Modeling and Verification of Reliable Messaging by Graph Transformation Systems [4]

László Gönczy [1], Máté Kovács [2] and Dániel Varró [3]

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary

Abstract

Due to the increasing need of highly dependable services in Service-Oriented Architectures (SOA), service-level agreements include more and more frequently such non-functional aspects as security, safety, availability, reliability, etc. Whenever a service can no longer be provided with the required QoS, the service requester needs to switch dynamically to a new service having adequate service parameters after exchanging a sequence of messages. In the current paper, we first extend the core SOA metamodel with parameters required for reliable messaging in services. Then we model reconfigurations for reliable message delivery by graph transformation rules. Finally, we carry out a formal verification of the proposed rule set by combining analysis tools for graph transformation and labeled transition systems.

Keywords: Service Oriented Architecture, Graph Transformation, Reliable Messaging

1 Introduction

Service-Oriented Architectures (SOA) provide a flexible and dynamic platform for implementing business-critical services. The main business-level driver of the SOA paradigm is componentization, which raises the level of abstraction from objects to services in the design process of distributed applications. The main architectural-level driver of the SOA paradigm is to provide a common middleware framework for dynamic discovery, interaction and reconfiguration of service components independently of the actual business environment.

[1] Email: gonczy@mit.bme.hu
[2] Email: km432@hszk.bme.hu
[3] Email: varro@mit.bme.hu
[4] This work was partially supported by the SENSORIA European project (IST-3-016004).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Recently, the identification of non-functional parameters of services have been addressed by various XML-based standards related to web services (such as WS-Reliable Messaging, WS-Reliable Messaging Policies, etc.). *Reliable messaging between services* — where the delivery of a message can be guaranteed by the underlying platform by appropriate reconfiguration mechanisms — plays an important role in many of these standards, because of the growing need for asynchronous yet reliable Web service invocations. Despite the wide range of standards addressing the specification of these reliability service properties, currently only very experimental solutions exist in the industry (such as RAMP-Toolkit [18] by IBM or RM4GS [22] by a consortium led by Fujitsu-Siemens, Hitachi and NEC) that actually implement these reconfigurations in order to maintain the required level of reliability.

In the current paper, we conceptually follow [2] where a semi-formal platform-independent and a SOA-specific metamodel (ontology) was developed to capture service architectures on various levels of abstraction in a model-driven service development process. Furthermore, reconfigurations for service publishing, querying and binding were captured by graph transformation rules [6], which provides a visual yet formal, rule and pattern-based specification formalism widely used in various application areas. This combination of metamodeling and graph transformation rules fits well to a model-based development process for service middleware.

This paper extends the core metamodel defined in [2] (and overviewed in Sec.2) by a new package for reliable messaging (Sec. 3.2). Moreover, we provide new high-level reconfiguration primitives for reliable message delivery in the form of graph transformation rules (Sec. 4.2) by integrating dependability techniques [16]. Finally, we carry out a formal verification of the proposed rule set by combining various analysis tools: the state space of the graph grammar will be first explored by GROOVE [19] while the generated graph transition system is transformed into a fromat accepetd by the Labeled Transition System Analyzer (LTSA) tool where the automated formal verification of certain safety properties is carried out. Our aim is to provide a generic way to capture the dynamic fault-tolerant behavior of a SOA. In the current paper we used the reliable messaging as a case study for this.

Note that we first modeled reconfiguration rules for reliable messaging by graph transformation rules in an ad hoc way in [10]. The current paper extends that approach by formally verifying the rules by integrating analysis tools (Sec 5.3). In fact, we managed to find conceptual flaws in this initial rule set during verification, and thus the current paper already presents the corrected version of the rules (in Sec. 4.2).

2 Core SOA Metamodel

The main architectural concepts of the domain of service-oriented architectures are captured by a corresponding metamodel. An extract of the meta-

model of "core" SOA functionality is shown in Fig. 1. It is based on the metamodel presented in [2], with minor simplifications and modifications to keep the current paper better focused.

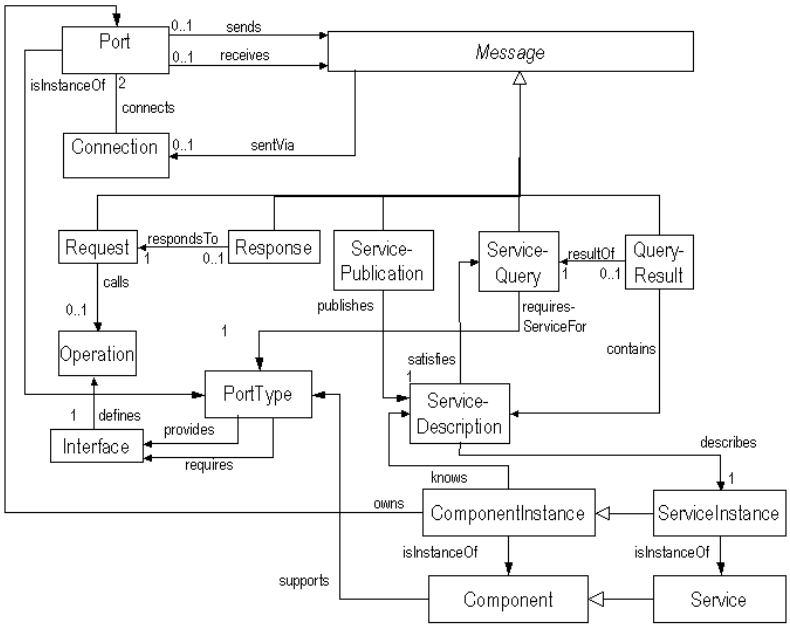


Fig. 1. Core metamodel of SOA

The core model to service-oriented architectures consists of the following main elements:

- A *component* is a basic "module" in the system which provides a service.
- A *service* is a set of functionalities with well-defined ports and interfaces. Note that in the paper, we merge the notions of service (and component) types and service instances into a single service (component) concept for the sake of simplicity.
- A *port* is the communication "endpoint" (with a set of abstract operations and messages) where a service can be accessed.
- A *connection* denotes a bidirectional channel between two ports at run-time.
- An *operation* is an "atomic" action with input and output messages. There can be multiple operations defined on the same port. .
- A *message* is a set of parameters with pre-defined subtypes such as *request*, *response*, *service publication*, *service query* and *query results*. For the current paper we treat these messages on an abstract level regardless of their actual subtypes. However, we will derive additional subtypes in Sec. 3.2 required for reliable messaging.
- A *service description* is a descriptor file containing all necessary information about the runtime cooperation with the service, such as description of port, operations, messages, etc.

3 Extensions for Reliable Messaging in Web Services

In this section, after a brief overview on capturing non-functional requirements in existing web service technologies, we extend the core SOA metamodel by non-functional attributes required for reliable messaging in order to provide a model-based solution.

3.1 Non-functional Requirements in Existing Web Service Technologies

While there are several initiatives to define the so-called "non-functional" properties of services, such as Web Services Modeling Ontology [25], W3C Web Services Architecture [24], DublinCore Metadata for ServiceDiscovery [5], the terminology is still ambiguous.

To illustrate the modeling of non-functional properties by a practical and simple example, hereby we present a model-based reconfiguration for reliable messaging to tolerate communication faults. As the consumers of the Web services are not aware of the details of underlying network protocol, the semantics of the message delivery has to be specified at the application level as requirements for reliable messaging. This needs a platform-independent representation of message attributes, which is reflected by a number of emerging standards [26,27]. Some reference implementations for popular application servers like IBM WebSphere or Apache Tomcat are available.

These industrial standards and initiatives usually suppose that the service provider signs a contract with each client about the Quality of Service, measured in terms such as average response time, minimal throughput, type of message delivery, etc. These contracts are typically identical for classes of similar clients (roles), for instance, Golden User, Business Partner, Individual Customer, etc. The runtime service instances send their messages according to these contracts, while additional information, including such non-functional aspects, is hidden from the application layer. As a consequence, it is not necessary to modify the original service clients on the consumers' side.

Additional information is handled by components aware of reliability attributes, called *"Reliable Message Endpoints"*. In technological terms, the header of SOAP envelopes is extended with some attributes by a "Reliable Message Endpoint" on the provider's side, which are then removed from the messages by another "Reliable Message Endpoint" at the client side. Since the concrete format of these attributes in message headers is out of scope, here we model an abstract envelope concept. In the future, we plan to map such concepts into existing technologies by model transformation techniques.

3.2 Metamodel Extensions for Reliable Messaging in Services

Now we extend the core SOA metamodel of [2] to capture properties of reliable messaging between services. After enriching the domain metamodel, our long term goal is to define a corresponding UML profile to provide extensions to the

UML language tailored to a specific application domain by introducing domain concepts, attributes and relations in the form of stereotypes and tagged values. However, the current paper only focuses on metamodel-level extensions for reliable messaging in the SOA metamodel.

We first derive a subclass from SOA element in the reliable SOA metamodel, and then create an association from the child class (e.g. RelMsgEnvelope) to the parent class (e.g. Message) in addition. As a result, unreliable messaging can be carried out by the original SOA reconfiguration rules defined in [2]. Furthermore, the original messages are kept but wrapped into an envelope by introducing a new association. As a consequence, only very minor extensions are required to the rules of [2] to transport these envelopes between services to properly memorize the sender and the receiver of a message.

The extensions of the SOA metamodel for reliable messaging is presented in Fig. 2:

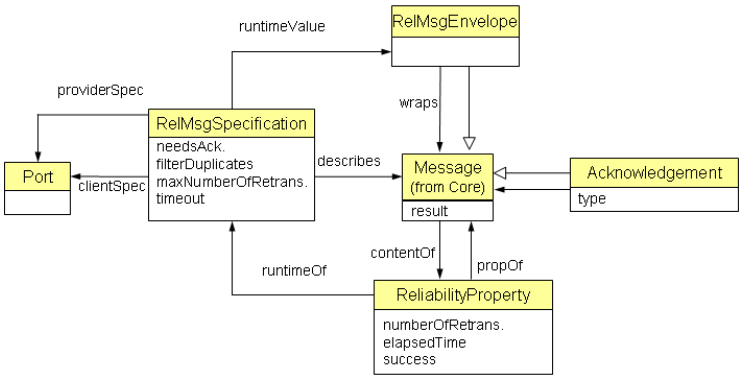


Fig. 2. Metamodel of Reliability Extensions

- RelMsgSpecification (shortly, RelSpec) is a class for specifying the requirements for reliable messaging between SOA services (see association describes, clientSpec, providerSpec).
  - Attribute needsAck is a boolean value to express if an acknowledgement should be sent to a message. If an acknowledgement arrives to the sender for a message, then it is guaranteed that the message is received at least once.
  - Attribute filterDuplicates is a boolean value to express that a message should be accepted and processed by the receiver at most once.
  - Attribute timeout is a timer constraint which specifies how much the sender waits for the acknowledgement of a message before retransmission.
  - Attribute maxNumberOfRetrans is an integer which puts an upper limit on how many times a message can be retransmitted by the sender due to the lack of acknowledgement from the receiver.
- RelMsgEnvelope (shortly, Envelope) is a subclass of core SOA Message which serves as an envelope for wrapping up the real message to be sent (wraps).
- ReliabilityProperty (shortly, RelProp) contains the runtime properties of a

message:
· Attribute numberOfRetrans is a serial number for the envelope which is increased by one each time the same message is retransmitted.
· Attribute timeElapsed denotes the time elapsed since the (last) transmission of a message.

The content of the message is also attached to the properties (contentOf) since the retransmission of the message has to be transparent for the application.

- Acknowledgement (shortly, Ack) is a subclass of core SOA Message which denotes an acknowledgement sent in response to a message.

As this extension is closely related to existing standards, we plan to map such high-level models into implementations of these standards following a model-driven approach: runtime values of XML descriptors will be derived from the attributes of our model.

3.3 Semantics for Message Delivery

In traditional distributed systems, communication middleware have to guarantee the desired semantics of message delivery. The most common semantics are the following:

- *At-Least-Once* is one of the weakest, requiring that every message has to arrive to the receiver at least once. This does not exclude the possibility of sending a message multiple times.
- *At-Most-Once* is ensuring that a message won't be sent more than once, which means the elimination of duplicates.
- *Exactly-Once* is the "subset" of the previous ones both messaeg delivery and filtering of duplicates are guranteed.

There are of course other semantics, hereby we will use *At-Least-Once* as a running example since this is the easier to present. However, our methodology naturally works for the other delivery semantics as well.

4 Reconfiguration for Reliable SOA Messaging by Graph Transformation

We now propose to describe the reconfiguration mechanisms of reliable SOA messaging by graph transformation rules (conceptually following [2]).

4.1 Overview of Graph Transformation

A main benefit of using graph transformations as a formal specification paradigm for capturing reconfiguration rules is that they are visual, intuitive, therefore they can be understood by service engineers as well. The interested reader

may find a detailed theoretical discussion of graph transformation in [6], here we present just a brief overview on it.

Furthermore, graph transformation allows dynamic metamodeling [12] in a certain domain. The high-level (ontological) concepts are visualized as UML class diagrams while graph patterns are considered to be UML object diagrams to express that concrete models are instances (objects) of the metamodel (classes) combining the advantage of precise modeling and visual design.

A graph transformation rule consists of a *Left Hand Side (LHS)*, a *Right Hand Side (RHS)* and optionally a *Negative Application Condition (NAC)*. The *LHS* is a graph pattern consisting of the *mandatory* elements which prescribes a precondition for the application of the rule. The *RHS* is a graph pattern containing all elements which should be present after the application of the rule. Elements in the $RHS \cap LHS$ are left unchanged by the execution of the transformation, elements in $LHS \setminus RHS$ are deleted while elements in $RHS \setminus LHS$ are newly created by the rule. The fulfillment of the negative condition prevents the rule from being executed on the particular matching. Hereby we follow the Single Pushout Approach (SPO) approach [6] with negative application conditions [11].

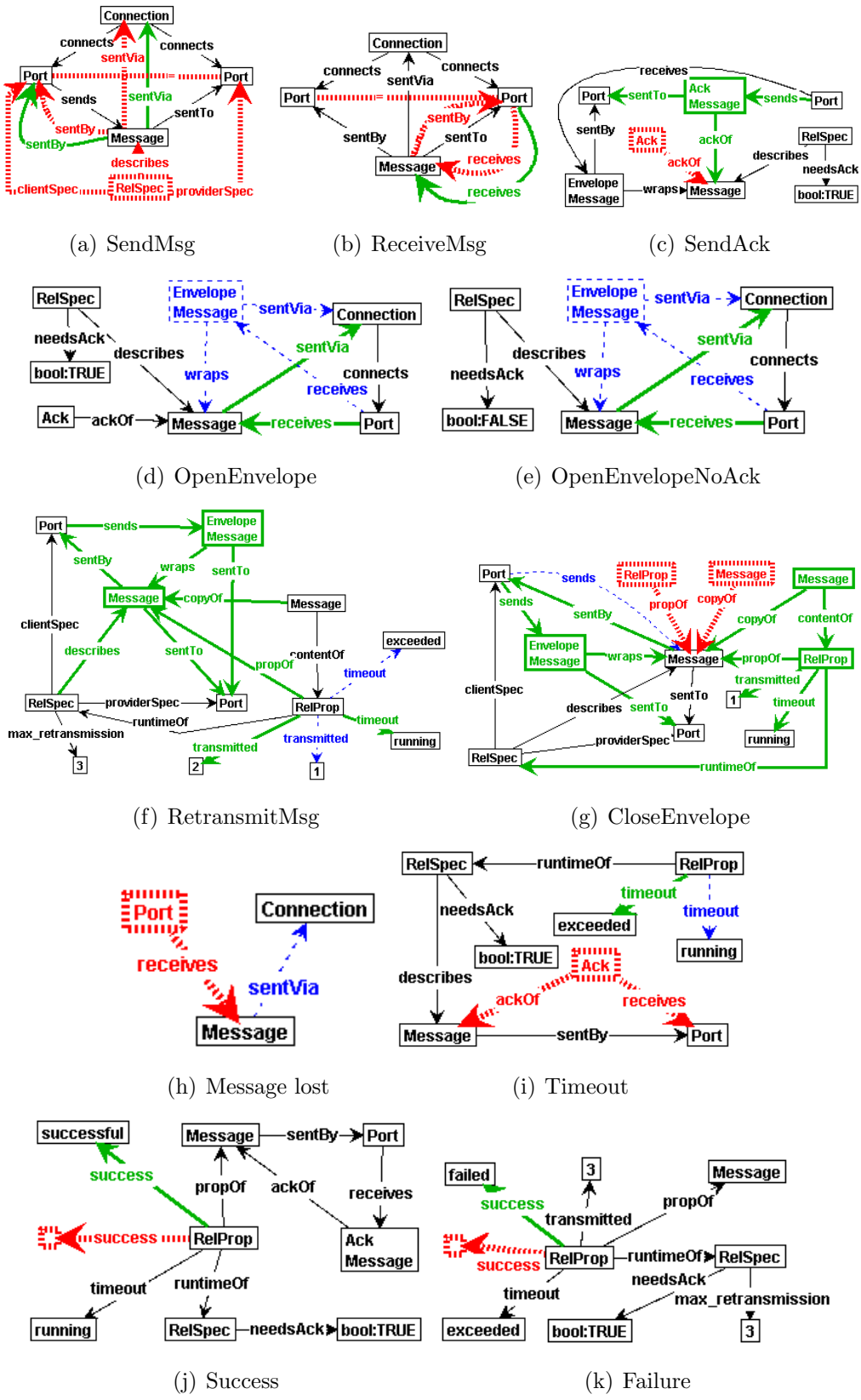
A *graph grammar (GG)* consists of a start graph and a set of graph transformation rules. A *graph transition system (GTS)* represents the state space generated by a graph grammar. The different states of the GG (i.e. the derived instance graphs) appear as nodes while edges denote state transition caused by the application of a graph transformation rule. An edge going from state *s1* to state *s2* with label *r,o* represents that from the graph instance *s1* one can get graph instance *s2* by the application of transformation rule *r* at match *o*.

In this paper, we use a compact visualization of graph transformation rules (first introduced in the Fujaba framework [8] and used in Groove [19]), when the entire rule is merged into a single pattern. Newly created elements are denoted by solid thick (green) lines (tagged as {*new*} in the editor) while deleted elements are depicted by dashed blue lines (tagged as {*deleted*}). Elements in the intersection of the *LHS* and the *RHS* are visualized normally (in black), and elements of NAC appear in thick dotted (red) lines. A negative condition is used in the current paper to prevent the rule from creating infinite number of new elements on the same matching (e.g. in the case of messaging, the same message is received only once).

4.2 Reconfiguration Rules

The reliable messaging with at least once message delivery can be assured by the reconfiguration rules captured by graph transformation in Fig. 3 (using the Groove notation).

First, the normal messages have to be packed into and wrapped from en-



(a) SendMsg (b) ReceiveMsg (c) SendAck

(d) OpenEnvelope (e) OpenEnvelopeNoAck

(f) RetransmitMsg (g) CloseEnvelope

(h) Message lost (i) Timeout

(j) Success (k) Failure

Fig. 3. Transformation rules in GROOVE for reliable messaging

velopes (as in the case of present reliable messaging technologies). Thus, the messages are wrapped up in the sender side instead of being transmitted (rule closeEnvelope in Fig. 3(g)) and envelopes are opened before receiving their content at the receiver side (rules openEnvelope and openEnvelopeNoAck in Fig. 3(d) and Fig. 3(e) where the two separate rules depend on whether a message needs an acknowledgement). As the most general type is used for messages, these rules will match for instances of every subclass of message class with a reliability specification. Thus, reliable messaging is also provided for asynchronous service invocations, discovery queries, etc with a typed, attributed graph transformation engine.

Basic delivery modes include *AtLeastOnce*, *AtMostOnce* and *ExactlyOnce*, determined by the parameters needsAck and filterDuplicates, respectively. Hereby we consider 'primitives' as basic operation, supported by the runtime Web service platform (such as RAMP) to ensure the desired delivery semantics.

At the sender side, there are basically two message sending modes, depending on the value of the needsAck parameter of the RelSpec object describing the requirements for messaging. If this parameter is true, reliable message sending required for a particular message, which corresponds to the *AtLeastOnce* messaging semantics. In this case, the sender will wait for an acknowledgement and consider the transmission of a message successful only if the acknowledgement arrives within the timeout interval. The rule of the successful message transmission (more precisely, the arrival of an acknowledgement in time) is shown in Fig. 3(j).

On the other hand, if the acknowledgement does not arrive in time (rule *Timeout*, Fig. 3(i)), then the next action (i.e. the next rule to be applied) depends on the number of retransmitted messages. If the actual retransmission number of a particular message is smaller than the allowed, then a new instance of the Envelope class is created and sent with the same content and a higher retransmission number (rule *RetransmitMsg*, Fig. 3(f)). If the same message content cannot be sent again (precondition of rule *TransmissionFailure*, Fig. 3(k)), then the transmission of the message is considered to be failed. Note that if no acknowledgement is needed, then no additional rules are applied at message sending, only the core *SendMsg* rule matches the instance graph.

On the receiver side, the messages are acknowledged if needed (see rule *SendAck* in Fig. 3(c)), otherwise the core *ReceiveMsg* rule is applied (Fig. 3(b)).

Additional rules have been introduced to inject faults into the system according to a fault model. In our fault model, we assume that the message may be lost during submission (Fig. 3(h)), or it eventually arrives but a timeout has already occured (Fig. 3(i)). Acknowledgements can also be lost.

These fault injection rules of Fig. 3 are an extension of [10]. Furthermore, since we used a richer graph model in our previous work, all the rules had to be translated into Groove manually (see Sec. 5.2). Finally, during verification, we also found conceptual flaws in the original rule set. For instance, there we

erroneously allowed a message to be received by the sender party itself. These changes are already included in Fig. 3.

5 Verification of Reliable Messaging Rules

5.1 Verification Tool Chain

The transformation rules were implemented in the Groove [19] tool, which supports the generation of the state space (i.e. a Graph Transition System - GTS) derived by a graph grammar. Using the Groove simulator, one can manually inspect the state space from a given start graph for verification purposes. While this is convenient for early tests of the GT specification, this is not very convincing in case of large state spaces. Unfortunately, the current public version of Groove (March, 2006) that we used in our experiments did not yet support the verification of CTL-like properties (reported recently in [20]).

For this reason, we decided to carry out the verification of the reconfiguration rules for reliable messaging by post-processing the generated GTS in the Labeled Transition System Analyzer (LTSA, [15]) tool. This tool supports the safety, deadlock and liveness analysis of Labeled Transition Systems. A requirement to be verified is defined by a normal *(requirement) process*, which explicitly captures correct and incorrect execution paths (wrt. a subset of actions) or a *property* from which the corresponding process is generated automatically by the tool. For verification runs, the requirement process and the system process are composed concurrently. The result of verification is either successful or a counterexample is provided in the form of a transition sequence which leads to the violation of the requirement.

In order to project GTSs into the input format of LTSAs, a translator was implemented which takes the GXL input of the GTS generated by Groove and creates LTSA processes accordingly.

Furthermore, since the LTSA analyzer always checks for the existence of deadlocks (even if a deadlock means correct termination of the system), we had to guarantee that the GTS is cyclic by introducing additional "restart" graph transformation rules. These rules are applicable to any configuration and delete all information regarding to the state of the system. Alternatively, this also could be done by implementing an extension in the translator from GTS to LTSA to create loops on the final states.

5.2 Groove-specific Adaptations of Transformation Rules

In order to encode the transformation rules into Groove (see Fig.3), we had to model concepts such as inheritance, types and instantiation in Groove which supports only labeled edges between nodes. Therefore, the types of the nodes were modeled as self-edges, and we used multiple edge labels in case of inheritance. For instance, the object in the top of Fig. 3(c) has a type of

Acknowledgement (shortly Ack) which is a specialization of Message. Concrete attribute values (such as the counter of the transmitted messages) were implemented as nodes, linked to their container nodes.

As the current version of Groove had some minor bugs which prevented the correct handling of primitive datatypes such as integers, the required constant values were inserted as individual nodes (e.g. node with the self-edged "exceeded" represented the string "exceeded", etc.). For the same reasons, we had to properly duplicate the rules which used comparison operations on integer values. For instance, sending a message for the second and third was implemented as two different rules.

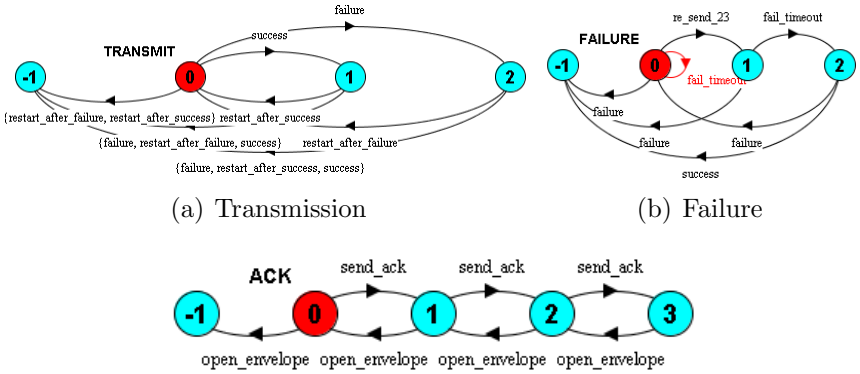
For the verification of these rules, the graph grammar had to be extended to ensure that *it will have an infinite lifecycle*. We ensured the start state could be reached from any subsequent state by systematic modifications of the rules. Firstly, two new rules were created to restore the start state of the system after the (either successful or a failed) transmission of a message. Secondly, three auxiliary transformations were implemented to delete the unnecessary elements such as messages, envelopes and acknowledgements to keep the state-space finite. Third, all other rules were extended by a NAC containing the success attribute of the MessageProperty class to prevent these rules from being concurrently executed with the initialization sequence.

5.3 Verification of Properties

We identified the following (non-exclusive) list of important requirements for reliable messaging:

- The transmission of a message is either successful or failed (but the submission has a definite result).
- The transmission is considered to be failed exactly when the timeout of the acknowledgement for the last transmittable message instance is exceeded.
- Incoming messages are read only after being acknowledged (if acknowledgement is required).
- Multiple messages of the same port (to the same or different ports) are managed correctly, i.e., their runtime properties are handled serapately.
- Sending and receiving normal (unreliable) messages can still be carried out.

When formalizing these requirements in LTSA, we ran into two main problems. On the one hand, the GTS generated by Groove only contains the applied rule as labels but no information is provided on the occurrence. For this reason, we can capture only those requirements where the identity of messages are irrelevant. In several cases, only a weakened form of the requirement was actually verified due to this problem. In our opinion, providing also information for the matching is an interesting direction for future improvements in Groove.



(a) Transmission (b) Failure

(c) Acknowledgement

Fig. 4. Automata of the properties (-1 represents the error state)

On the other hand, LTSA offers a limited way for checking liveness properties, therefore, our attention was mainly focused on to verify safety properties of reliable messaging.

The main benefits of using these two tools together was the easy generation of the available states and an automated check of properties. The previous requirements were interpreted and formalized in the form of LTSAprocesses / properties (see Fig. 4) as follows.

- The transmission of a message is either successful or failed: Exactly one of the transformation rules Success, Failure is applied in each path to the restart state(Fig. 4(a)).
- The transmission is considered failed exactly in the case when the timeout of the acknowledgement of the last transmittable message instance is exceeded (Fig. 4(b)).
- Incoming messages are read only after being acknowledged, i.e. the application of the SendAck rules precedes that of the OpenEnvelope rule. Note that if no acknowledgement is needed, then the property is not violated as OpenEnvelopeNoAck rule is applied instead (Fig. 4(c)).

By carefully selecting initial models to capture small but typical configurations, we were able to verify that our reconfiguration rules fulfill these requirements. We believe that these models are minimal but representative configurations which could be extended if this didn't raise a conflict with the limitation of the state space generation. On the other hand, some of our preliminary expectations turned out to be false during the verification (for instance, that a message lost would always cause timeout).

The main lessons we learned from this verification case study for reconfiguration rules used in reliable messaging are the following:

- *High-level vs. low-level graph models and rules:* We were able to translate (by hand) rich graph transformation rules and models (as used in [10] with inheritance, types, attributes, etc.) into lower level verification mod-

els (used in Groove) with relatively simple modeling tricks. Problems have mainly arisen in case of integer attributes where rules have to be copied to handle all different matches of the integer attributes. Future support for core datatypes in Groove would further reduce the complexity of this last task.

- *Testing in Groove:* Minor conceptual flaws have been identified in the rules of [10] by manually inspecting the generated GTS for smaller examples. However, such manual inspection for deciding the correctness of a property was infeasible for large state spaces.
- *Verification in LTSA:* Automated post-processing in LTSA is a feasible solution for verifying meaningful safety properties with obvious limitations due to the lack of identity information in the GTS generated by Groove.

6 Related work

Related work in this field usually concentrate either on describing the non-functional attributes of services, or modeling dynamic aspects of Service Oriented Architectures by graph transformation.

Our work conceptually follows the approach of [2] for specifying services in SOA. The authors of [3] describe the application of graph transformations in the runtime matching of behavioral Web service specifications. In [13], the conformance testing of Web services is based on graph transformations, focusing on the automated test case generation. However, none of these works discusses the aspects of reliable messaging. Our aim was to utilize the benefits of this approach by extending the metamodel and the transformation rules.

Graph transformation is used as a specification technique for dynamic architectural reconfigurations in [7] using the algebraic framework CommUnity. Hirsch uses graph transformations over hypergraphs in [14] to to specify runtime interactions among components, reconfigurations, and mobility in a given architectural style. However, the problem of reliable messaging in SOA is not addressed in either case.

LTSA [15] has already been applied successfully for the formal analysis of business processes given in the form of BPEL specifications in [9], but reliable messages are not considered in these papers.

In the future, we plan to investigate the use of other verification tools for graph grammars. A primary candidate is Augur [21], which uses unfolding techniques to derive a finite approximation of possible traces in a GTS.

The specification and analysis of fault behaviors have been carried out in [4] using graph grammars. While this approach is not directly related to SOA, it may serve as a starting point for incorporating additional dependability aspects for our research.

Reliable messaging were verified in other papers, (for instance, [1]), our technique differs mostly in the level of modeling, which is closer to that of the

usual SOA description, and therefore, is more appropriate to apply for the verification of fault tolerant mechanisms in SOA.

Finally, in the industrial field, there are existing and emerging specifications and technologies like [26,27]. However, their use is still ad-hoc and no model-based design-time support is available for reliable messaging.

7 Conclusion

In this paper we first proposed an extension to the core SOA metamodel of [2] and a technique to capture the reconfiguration mechanisms to enhance the development of more robust SOA middleware. Reconfiguration rules for reliable messaging in SOA have been captured by graph transformation rules.

Then the correctness of these rules were verified by first generating the state space of the graph grammar by Groove (in the form of graph transition systems), then transforming it into labeled transition system in order to carry out formal verification of correctness requirements using the LTSA analyzer tool.

As the next step in the future, we plan to focus on bridging the gap between our abstract reconfiguration rules and existing implementation technologies for reliable web services. The formally verified set of reconfiguration rules will definitely serve as a sound starting point for this activity. Our long term goal is to automatically derive implementations of reliable messaging on various existing platforms based directly upon provenly correct dynamic reconfiguration mechanisms.

References

[1] P. D'Argenio, J.-P. Katoen, T. Ruys, and J. Tretmans. *Modeling and verifying a bounded retransmission protocol.* In Proc. of COST-247 Int. Workshop on Applied Formal Methods in System Design, 1996, Slovenia.

[2] L. Baresi, R. Heckel, S. Thöne and D. Varró, *Style-Based Modeling and Refinement of Service-Oriented Architectures.* To appear in Journal of Software and Systems Modelling, 2006.

[3] A. Cherchago and R. Heckel. *Specification Matching of Web Services Using Conditional Graph Transformation Rules.* In Proc. of Int. Conference on Graph Transformations, 2004, LNCS Vol.3256., Springer, pp. 304-318.

[4] F. L. Dotti, L. Ribeiro, and O. M. dos Santos. *Specification and analysis of fault behaviours using graph grammars.* In Applications of Graph Transformations with Industrial Relevance, Second Int. Workshop, AGTIVE 2003, USA, vol. 3062 of LNCS, pp. 120–133. Springer, 2003.

[5] DublinCore Metadata Initiative, 2006. http://dublincore.org/

[6] H. Ehrig, R. Heckel, M. Korff, M. Lowe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic approaches to graph transformation, Part II: Single pushout approach and comparison with double pushout approach.* In Handbook of Graph Grammars and Computing by Graph Transformation, volume I: Foundations, pp. 247–312. World Scientific, 1997.

[7] M. Wermelinger and J. L. Fiadeiro. *A graph transformation approach to software architecture reconfiguration.* Science of Comp. Progr., 44(2):133–155, 2002.

[8] T. Fischer, J. Niere, L. Torunski and A. Zündorf, "Story Diagrams: A new Graph Transformation Language based on UML and Java", Proc. Theory and Application to Graph Transformations (TAGT'98), vol. 1764 of LNCS, 2000, Springer.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer. *Model-based verification of web service compositions.* In 18th IEEE Int. Conf. on Automated Software Engineering (ASE 2003), Canada, pp. 152–163. IEEE Computer Society, 2003.

[10] L. Gönczy, D. Varró: *Modeling of Reliable Messaging in Service Oriented Architectures*, In Proc. Int. Workshop on Web Service Modeling and Testing (WS-MATE 2006). To appear.

[11] A. Habel, R. Heckel, and G. Taentzer. *Graph grammars with negative application conditions.* Fundamenta Informaticae, 26(3-4):287313, 1996.

[12] J.H. Hausmann, Heckel, R., Lohmann, M.: *Model-based Discovery of Web Services*, In Proc. of the IEEE Int. Conference on Web Services (ICWS), June 6-9, 2004, USA,

[13] R. Heckel, and L. Mariani. *Automated Conformance Testing of Web Services.* In Proc. of 8th Int. Conference on Fundamental Approaches to Sofware Engineering (FASE 2005), vol. 3442 of LNCS, Springer, pp. 34-48.

[14] D. Hirsch. *Graph transformation models for software architecture styles.* PhD thesis, Departamento de Computacion, Universidad de Buenos Aires, 2003.

[15] Labelled Transition System Analyser (Version 2.2) http://www-dse.doc.ic.ac.uk/concurrency/ltsa-v2/index.html

[16] J. Laprie, B. Randell, C. Landwehr, *Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE Transactions on Dependable and Secure Computing, (Vol.1, No.1.) (2004) pp. 11-33

[17] Object Management Group. Model Driven Architecture. http://www.omg.org/mda

[18] Reliable Asynchronous Message Profile (RAMP) Toolkit, IBM alphaworks. http://www.alphaworks.ibm.com/tech/ramptk

[19] A. Rensink, *The GROOVE simulator: A tool for state space generation.* In Proc. of Application of Graph Transformations with Industrial Relevance (AGTIVE'03), 2003, LNCS Vol.3062, Springer, pp. 479-485.

[20] Kastenberg H., and Rensink A. *Model Checking Dynamic States in GROOVE.* In Proc. of the 13th Int. Workshop on Software Model Checking (SPIN'06), Volume 3925 of LNCS, Springer-Verlag, 2006, to appear.

[21] B. König and V. Kozioura. *Augur - a tool for the analysis of graph transformation systems.* Bulletin of the EATCS, vol. 87:pp. 126–137, 2005.

[22] RM4GS Reference Guide, Version 1.0, FUJITSU LIMITED, Hitachi, Ltd. and NEC Corporation, 2004.
http://xml.coverpages.org/rm4gs20041125-reference.pdf

[23] Software Engineering for Service-Oriented Overlay Computers (SENSORIA) European project IST-3-016004. http://sensoria.fast.de

[24] Web Services Architecture, 2004. http://www.w3.org/TR/ws-arch/

[25] Web Service Modeling Ontology (WSMO), W3C Member Submission, 2005. http://www.w3.org/Submission/2005/SUBM-WSMO-20050603/

[26] Web Services Reliable Messaging Protocol (WS-ReliableMessaging) BEA Systems, IBM, Microsoft Corporation, Inc, and TIBCO Software Inc., 2002.

[27] Web Services Reliable Messaging TC WS-Reliability 1.1 Committee Draft 1.086, OASIS Open Consortium, 2004.