set of possible message types, denoting the set of messages being exchanged. It is initialized with $\varnothing$. 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 [13] 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 [5] 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.

4 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. These can be considered as refinements of their communication interfaces (i.e., local partner models), as illustrated in Figure 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 [8].

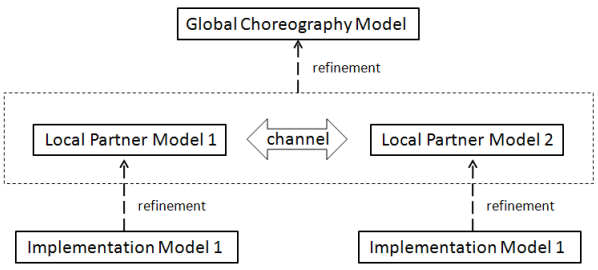

Figure 2: Refinement relations

We define implementation models as model “templates” from which many model in-

stances 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.

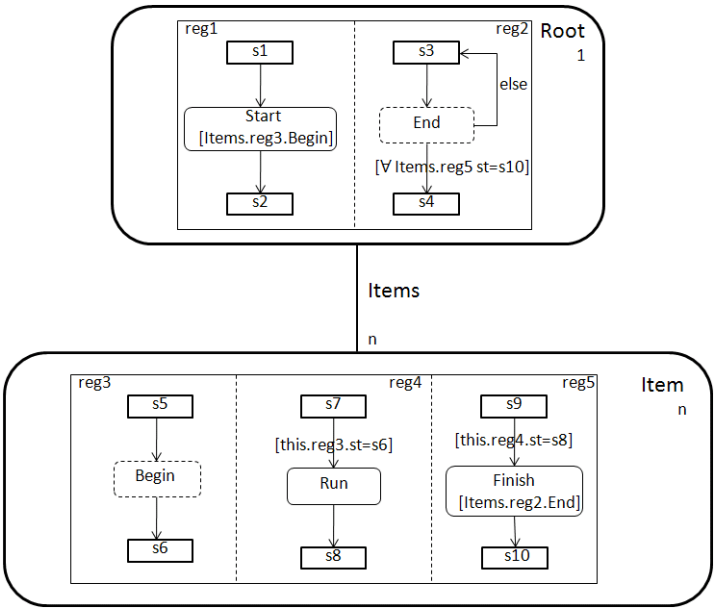

Figure 3: Example of an implementation model

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 $n.parent$ 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, $n.reg$ represents the region reg in n, whose current state is represented by $n.reg.st$. Finally, $n.reg.t$ refers to the transition t in reg.

A transition has a firing guard, a set of (pre_i, s_i) pairs that selects the next state s_i according to a certain pre-condition pre_i, and a sequence of transitions that must be invoked in order. In our example, `Start` is an active transition, whose guard is that the current state must be `s1`. When it is fired, the next state is `s2`, and it will enforce the transitions `Begin` in all `Item` nodes to be taken. The transition `End` is a passive transition, and can be fired only if it is invoked by the `Finish` in one of the `Item` nodes. It ends up in two possible next states: (1) If the regions `reg5` in all `Item` nodes are in state `s10`, then the next state is `s4`; (2) Otherwise, the next state is `s3`. The firing of `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.

5 Translation of Implementation Models to Event-B

In this section we describe a translation of IMs (node structures and state machines) into Event-B (For more information on EventB see [1]). 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 5.3 we show how the translation can be optimized in order to simplify Event-B representations and proofs and in Section 6.3 we compare results obtained by optimized and non-optimized translations.

5.1 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 is a simplified version of the example in Figure 4. We consider the more complex version of the example in Figure 4 only in Section 5.1 in order to demonstrate some aspects of the translation of a tree-structure. In Section 5.2 and further we will continue with the simplified version in Figure 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

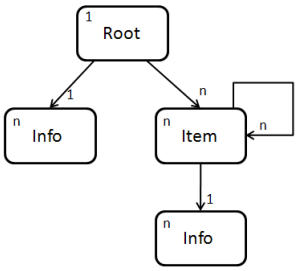

Figure 4: Example of a node structure

case for node types `Item` and `Info`. For these carrier sets, we need additional constraints stating that they are non-empty.

carrier sets: $Item, Info$

axioms:
$ax1 : Item \neq \varnothing$
$ax2 : Info \neq \varnothing$

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 4 are translated in Event-B.

variables: $root\_info, root\_item, item\_item, item\_info$

invariants:
$inv1 : root\_info \in Info$
$inv2 : root\_item \in \mathscr{P}(Item)$
$inv3 : item\_item \in root\_item \nrightarrow Item \setminus root\_item$
$inv4 : item\_info \in Info \rightarrowtail Item$

In our example, an `Item` node can be associated with a child `Item` node, which we refer to as a *sub-item* (Sub-items do not have further sub-items) [5].

[5] 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 [9].

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 $root\_item' \cap dom(item\_item') = \varnothing$ says that the set of items related to the root is disjoint from the set of items related to other items.

```
init
  begin
    act1 : item_info, root_info :|
    item_info' ∈ Info ⤖ Item ∧ root_info' ∈ Info ∧
    root_info' ∉ dom(item_info') ∧ root_info' ∪ dom(item_info') = Info

    act2 : root_item, item_item :|
    root_item' ∈ 𝒫(Item) ∧ item_item' ∈ Item ⇸ Item ∧
    root_item' ∩ dom(item_item') = ∅ ∧
    root_item' ∪ dom(item_item') = Tasks
  end
```

5.2 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 `reg1` in Figure 3 is translated to a variable $reg1.st \in reg1.States$; and the current state of `reg3` is a function $reg3.st \in Items \to reg3.States$.

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 `true`, a set of pairs $(pre_{i1}, s_{i_1}), \ldots, (pre_{ik_i}, s_{ik_i})$, and does not further enforce other transitions. The transition t is translated as follows:

```
EventT
  when
    grd1 : g
  then
    act1 : st, st_1, ..., st_m :|
    ((pre_1 ∧ st' = s_1) ∨ ... ∨ (pre_n ∧ st' = s_n)) ∧
    ...
    ((pre_{i_1} ∧ st'_i = s_{i_1}) ∨ ... ∨ (pre_{ik_i} ∧ st'_i = s_{ik_i})) ∧
    ...
  end
```

In the above code we use st and $st_1, \ldots, st_m$ to denote the current states of those regions that contain transitions t and $t_1, \ldots, t_n$, 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 are translated. The relation `Items` is represented as a subset of all `Item` nodes since there is only one `Root` node.

Start
when
$grd1 : reg1.st = s1$
then
$act1 : reg1.st := s2$
$act2 : reg3.st := Items \times \{s6\}$
end

Run
any *node*
where
$grd1 : node \in Items$
$grd2 : reg4.st(node) = s7$
$grd3 : reg3.st(node) = s6$
then
$act1 : reg4.st(node) := s8$
end

Finish
any *node*
where
$grd1 : node \in Items$
$grd2 : reg5.st(node) = s9$
$grd3 : reg4.st(node) = s8$
then
$act1 : reg5.st, reg2.st :|$
$(reg5.st'(node) = s10) \wedge$
$(\forall n \in Items \wedge n \neq node \Rightarrow reg5.st'(n) = reg5.st(n)) \wedge$
$(\quad ((\forall n \in Items \Rightarrow reg5.st'(n) = s10) \wedge reg2.st' = s4) \vee$
$((\exists n \in Items \Rightarrow reg5.st'(n) \neq s10) \wedge reg2.st' = s3) \quad)$
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.

5.3 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[6], we may

[6] We thank Michael Buttler for his suggestions and discussions for this optimization solution.

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 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.

Finish
any $node, st5, st2$
where
$grd1 : node \in Items$
$grd2 : reg5.st(node) = s9$
$grd3 : reg4.st(node) = s8$
$grd4 : st5 \in (Item \rightarrow reg5.States)$
$grd5 : st2 \in reg2.States$
$grd6 : st5(node) = s10$
$grd7 : \forall n \in Items \ \wedge n \neq node \Rightarrow st5(n) = reg5.st(n)$
$grd8 : (\forall n \in Items \Rightarrow st5(n) = s10) \Rightarrow st2 = s4$
$grd9 : (\exists n \in Items \Rightarrow st5(n) \neq s10) \Rightarrow st2 = s3$
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 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 `grd7` can be rewritten as below, where $\lhd$ is domain subtraction:

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

In a similar manner, the guards `grd8` and `grd9` can be written as

$$grd8 : ran(st5) = \{s10\} \Rightarrow st2 = s4$$
$$grd9 : st5 \rhd \{s10\} \neq \varnothing \Rightarrow st2 = s3$$

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.

6 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.

6.1 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 [5], 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 [5] 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 [7] or the Atelier B provers [9]. 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 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.

$$(Root.reg1.st = st1) \Rightarrow (\forall item \in Items \Rightarrow item.reg3.st = s5).$$

6.2 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 expressing relations between states in the root node and in item nodes:

$$(Root.reg1.st = s1) \Rightarrow (\forall item \in Items \Rightarrow item.reg3.st = s5)$$

$$(\forall item \in Items \Rightarrow item.reg5.st = s10) \Rightarrow (Root.reg2.st = s4)$$

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.

6.3 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 [11]. 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. 5.3). 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. In the future we will design some automated procedures to assist in discovering auxiliary lemmas.

7 Conclusion

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 paper 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.

Bibliography

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010. To appear. See also http://www.event-b.org.

[2] Gero Decker and Mathias Weske. Local enforceability in interaction Petri Nets. In *Proceedings of the 5th International Conference on Business Process Management (BPM'07)*, volume 4714 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2007.

[3] Randy Heffner. Across all vertical industry groups, the majority of SOA users are expanding its use. Research report, Forrester Research, May 2009.

[4] Stefan Kätker and Sabine Patig. Model-driven development of service-oriented business application systems. In *Business Services: Konzepte, Technologien, Anwendungen*, volume Band 1, pages 171–180. Österreichische Computer Gesellschaft, 2009.

[5] Vitaly Kozyura and Andreas Roth. Generation of gluing invariants for checking local enforceability of message choreographies. In Michael Jastram, Linas Laibinis, Felix Lösch, and Manuel Mazzara, editors, *Proceedings of Deploy Technical Workshop 2009*. Newcastle University, Technical Report, 2009.

[6] Vitaly Kozyura, Andreas Roth, and Wei Wei. Local enforceability and inconsumable messages in choreography models. In *Proceedings of 4th South-East European Workshop on Formal Methods (SEEFM)*. IEEE Computer Society, 2009.

[7] Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

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

[9] RODIN. http://www.event-b.org/. accessed 2010-04-08.

[10] R.J. van Glabbeek. The linear time-branching time spectrum. In *in Proceedings of Theories of Concurrency: Unification and Extension (CONCUR'90)*. Springer, 1990.

[11] Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Proceedings*

of the 21st IFIP Int. Conference on Testing of Communicating Systems (TESTCOM'09), LNCS. Springer, 2009.

[12] Sebastian Wieczorek, Andreas Roth, Alin Stefanescu, and Anis Charfi. Precise steps for choreography modeling for SOA validation and verification. In *Proceedings of the IEEE 4th International Symposium on Service-Oriented Software Engineering (SOSE'08)*, pages 148–153. IEEE Computer Society, 2008.

[13] Sebastian Wieczorek, Andreas Roth, Alin Stefanescu, Vitaly Kozyura, Anis Charfi, Frank Michael Kraft, and Ina Schieferdecker. Viewpoints for modeling choreographies in service-oriented architectures. In *Proceedings of the 8th IEEE/IFIP Conference on Software Architecture (WICSA'09)*. IEEE Computer Society, 2009.