Formal Modelling and Analysis of Business Information Applications with Fault Tolerant Middleware

Submission Category: Practical Experience Report
Material has been cleared through the author affiliations
approximate word count: 4500

Jeremy Bryans, John Fitzgerald, Alexander Romanovsky
School of Computing Science
Newcastle University
Newcastle upon Tyne NE1 7RU, UK
Firstname.Lastname@ncl.ac.uk
t: +44 191 222-7999
f: +44 191 222-8788
Affiliation: Newcastle University

Andreas Roth
SAP Research CEC Darmstadt
SAP AG, Bleichstr. 8,
64283 Darmstadt, Germany
Andreas.Roth@sap.com
t: +49 6227 7-68469
f: +49 6227 78-47155
Affiliation: SAP AG

KEYWORDS: Verification, Fault Assumptions, Service-Oriented Architectures, Event-B, Tool Support

Abstract

Distributed information systems are critical to the functioning of many businesses; designing them to be dependable is a challenging but important task. We report our experience in using formal methods to enhance processes and tools for development of business information software based on service-oriented architectures. In our work, which takes place in an industrial setting, we focus on the configuration of middleware, verifying application-level requirements in the presence of faults. In pilot studies provided by SAP, we used the Event-B formalism and the open RODIN tools platform to prove properties of models of business protocols and expose weaknesses of certain middleware configurations with respect to particular protocols. We then extended the approach to use models automatically generated from diagrammatic design tools, opening the possibility of seamless integration with current development environments. Increased automation in the verification process, through domain-specific models and theories, is a goal for future work.

1. Introduction

Many business information applications are large-scale software systems that provide essential support to companies in their business processes; their design is therefore a demanding but important task. The software engineering challenge is to integrate various organisational parts and functions into one large and complex but consistent system. The principles of service-oriented architecture (SOA) can help to master this complexity. In SOA systems, such as manufactured by SAP, complex applications are composed from independent business components that offer enterprise services.

Although SOA can help to manage complexity, the design decisions for business information applications, especially those made at early stages, are critical because, if made wrongly, they can be expensive to correct later. This is particularly true of decisions relating to the dependability characteristics of components and systems. Formal modelling and analysis techniques offer the possibility of analysing design alternatives at an early design stage, and to a level of rigour not supported by conventional approaches.

In spite of the potential benefits, formal methods are rarely used in the development of business information systems. This is in part because they are associated with high-cost critical applications, and in part because they are perceived to present high barriers to adoption in terms of the training required and the modifications to existing processes and tools. Successful developers therefore have little incentive to adopt them.

The challenge we address is that of gaining the benefits of formal modelling and analysis in the development of SOA-based business information systems, while retaining a beneficial trade-off between the effort invested by developers insights that they gain. Our approach aims for a smooth transition between the traditional development processes

[0]Contact author: *John.Fitzgerald@newcastle.ac.uk*

and formal approaches. Initially, developers might not directly interact with a formal modelling tool, but continue to use pre-existing diagrammatic domain-specific modelling environments. The models developed in these environments could be automatically translated into a suitable formal notation and treated with automated analysis tools. While the insights gained from such purely automatic analysis might be less than those arising from a more thorough adoption, we expect that the formal methods would be seen by developers as a benefit demanding little additional effort. In the long run, we expect that subjectively experienced and objectively measurable benefits will lead to a positive attitude towards the methods and tools and then to their more extensive direct use.

This paper reports our experience investigating the technical feasibility of our approach. We aimed to establish whether formal methods could be used beneficially in the early design stages of SOA-based business information systems of the kind developed by SAP, and to show that the application of such modelling may be automated, so that developers could continue to use existing tools in the manner described above. In Sections 3 and 2 we consider the background and industrial setting of our studies. Section 4 introduces our chosen technology, and Section 5 describes the pilot studies undertaken. Section 6 discusses the lessons learned, and Section 7 gives directions for future work. This work was carried out as part of the EU project DEPLOY [5].

2 The Industrial Setting

In the business setting in which our work has taken place, developers construct applications supporting companies' business processes from components describing parts of processes, such as buying, selling, planning, site logistics and accounting. These components, based on an SOA, form a complex network using (mostly asynchronous) messaging to satisfy the components' communication needs without giving up their loose coupling.

In an ideal development process, the interior of the components is designed alongside their communication with other components. Developers decide on the inbound and outbound service interfaces and operations the components offer, as well as on the types and structure of the messages. The developers then also decide on how to configure the process integration layer or middleware (e.g., SAP NetWeaver Process Integration [14]) which is responsible for actually transferring the messages. This includes the choice of one of a set of reliability parameters as defined by the WS Reliable Messaging [12] standard. These include that messages should arrive "exactly once" (EO), or "exactly once in order" (EOIO). EOIO middleware will deliver all messages sent between two parties in the order they are sent without losing, corrupting or duplicating them and without inserting any new random messages, while EO middleware is the same except that it may reorder messages. A wrong choice can have serious consequences, since an EOIO configuration is a more expensive run-time option than EO. However, carelessly deciding on a weaker middleware and correcting the protocol in later stages of development will lead to high costs of revising design and implementation.

In current practice, the design steps sketched above are accompanied by modelling the systems with the help of domain specific diagrammatic languages. A strict validation process is in place to manually check the models for consistency. Then, the models are transformed into executable code and the code is tested.

A significant source of errors in distributed systems is poor communication media, which can, for example, delay, corrupt, reorder, lose or duplicate messages. Typical examples of such media are the internet and wireless intra-organisational networks. Our work develops an approach to systematic modelling of a family of middleware components ensuring a range of fault assumptions and to analysing the correctness of the business applications built by formal refinement using this middleware. We aim to allow, at an early stage of design, the selection the least expensive middleware with which the designed application will operate correctly.

3 Fault Tolerance and Formal Modelling

Ensuring high availability of modern business information applications depends, among other things, on a systematic use of the appropriate fault tolerance mechanisms. Fault tolerance [10] is a general means of attaining system dependability, applied when faults, and errors caused by them, cannot be avoided or eliminated during system design. Achieving dependability attributes, such as reliability, availability and safety, heavily relies on the use of the appropriate error detection and recovery mechanisms.

The choice of these fault tolerance mechanisms largely depends on the fault assumptions under which the system will function. Fault assumptions for distributed systems are typically classified into some of the following [4, 13]: omission, timing, value, state transition, impromptu and crash failures of the system components.

Applying formal modelling to specify and verify fault tolerance properties is becoming now an area of active research (see, for example [3]). This work often focuses on integrating fault tolerance means (for example exception handling, replication, error monitoring, reconfiguration) into application system models. Another area of active research is formal reasoning about correctness of distributed fault tolerance protocols (for example [9]). In the context of software architecture there is a substantial body of work on for-

mal specification and verification of fault tolerance connectors (for example [6]).

4. Event-B and the RODIN Platform

Event-B and the RODIN [15] tools platform have several features that make them appropriate as a basis for our study. The formal modelling language itself allows description both of structured data and behaviour. The available abstractions form a promising basis for describing information systems applications. The availability of a range of extensible tools, including proof support, model checking and animation is also important. Openness of the tool set is crucial for integration with existing development environments.

Event-B uses a model-oriented language in which data is modelled through a collection of built-in abstract data types from which more sophisticated types may be constructed. Data values may be constrained by logical predicates in the form of *invariants*. State variables modelling persistent data may be modified by *events* which describe functionality. Events are guarded by *conditions* that must hold in order for them to be enabled. The functionality performed by an event is described as an *action*.

A *machine* includes a set of invariants and a set of events. The logical conjecture that a machine is internally consistent (e.g. that events will not cause invariant properties to be violated) is given as a collection of *proof obligations*. Proof obligations may be discharged manually or, more likely, with the aid of an automated proof tool. *Contexts* contain *carrier sets* (which model abstract types) and *constants*, and are visible to machines.

Figure 1 gives a fragment from an Event-B machine. The syntax is slightly simplified. Three invariants and one event are shown. The first two invariants restrict the type of (previously declared) variables a and b. The event $change$ uses a parameter x. A proof tool quickly demonstrates that the event respects the first and third invariants, but the proof obligation arising from the second invariant cannot be proved. To make the machine consistent, either the second invariant or the event definition must be changed.

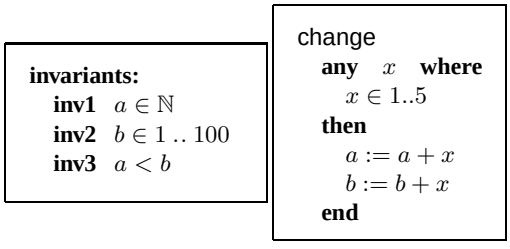

Figure 1. An Event-B fragment

A system model in Event-B typically consists of a chain of Event-B machines. Each machine (apart from the first) is linked to its predecessor by a *refinement relation*. *Linking invariants* relate the state of a machine with the state of its predecessor. Proof obligations ensure behaviour preservation between the linked machines. In a typical Event-B model, the initial machine is extremely simple, with detail being added in a controlled way step by step through a chain of refinements.

Using the RODIN tools platform, Event-B machines and refinement steps are constructed via a model editing interface. Proof obligations are automatically generated and discharged (so far as is possible) by proof tools built into the platform. In the event of an obligation not being automatically proved, an interface for manual proof guidance is used. The use of the Eclipse framework means that the tools may be extended with specialised provers, interpreters, model checkers, pretty printers and other new capabilities. In the work reported here, we focus mainly on the proof capability of the RODIN platform. Achieving a high level of automated proof is, however, important for our studies because we aim to give developers the benefits of proof-based analysis of models without the overhead of interacting directly with the Event-B formalism.

5. The Pilot Studies

As indicated in Section 1, the purpose of our study is to determine the technical feasibility of using formal modelling (in Event-B/RODIN) to support the analysis of design models of SOA-based business information applications. The specific focus is on selecting an appropriate configuration for middleware from among alternatives offering different levels of fault tolerance (EO or EOIO). The applications models should be derived automatically from existing graphical design tools and there should be a good level of automation in the analysis of the models.

Our approach is to use a series of case studies, with the aim of producing a proof-of-concept of the automated analysis discussed above. In our studies, we needed to:

1. Establish that formal proof tools such as Event-B/RODIN can indeed support the comparison of alternative middleware components with respect to application-level properties.

2. Define the process for interfacing alternative middleware models (e.g. EO or EOIO) to pre-existing application-level models.

3. Develop appropriate strategies for combining middleware models with application models derived from the pre-existing graphical design tools so as to yield a good degree of automation in the analysis.

These are the subjects of the studies described in Sections 5.1 to 5.3 respectively. The studies used two realistic but simplified SOA choreography examples. The first example is a business-to-business (B2B) choreography (or protocol) [16]. Two components, a buyer and seller, exchange messages in order to negotiate the price of a product or service. The negotiation is initiated by a proposal from the buyer detailing purchase conditions such as price, quantity, or delivery date. The two parties may then arbitrarily exchange further proposals. A party indicates agreement to a proposal by returning that proposal. The negotiation may be cancelled at any time. The critical property that B2B is designed to establish is:

Property 1 *When a run of the protocol terminates, either the buyer and seller should have agreed to the same price, or they should agree that the negotiation has been cancelled.*

The second example is an application-to-application (A2A) choreography in which two components interact to meet a requirement from a customer. The *ordering component* is responsible for managing customer requirements, and the *supply chain requirements component* coordinates the services used to process these requirements. The protocol starts when the supply chain component receives customer requirements from the ordering component. The supply chain component may then send notification of (partial) fulfilment of these requirements (e.g. delivery) back to the ordering component. The ordering component may also send queries and preliminary reservation requests and the supply chain component sends current supply planning and delivery information to the ordering component.

5.1. Study 1: Middleware Models

The aim of the initial study was to confirm that the Event-B/RODIN tools could support the comparison of alternative middleware components with respect to application-level properties. It used the B2B protocol. The application is built from a buyer and a seller component, and either EO or EOIO middleware. Our method was first to build Event-B models of EO and EOIO middleware, and an abstract model of the B2B protocol that did not contain an explicit component representing middleware. Each of the middleware models was composed in turn with the B2B model, and the Event-B/RODIN tools were used to compare the two combinations.

Each application-level event involves both a protocol party (buyer or seller) and the middleware. It is therefore partially described in each model. The protocol model describes the effects of the action local to the buyer or seller and the middleware model describes the effects of the action local to the middleware. Composing the middleware with the protocol involves composing each of these actions.

Buyer_send
any p **where**
$p \in PROPOSAL$
$p \notin \{empty, cancel\}$
$p \neq last\_s\_o\_rec$
$BAgreeStatus = NoAgreement$
$BCancelStatus = NotCancelled$
then
$curr\_b\_o := p$
end

Figure 2. Buyer_send in the protocol

For example, the application-level action of the buyer sending a proposal p to the seller appears in the protocol model as $Buyer\_send$ (see Figure 2). $PROPOSAL$ is a carrier set defined in a context visible to the protocol model. It contains all legitimate proposals; $empty$ and $cancel$ are designated elements of $PROPOSAL$. The state variable $last\_s\_o\_rec$ is the last seller offer received by the buyer. A separate event describes the case where $p = last\_s\_o\_rec$. *BAgreeStatus* and *BCancelStatus* record whether or not the protocol has been agreed or cancelled, from the point of view of the buyer. The current buyer offer is given by *curr_b_o*.

In EO middleware, in the case where the proposal is already in the middleware, the local effects of the buyer sending a proposal are defined as $Buyer\_send\_mw$, shown in Figure 3. The variable $mware\_to\_seller$ represents the middleware carrying messages to the seller, and has type $PROPOSAL \nrightarrow \mathbb{N}1$. A separate event describes the case where the proposal being sent is already in the middleware.

Buyer_send_mw
any p **where**
$p \notin dom(mware\_to\_seller)$
then
$mware\_to\_seller(p) := 1$
end

Figure 3. Buyer_send in middleware

The combined event in the application model retains the parameter p and conjoins guards and actions from each of the events above. Property 1 is added as an invariant to the combined model. This process is automated by a composition plugin available for the Event-B tool.

When the B2B protocol runs on EO middleware, Property 1 cannot be proved. A sequence of events which falsifies Property 1 is depicted in Figure 4, in which both the buyer and the seller accept old proposals as representing the

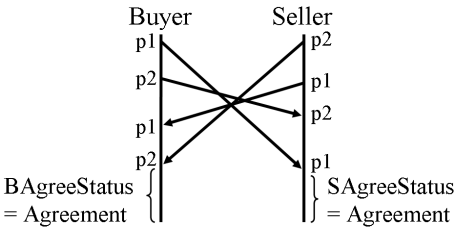

Figure 4. Falsifying Property 1

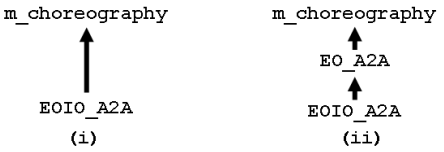

Figure 5. Two refinement techniques

current state of the other party. This sequence can be identified on the combined model using an Event-B animator (e.g. ProB [11] or AnimB [1]).

5.2. Study 2: Refinement-based modelling and analysis

In this study our aim was to further investigate how to develop models of business applications which allow for the introduction of middleware representations from a range of components and to develop standards for the integration of middleware into application models.

In this study, we integrated the middleware models with an independently developed model of the B2B protocol. This allowed us to identify more clearly the interface between the middleware and the protocol parties, and to develop a set of guidelines for protocols developers wishing to use the middleware.

The identified guidelines include:

- **protocol parties** should be developed in one machine, with no representation of middleware. Each send or receive event should instead use a reserved variable name as a parameter (e.g. "p" in Section 5.1).
- **correctness criteria** for the protocol should be expressed as application invariants, (although they will not in general be provable before a middleware model is integrated).
- **complex message sets** should be defined in a new context visible to both the protocol model and the middleware model.

5.3 Study 3: Investigating modelling options

Process components and middleware may be modelled in many different ways. The choice of level of abstraction and the particular representations of the data and events has an impact on the ease of comprehension and the ease of automated analysis. In this study, we sought to find suitable representations that would support automated analysis where the process component models are derived from the pre-existing modelling language and tools.

We used two different techniques to produce a new machine containing EOIO middleware, illustrated in Figure 5. The first technique produced the new machine by refining a machine in the original model containing the abstract choreography, and the second by refining a machine containing the low-level behaviour of the protocol.

An Event-B model of the A2A protocol was automatically generated from existing diagrammatic domain-specific modelling languages, rather than hand-crafted. It contained two machines, `m_choreography` and `EO_A2A`. The first machine, `m_choreography`, had a high-level view of behaviour and no explicit component representing the middleware. It contained seven events and two invariants, and produced no proof obligations. The second machine, `EO_A2A`, contained the local behaviour of the ordering and supplier components and a model of EO middleware. In `EO_A2A` each of the events from `m_choreography` was refined by a "send" and a "receive" event, giving 14 events. It had 22 invariants and 268 proof obligations of which 263 were proved automatically and 5 required (trivial) intervention.

To investigate the modelling options, two machines containing EOIO middleware were developed by hand. `EOIO_A2A_ONE` was a refinement of `m_choreography` and `EOIO_A2A_TWO` was a refinement of `EO_A2A`.

There were 257 proof obligations in `EOIO_A2A_ONE`, 162 of which were proved automatically. The remaining 95 were significantly more complex than the invariants in `EO_A2A`. The primary source of the increased complexity was ten invariants that relate messages in middleware to states within the machine. In `EO_A2A` the process components exchange messages from the set $MESSAGES$. The EO middleware does not offer an ordering guarantee, so the representation is an unordered bag *channel* of type $MESSAGES \rightarrow \mathbb{N}$. The quantity of a message M in the middleware is given as $channel(M)$.

In `EOIO_A2A_ONE` the middleware offers an ordering guarantee, so the middleware representation is a sequence of type

$$1 \mathinner{..} max\_chan\_len \rightarrowtail MESSAGES$$

and the quantity of a message M in middleware mw is given

by

$$card(dom((f \mathinner{..} l \lhd mw) \rhd \{M\}))$$

where f and l are the first and last unread messages in mw.

EOIO_A2A_TWO includes EO_A2A in the refinement chain. Linking invariants are defined between EOIO_A2A_TWO and EO_A2A. This gave rise to 25 invariants in EOIO_A2A_TWO. Seven of these are "linking invariants" linking the value of the middleware variables in EOIO_A2A_TWO to the value of the middleware variables in EO_A2A. These have the form

$$channel(M) = card(dom((f \mathinner{..} l \lhd mw) \rhd \{M\}))$$

EOIO_A2A_TWO contains 186 proof obligations, of which 116 were proved automatically. The remainder required about 6 hours of effort. The bulk of this effort went in proving the linking invariants between the two middleware representations.

The benefit of the first approach is that there is no need for the local machine EO_A2A. It represents the case where a machine containing a representation of EO middleware is not available, or the developer is interested exclusively in EOIO middleware. The (overwhelming) disadvantage is that the level of manual intervention required to prove the proof obligations is too high. Conversely, the second approach requires an intermediate machine (EO_A2A) to be built and proved. It represents the case where a machine containing a representation of EO middleware is available to be used.

The level of manual intervention required for the proofs at the second refinement stage is manageable, although we believe it could be further reduced significantly.

6. Lessons

The objective of our studies was to provide a proof of concept for our vision of verification tools supporting the selection of process components for business information applications built on service-oriented architectures. We have focussed on design decisions that affect the dependability of the application built from distributed components and middleware in a service-oriented architecture.

Our experience has shown that it is possible to use the Event-B language and tools to analyse design alternatives in the manner described. The approach and tools make it possible to assess the consequences of selecting different middleware configurations. The level of abstraction in the models allows this analysis to be done at an early stage in the development process. The formal tools underpinning the analysis can be integrated with existing non-formal design tools and augment the engineering judgement and experience that form the basis of such design decisions at present. The conjectures that can already be proved represent a significant step forward in tool support from the level of informal analysis offered by less formal design tools.

The verification technology that we have studied is intended for broad deployment in the sense that a large number of developers will use the tools without needing deep training in Event-B directly, although we expect that, in time, a proportion of developers would like to interact directly with the RODIN platform. This implies that the degree of automation in the verification process is important. Although a large proportion of proof obligations are discharged by the tools without user intervention, the overall proportion is rather lower than for some Event-B applications, suggesting that this level of automation can be raised substantially [2].

Our ability to discriminate between suitable and unsuitable middlewares is limited by the capabilities of the proof framework. In particular, failure to prove a property of middleware does not necessarily mean that the property does not hold. Thus a proportion of proof failures are "false alarms", as is inevitable in an expressive formal language.

It is important to select a good series of refinement steps to introduce a middleware model to a protocol, and considerable care is needed to choose abstractions which are effective in proof. Our objective is high automation in proof, but there is a risk that the models produced are less easily comprehended by the human reader. This is a challenge we expect to address in the future.

The protocols and middleware that we have examined so far are realistic, although relatively simple. As Study 3 showed, there are many modelling options and trade-offs that can affect ease of comprehension, the richness of fault assumptions considered and the degree of automation in proof.

The discovery of an invalid conjecture during verification may lead to either the selection of a middleware configuration offering stronger guarantees or the redesign of the process components to handle the identified faults. As a general observation, we think that the selection of stronger middleware needs to be traded off against the possible increase in complexity of protocol and component logic that results from the latter course of action. The advantage of our approach appears to be that this trade-off, which must be done at some point during system design, can be done explicitly and at an early design stage.

Although our results are preliminary at this stage, our observation is that this level of automated analysis represents a good trade-off of time for insight [7]. In the longer term, we expect that some enhanced support for failing proofs will prove valuable.

7. Future Work

Both the level of automation and the power of the tools to discriminate valid and invalid conjectures can be improved substantially. In Event-B, the refinement chain breaks the verification task down into steps that can be handled more readily by tools. In our case, when the middleware model is introduced in two refinement steps the proportion of proofs automatically discharged is substantially higher than when the middleware is introduced in a single step. A more sophisticated approach to the introduction of the middleware model could have a further significant effect on our automated proof completion rates. We expect further improvements in the level of automation from the ongoing developments of the RODIN provers, tactics and theories, encouraged by the openness of the platform.

Although we have focussed on an immediate industrial benefit, the breadth of possible applications suggests that there may be value in developing a library of middleware models (represented as patterns [8]) offering different fault assumptions corresponding to a range of media including wireless, internet and other communications mechanisms. A suitable structure for such a library may be a lattice, similar to the lattice of failure modes in [13].

Acknowledgements

This work has been supported by the EU project DEPLOY. We are grateful to many colleagues in the project for their contributions to this work, especially Thai Son Hoang, Vitaly Kozyura and Renato Silva.

References

[1] *AnimB: B model animator*. http://www.animb.org. accessed 2008-12-01.

[2] F. Badeau and A. Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 334–354. Springer, 2005.

[3] M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors. *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *Lecture Notes in Computer Science*. Springer, 2006. ISBN 978-3-540-48265-9.

[4] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[5] DEPLOY. http://www.deploy-project.eu/. accessed 2008-12-08.

[6] F. C. Filho, P. H. da S. Brito, and C. M. F. Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, 79(10):1397–1418, 2006.

[7] J. S. Fitzgerald and P. G. Larsen. Balancing Insight and Effort: the Industrial Uptake of Formal Methods. In C. B. Jones, Z. Liu, and J. Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, pages 237–254, Volume 4700, September 2007. Springer, Lecture Notes in Computer Science. ISBN 978-3-540-75220-2.

[8] A. Iliasov. Refinement patterns for rapid development of dependable systems. In N. Guelfi, H. Muccini, P. Pelliccione, and A. Romanovsky, editors, *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*. ACM, 2007.

[9] L. Lamport. Lower bounds for asynchronous consensus. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.

[10] P. Lee and T. Anderson. *Fault Tolerance Principles and Practice*. Springer-Verlag, Second edition, 1990.

[11] M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Software Tools for Technology Transfer*, 2008. to appear.

[12] OASIS. Web Services Reliable Messaging TC WS-Reliability 1.1. available at http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf, accessed 2008-12-05.

[13] D. Powell. Failure mode assumptions and assumption coverage. In *Procs. 22nd IEEE Intl. Symp. Fault-Tolerant Computing (FTCS-22)*, pages 386–395, June 1992.

[14] S. Raju and C. Wallacher. *B2B Integration Using SAP Netweaver PI*. SAP Press, 2008. ISBN 978-1-59229-163-2.

[15] RODIN. http://www.event-b.org/. accessed 2008-12-08.

[16] S. Wieczorek, A. Roth, A. Stefanescu, and A. 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)*. IEEE Computer Society, 2008.