Precise Steps for Choreography Modeling for SOA Validation and Verification

Sebastian Wieczorek, Andreas Roth, Alin Ştefănescu, Anis Charfi
SAP Research, CEC Darmstadt, Germany
{firstname.lastname}@sap.com

Abstract

Service-oriented architecture (SOA) enables organizations to transform their existing IT infrastructure into a more flexible business process platform.. In this architecture, decoupled components that provide standard services can be composed to form individually configured and highly flexible applications. When building such applications it is important to have a formal specification of the interaction protocols between the composed services not only because such a specification provides an accurate and unambiguous description of the interactions and their ordering but also to enable automated verification and validation. In this paper, we present a case study from the SAP context showing the interactions between two SAP service components and use that case study to derive a set of modeling requirements. This motivates a discussion about applicable techniques for service choreography modeling and whether existing choreography languages cover the identified needs.

1. Introduction

Enterprise Resource Planning (ERP) software [6] supports business processes for whole companies, with SAP being the world's leading provider of ERP software. ERP software integrates many organizational parts and functions into one logical software system, posing unique challenges to software development and testing [12],[2]. Such software systems are typically very large and very complex.

Service-oriented architecture (SOA) is regarded as the next evolutionary step to cope with the software complexity of ERP systems where monolithic approaches are not applicable anymore. SAP makes it easy to adopt SOA by delivering SOA-enabled software, SOA methodology guidelines, and professional services [13],[9]. In SAP's approach, independent business components exhibit enterprise services that can be composed individually to form customized business processes. Complex applications are built by combining services into composite applications. This integration takes place on a higher level of abstraction than system development.

In the area of SOA, choreography models describe the communication between a set of loosely coupled components. Choreography in that sense defines the allowed ordering of message exchanges between the components. The communication is observed from a global perspective and the local behavior of the involved components is usually not considered. According to the W3C Web Service Glossary [5] "a *choreography* defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state". A characteristic of choreography is that there exists no centralized control.

In this paper, we present a motivating message choreography example showing the interactions between two process components as they could occur in a real SOA based SAP system. Based on that example we derive a set of properties that should be supported by an appropriate formal choreography modeling language. We also discuss several issues related to these requirements and shortly survey which of the recently designed choreography modeling languages are suitable to fulfill these properties.

Our investigations are oriented towards SAP's approach to SOA. However, the results of this paper can be easily generalized to other settings where inter-component communication takes place. In the particular case of SAP, the choreography models are explicitly intended to be used for tools at SAP that can derive integration tests and also by tools that ensure consistency of existing component models by verification. This fits well into the model-driven development approach practiced at SAP.

The remainder of this paper is structured as follows. Section 2 presents a short motivating message choreography example. Section 3 defines the concrete

purposes of the choreography modeling. Section 4 presents a set of requirements that a formal choreography modeling language should support. Section 5 discusses related work on choreography languages, while Section 6 concludes the paper.

2. Choreography Example

As a running example we describe the message choreography between two business components: Purchase Order Processing and Sales Order Processing as part of a Sell-From-Stock scenario. In our case, there are two interacting parties, a buyer and a seller, which exchange messages in order to buy, respectively sell a product or a service. The buyer makes use of the purchase order component to control the purchasing process, while the seller uses the sales order component in his system to manage the received sales orders.

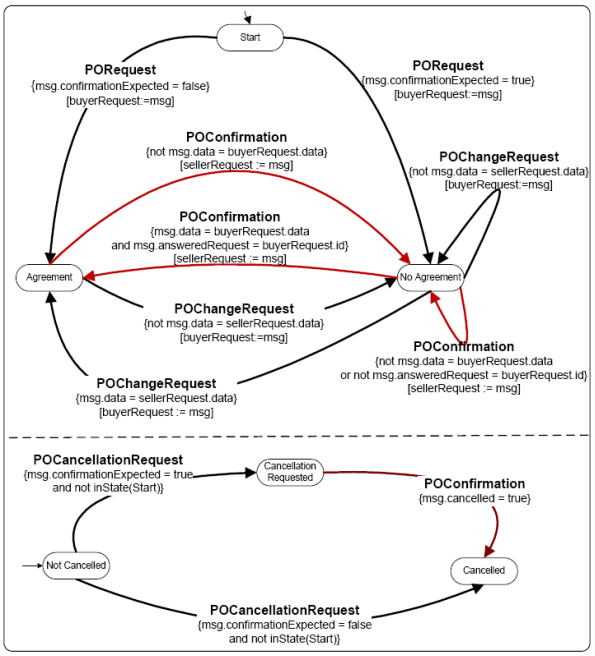

Figure 1. An extended concurrent state machine modeling the choreography between the Purchase Order (PO) and Sales Order (SO) components

Figure 1 presents a possible message choreography between the parties modeled as a UML-like extended concurrent finite state machine. The choreography is not very complex, but it is rich enough to exemplify the different aspects that we want to discuss later. Note that the graphical representation in Figure 1 is not a notation that we propose, but only serves to illustrate our investigations.

The state machine describes a global choreography protocol with the main observation that the messages exchanged between the parties are represented on the transitions. The messages are *PORequest* (Purchase Order Request from the buyer to seller), *POChangeRequest* (buyer notifies the seller on a change), *POCancellationRequest* (buyer cancel his order), and *POConfirmation* (seller confirms the request, changes or cancellations). The possible sequences of the exchanged messages are given by the possible executions of the state machine (taking into account also additional guards and side-effects as seen in the picture). The detailed description of the choreography is not provided here due to space limitations and the rest of the paper does not depend on these details.

3. Exploiting Choreography Models for Software Quality Assurance

After introducing the motivating example, in this section we are describing our specific objectives concerning choreography models in the context of the development of ERP systems based on SOA: comprehensibility, suitability for test case generation, and verifiability.

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

Suitability for test case generation. Second, we would like to derive integration tests from choreography models using model-based testing techniques. Model-based testing (MBT) [10] is a form of black-box testing that uses structural and behavioral models, described for instance in UML diagrams, to automatically generate test cases, thus automating the test design process. MBT test case generators aim to cover model related features, e.g. all states in a UML state machine, or data features, e.g. all boundary values. Therefore they are highly suitable for integration testing of SOA components [11].

Choreography models should thus cater for rich expressiveness so that meaningful integration tests can be derived and it should be possible to annotate them with information needed by the test case generator like local actions triggering certain messages. Only with such models advanced test coverage criteria like state or transition coverage for the communication protocol between the parties can be targeted.

Verifiability. Finally, we aim at discovering inconsistencies between choreography models and models describing the participating local components by static verification techniques. Note that the local models to a large extent already exist in the model-based development environment of SAP and are very

specific in their nature, thus approaches similar to the ones reported in [4] or [15] do not seem to be directly suitable. Formal techniques can be used for checking various properties of the models themselves and for checking the choreography models against the local components using a theorem prover or a model checker. Refinement based approaches [7] appear appropriate to address the issues of consistency between the choreography and the local models.

4. Choreography Model Properties and Requirements

As stated before our aim is to specify choreographies for SOA component interaction in order to obtain a formal and hence unambiguous description that can also be used for verification and validation. Having the three objectives from the last section in mind, we investigate now properties of choreography models needed for these purposes.

4.1 Detailed message description

As seen in Section 2, each component interaction is associated with a message type like *POCancellationRequest* or *POConfirmation*. Whenever a message of a certain type is sent, the global choreography moves into a new state. Message types alone are however insufficient to characterize such a transition. If we look at the second region at the bottom of Figure 1, we see that both transitions from state *Not Cancelled* have the message type *POCancellationRequest*, but their target states are different: the first one moves in the state *Cancellation Requested* in which a confirmation is expected, while the second one moves to the state *Cancelled*. If we had not considered the detailed information that *confirmationExpected=true*, we would have obtained a nondeterministic state machine, which would not be precise enough for applying model-based testing or verification. Therefore we require a message constraint language for the guards that can define under which conditions a transition may be triggered. The message constraint language should be capable of specifying that a certain field of a message has a certain value and references to earlier exchanged messages. Moreover it must at least be as expressive as a first order logic, i.e. Boolean operations and quantifiers as like *for all* and *there exists*.

4.2 Infinite state space

As mentioned before data structures and rich logical operators are necessary in order to model the choreography sufficiently precise. Unbounded data structures like integer values or lists and sets that may be used in the choreography modeling would generate an infinite state space.

The following situation also shows that an infinite state space must be considered. Suppose we want to model that a number of requests are sent asynchronously from the PO while the SO should respond with the same number of confirmations. This cannot be modeled with a simple finite state machine with two labels *request* and *confirm* (and no counters). The reason is that the language over the alphabet {*request, confirm*} consisting of all the words with equal number of *request* and *confirm* occurrences is not regular (according to a classical result in formal language theory), so there exists no finite state machine accepting it. A solution to this modeling problem can be given using an *unbounded* variable *x* that keeps track of the number of requests that were not confirmed yet. Other unbounded variables might be needed to refer to previous exchanged messages.

Though an infinite state space imposes challenges on the static verification aspect, we see it as unavoidable to extend the reach of choreography models beyond the borders of finite state machines if we want to derive rich test sequences from the models.

4.3 Sending and receiving messages

In this subsection we discuss whether in a choreography model, sending a message and receiving it should be modeled using two distinct actions or using only one atomic action. We refer to the latter as the *atomic approach*.

The atomic approach raises the question what semantics an atomic action has. Does it describe the send or the receive event? We suggest adopting neither of these options. Instead we propose a *global observer* semantics. As illustrated in **Figure 2**, a choreography model should assume the perspective of a global observer which has a limited view on the interaction between the involved parties. It can only observe the messages which are on the channel(s) between the two parties. This observation is the sole basis of his knowledge on the ordering of messages (the "global state").

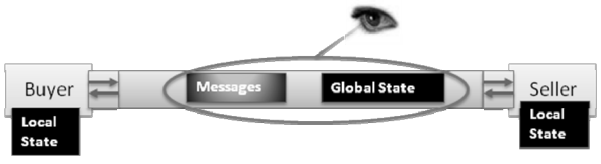

Figure 2. Global observer

In the example of Section 2, we used this atomic approach. Each transition labeled by a message type denotes the *observation* of a message of that type on the network between the parties.

In software engineering practice, it is usual that the process integration experts tend to use the atomic view of choreography interactions, whereas the developers and testers rather prefer to use a local perspective that inheritably would consider the local send and receive actions, because these are the building blocks of the communication.

Since each atomic message exchange action can be decomposed in the sending part and the receiving part, the number of transitions of a model using explicit send/receive is doubled compared to the atomic approach. If concurrency is not explicitly modeled, the increase is even higher, cf. Section 4.7. Therefore the complexity as perceived by a human modeler is significantly higher than in the atomic model.

The advantage of explicit send/receive is the ability to introduce additional constraints in the model and hence to specify more precisely the number of message exchange sequences. Let us investigate possible classes of these additional constraints:

1. Send-after-send at same partner: These are properties like "Sending a *POChangeRequest* after sending a *POCancellationRequest* is not allowed". Since the originator of both messages is the same party, such constraints can as well be covered in the atomic model; in the example by stating: "The observation of *POChangeRequest* after the observation of a *POCancellationRequest* is not possible."
2. Send-after-receive at same partner: see below.
3. Receive-after-send at same partner: "Receiving a *POConfirmation* after sending a *POCancellationRequest* is not allowed". The sending of a message cannot have an *immediate* influence on the behavior of the other partner. Such a constraint is thus obviously not enforceable.
4. Receive-after-receive at same partner: Similar to the first case.
5. Send/receive dependencies at different partners: These are properties like "Sending a *POCancellationRequest* after sending a *POConfirmation* is not allowed". Such constraints cannot be enforced in a distributed environment. It is thus an advantage of the atomic model that they cannot be expressed and a danger to abuse a model based on explicit send/receive modeling.

In the following we discuss Case 2 (Send-after-receive at same partner). All other cases discussed above clearly advocate the use of the atomic approach.

An example for Case 2 would be "Sending a POConfirmation after receiving a POCancellationRequest is not allowed". This local constraint on the seller party is illustrated in Figure 3.

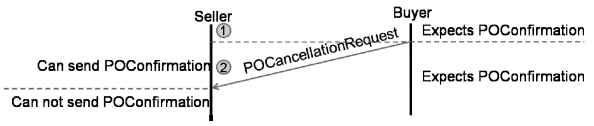

Figure 3. Send-after-receive dependency at same partner

Note that after the buyer sent the *POCancellationRequest*, the situation for the buyer remains unchanged even after the seller has received the message. (Only after receiving a confirmation on the cancellation request, this changes.)

For the buyer, the local constraint on seller is irrelevant: Even if the constraint was violated, the buyer must be able to react to a *POConfirmation*.

This is reflected by the atomic model: It may be the case that *POCancellationRequest* is observed before a *POConfirmation* since the latter was sent at time ②. Thus, in an atomic model we have to take into account, in this case, arbitrary ordering of *POCancellationRequest* and *POConfirmation*.

However, the lack of this local constraint leads to technical difficulties when executing a derived test sequence: When implementing the legal choreography sequence

(*POCancellationRequest* → *POConfirmation*)

sending of *POConfirmation* must be triggered *after* *POCancellationRequest* has been sent but *before* it arrives. The test execution tool must know how to trigger this sequence and thus has to make use of the local send-after-receive constraint.

We thus envisage a hybrid model which essentially follows the atomic approach, but which allows for the addition of explicit send-after-receive constraints only when needed.

4.4 Number of involved parties

In our running example, an existing PO can only be cancelled as long as no invoice for the order was received (via a Supplier Invoicing component), so the possibility of sending a cancellation message does not only depend on the existing communication with the SO, but also on other components, i.e. the Supplier Invoice and Customer Invoice. It is thus sometimes necessary to model choreographies that involve more than two parties because information may be lost when projecting such multi-party choreography to pair-wise choreographies. We observed however that most interactions in the investigated cases at SAP involve exactly two process components. Thus we aim to focus on pair-wise choreographies as it is supposed to ensure separation of concerns. Otherwise we may end up with

only one big (incomprehensible) model containing all constraints in a landscape of hundreds of components.

4.5 State-based vs. activity-based modeling

Two major directions can be followed for choreography modeling, as in any behavioral model: an activity-based approach and a state-based approach. In the activity-based approach, the interactions between the parties and their ordering is the primary focus whereas the state of the interaction is not explicitly modeled or only partly modeled using variables. In the state-based approach, the states of the choreography are modeled as first-class entities together with the interactions, which are then modeled as transitions between states. Since activity-based models are cluttered with variables for bookkeeping of the choreography state and the message contents, we advocate a state-based approach.

4.6 Asynchronous communication

For several reasons many SOA implementations rely on asynchronous communication. One of the reasons is that synchronous communication blocks senders until the receiver processes a message and therefore hinders the SOA paradigm of loosely coupled components. Surely synchronous communication might be inevitable in some cases but a modeling language that implicitly assumes only synchronous communication might not be applicable for choreography modeling.

4.7 Explicit concurrency

Concurrent interaction between two components exists, if both parties are able to act independently in triggering a message exchange. For interacting business components in an asynchronous environment it is quite a common pattern to negotiate while enjoying equal rights, rather than enforcing actions in a clients-server relation. It is always possible to model concurrent behavior sequentially by introducing choices that spawn every possible combination of parallel interaction. However, if the parallel interaction parts are complex, the model tends to become cluttered and error prone. By modeling concurrent interactions separately much redundancy can be omitted. Although explicit concurrency might not be a natural way of modeling for some of the users, choreography modeling should allow explicit concurrency.

4.8 Interaction termination

Since one of our main goals is to use the choreography models for test generation, we need to define where the test cases should start, respectively stop. This can be realized using an initial state, respectively final states in the model, a test case then being given by the sequence of interactions starting in the initial state and ending in a final state.

Defining final states for explicit concurrency is not always straightforward. One possibility is to explicitly define a set of global states as final. Another possibility is to define local final states for the concurrent regions and then define the global final states as the product of local final states. The first option offers more flexibility and expressiveness in defining accepted executions, but might complicate the choreography meta-model and pose difficulties in case of a visual representation.

5. Existing Choreography Approaches

Based on the requirements and properties discussed in the previous sections, we surveyed existing modeling languages that were designed or may be used for service choreography modeling. In particular, we investigated: BPMN [8],[3], BPEL4Chor [15], Let's Dance [14],[1], WS-CDL [5], and UML [17]. While BPMN, Let's dance and BPEL4Chor are rather high-level, WS-CDL and UML allow more detailed models. Moreover, only UML provides state-based modeling, while the others are activity-based.

Due to space limitations, we cannot provide the details of the survey in this paper. Although these modeling languages may not satisfy one or another of the requirements discussed before, one could to try to extend them with the necessary missing constructs. However, extending the standards to support the proper level of details is usually not a trivial task. Another demanding task is that sometimes the semantics of the standards is not the one that is needed for the choreography modeling. Finally the integration of a modeling language in the modeling landscape of a specific company like SAP that uses proprietary models at different levels of abstractions (business object, process component, or scenario level) needs detailed consideration.

6. Conclusion

In this work we discussed the message choreography modeling from the point of view of software development for SOA-enabled ERP systems, with a focus on quality assurance. We have presented a choreography case study within an ERP system and further identified and discussed a number of properties that choreography models should have in order to facilitate acceptance of developers, derivation of integration tests, and static verification. Especially we pointed out the need for a dedicated message constraint language and investigated whether message sending and receiving should be modeled explicitly or

atomically for these purposes. By investigating existing choreography languages we have discovered the need for a language suiting the discussed aspects. We are currently researching how a choreography modeling language satisfying the requirements sketched in this paper would fit into the software development and quality assurance modeling landscape. This work comprises the prototyping of integrated modeling tools supporting model-based testing and verification. The detailed description of the developed domain-specific language satisfying the requirements is the subject of a future paper.

Acknowledgements: This work was partially supported by the EC-funded projects *MODELPLEX*[1], *Deploy*[2], and *VIDE*[3] (grants no. 034081, 214158, and 033606). We thank Frank Michael Kraft for his valuable feedback on this work.

References

[1] A. Barros, M. Dumas, A. ter Hofstede, Service interaction patterns, In Proc. of *Conf. on Business Process Management (BPM'05),* Springer, 2005, pp. 302–318.

[2] C. Benedetto, SOA and integration testing: The end-to-end view, *SOA World Magazine* 6(8), 2006. Online at: *http://webservices.sys-con.com/read/275057.htm*

[3] G. Decker and A. Barros, Interaction modeling using BPMN, In Proc. of *Business Process Management Workshop 2007 (BPM'07)*, LNCS 4928, Springer, 2007, pp. 208-219.

[4] G. Decker and M. Weske, Behavioral consistency for B2B process integration. In Proc. of *19th Int. Conf. on Advanced Information Systems Engineering (CAiSE'07)*, LNCS 4495, Springer, 2007, pp. 81-95.

[5] N. Kavantzas, D. Burdett, G. Ritzinger, Y. Lafon, Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation, Technical report, 2005, Online at: *http://www.w3.org/TR/ws-cdl-10*

[6] D.E. O'Leary, *Enterprise resource planning systems-systems, life cycle, electronic commerce and risks.* Cambridge University Press, 2000.

[7] C. Métayer, J. R. Abrial, and L. Voisin, *Event-B Language*, RODIN Deliverable D7 [Rodin], 2005. Available at *http://rodin.cs.ncl.ac.uk*

[8] Object Management Group (OMG), *Business Process Modeling Notation (BPMN) Specification*, Final Adopted Specification, Technical Report, 2006. Online at: *http://www.bpmn.org/*

[9] SAP AG, *Enterprise SOA in a Nutshell*, 2007. Online: h*ttp://help.sap.com/redirect_sdn_esoa/redirect_esoainanutshell.htm*

[10] M. Utting and B. Legeard, *Practical model-based testing, a tools approach*. Morgan Kaufmann Publ., 2007.

[11] S. Wieczorek, A. Stefanescu, and J. Großmann, Enabling model-based testing for SOA integration testing, In Proc. of *Model-based testing in practice (MOTIP'08)*, satellite workshop of ECMDA'08, 2008, pp. 77-82.

[12] S. Wieczorek, A. Stefanescu, and I. Schieferdecker, Test data provision for ERP systems, In Proc. of *Int. Conf. on Software Testing, Verification and Validation (ICST'08)*, IEEE Computer Society, 2008, pp 396-403.

[13] D. Woods, and T. Mattern*, Enterprise SOA - Designing IT for Business Innovation*, O'Reilly, 2006.

[14] J.M. Zaha, A. Barros, M. Dumas, A. ter Hofstede, A Language for Service Behavior Modeling, In Proc. of *Int. Conf. on Cooperative Information Systems* (CoopIS'06), Springer, 2006.

[15] J.M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, G. Decker, Service interaction modeling: bridging global and local views. In Proc. of *Enterprise Distributed Object Computing. (EDOC'06)*, IEEE Computer Society, pp. 45-55.

[16] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, M. Weske, Modeling service choreographies using BPMN and BPEL4Chor. In Proc. of *Advanced Information Systems Engineering (CAiSE'08)*, LNCS 5074, Springer, pp. 79-93.

[17] The Unified Modeling Language (UML). Object Management Group (OMG). *http://www.uml.org.*

[1] *http://www.modelplex-ist.org*
[2] *http://www.deploy-project.eu*
[3] *http://vide.tnmsoft.de*