2009 14th IEEE International Conference on Engineering of Complex Computer Systems

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

Jeremy Bryans, John Fitzgerald, Alexander Romanovsky
School of Computing Science
Newcastle University
Newcastle upon Tyne NE1 7RU, UK
Firstname.Lastname@ncl.ac.uk

Andreas Roth
SAP Research CEC Darmstadt
SAP AG, Bleichstr. 8,
64283 Darmstadt, Germany
Andreas.Roth@sap.com

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.

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

1. Introduction

Many business information applications are large-scale software systems that provide essential support to companies in their business processes. Designing such systems for dependability is therefore a demanding but important task. The software engineering challenge is to integrate many organisational parts and functions into one large and complex but consistent system.

The principles of service-oriented architecture (SOA) can help to master the complexity of business information applications. In SOA systems, such as those manufactured by SAP, complex applications are composed from independent business components that offer enterprise services. In a typical business application, several hundreds of service components may communicate with each other. Although the basic communication protocols are usually not individually complex, the large number of them means that the detection of problems, such as race conditions, requires considerable effort if performed manually. There is therefore an argument for investigating machine-assisted verification of these applications.

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, such as fault assumptions. Formal modelling and analysis techniques offer the possibility of analysing design alternatives at an early development 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 often 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. Our work as part of the European Deploy project [1] aims to achieve a long-term deployment of formal methods rather than a “one-off” demonstration of their capability. We therefore aim to lower some of these barriers to adoption by providing developers with a modelling and analysis capability that requires a relatively small change to current best practice.

In the light of these conditions, our work aims to answer three questions. First, can state-of-the-art formal modelling and analysis technology be used beneficially in the early design stages of dependable SOA-based business information systems of the kind developed by SAP? Second, what level of support is possible and can it be integrated smoothly with existing development practice? Third, what advances are needed in formal methods support to make this approach as effective as possible?

In this paper, we report on the results of initial studies in answer to our questions. We have investigated the early-stage application of formal methods to the problem of determining the reliability attributes required from SOA middleware to correctly design a business information application using a certain set of services. This analysis is supported by the Event-B formalism and Rodin tools and integrated with existing graphical design notations in SAP. In Section 2 we consider the industrial setting of our studies. Section 3 describes the formalism used, the modelling of faults and our approach to the achievement of successful deployment. We then describe the pilot studies undertaken (Section 4), the lessons learned (Section 5) and the directions of future work (Section 6).

2. The Industrial Setting

Our work concerns SOA-based business information systems of the kind developed using SAP technology. In this setting, developers construct applications that support companies' business processes, using 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 internals of the components are 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 [2]) which is responsible for actually transferring the messages.

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. Consequently, the configuration of the middleware is one of the most significant parts of the development process. This includes the choice of one of a set of reliability attributes as defined by the WS Reliable Messaging [3] 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.

The choice of middleware attributes for an application can have serious consequences on both maintenance and bandwidth costs. An EOIO configuration bears a higher risk of blocking queues, affecting maintenance costs. For example, consider an error occurring at the application level when processing input from an EOIO channel. Such errors most often have to be resolved by manual effort. While the repair is being effected, all other EOIO messages behind the blocked one remain blocked; an EO middleware might have allowed such messages to pass the blocking one, which may be judged as "better" behaviour. There are also consequences for bandwidth cost, since EOIO requires more effort at a lower protocol level for creating and closing message sequences. On the other hand, carelessly deciding on a weaker middleware that is eventually discovered not to provide strong enough guarantees for the application will lead to high costs in revising the design and implementation.

Given its significance, we focus on the configuration of middleware as an example of design decision that could be supported by formal modelling and analysis tools. Models of communicating business objects can be combined with models of different middlewares (Figure 1), allowing, at an early stage of design, the selection of the least expensive middleware that offers sufficient guarantees for the application to operate correctly.

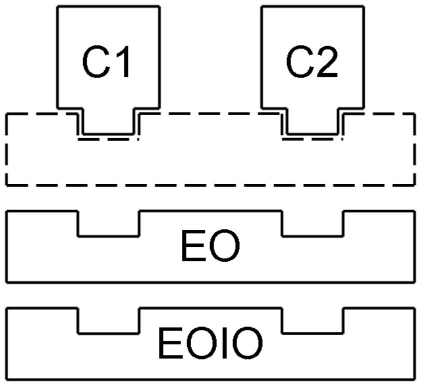

Figure 1. Choosing middlewares

3. Technology and Deployment Strategy

The goal of our work is to assess the feasibility of applying state-of-the-art formal modelling techniques in the development of dependable SOA-based business information applications in the industrial setting described above. We have taken a practical approach to this by developing and analysing formal models of real protocols that are implemented in business information applications, using a formal method and tools that have potential to be integrated with existing processes and tools. In order to do this, we first require a formal modelling and analysis framework that has robust tool support. Second, we need a means of expressing and analysing faults in middleware. Third, we require an approach to deployment of the formalism and tools into practice in the industrial setting outlined in Section 2. We discuss our response to each of these requirements below.

3.1. Event-B and the Rodin Platform

We have used the Event-B modelling formalism [4] and the Rodin [5] tools platform. Event-B and Rodin have several features that make them appropriate as a basis for our study. The formal modelling language allows description both of structured data and functionality. The available abstractions form a promising basis for describing information systems applications. The use of the Eclipse framework as a basis for the Rodin platform means that the tools may be extended with specialised provers, interpreters, model checkers, pretty printers and other new capabilities. The availability and extensibility of a range of such tools is important. The openness of the Rodin tools platform is also an important factor in 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*.

The basic unit of an Event-B model is a *machine*. Each machine may include invariants and 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. Definitions of *carrier sets* (which model abstract types) and *constants* may be defined in units called *contexts*, which are visible to machines.

Figure 2 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 because it would be violated if the event *change* were performed when $b > 95$. To make the machine consistent, either the second invariant or the event definition must be changed.

```
invariants:
  inv1   $a \in \mathbb{N}$
  inv2   $b \in 1 \,..\, 100$
  inv3   $a < b$
```

```
change
  any  $x$  where
    $x \in 1..5$
  then
    $a := a + x$
    $b := b + x$
  end
```

Figure 2. 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. In the work reported here, we focus mainly on the proof capability of the 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 formalism and proof tools.

3.2. Fault Modelling

Our second requirement is for a means of modelling and analysing faults in middleware. A *fault* is the cause of a system entering a state (termed an *error* state) that may lead to the system's deviating from specified behaviour, such deviant behaviour being termed a *failure* [6]. A failure of a component may in turn cause an error in the system of which it is a part, continuing the causal chain. A *fault assumption* is a specification of deviant behaviour that it is assumed may occur. For example, fault assumptions for distributed systems may include omission faults, faults in timing, value, state transition, impromptu and crash failures of the components [7], [8]. Stating such fault assumptions explicitly allows them to be brought into the design process so that appropriate fault tolerance [9] mechanisms can be selected and applied. Such mechanisms usually rely on the use of appropriate error detection and recovery techniques that bring the system back to a non-error state.

Formal specification and verification of fault tolerance properties is an area of active research (e.g. [10]), as is formal reasoning about correctness of distributed fault tolerance protocols (e.g. [11]) and formal specification and verification of fault tolerance connectors (e.g. [12]). Most of the approaches to modelling fault tolerance start with understanding and specifying fault assumptions, followed by defining appropriate fault tolerance mechanisms such as exception handling, replication, error monitoring or reconfiguration.

In our study fault tolerance is not always integrated into the application but rather is dealt with at the middleware level – the challenge is in selecting a sufficiently fault-tolerant middleware to guarantee key properties of the application working over that middleware. In an Event-B model of a business application over a SOA, fault assumptions might be recorded by including events that take the model into error states. For example, in an Event-B model of one of our business information systems, an assumption of omission failure in the middleware might be recorded by an additional "fault" event in the machine that is capable of "dropping" messages in transit. The verification challenge is in showing that middleware which has such events does or does not invalidate a system-level property expressed as an invariant.

3.3. Industrial Deployment

Our third requirement is for an approach to deployment of the formalism and tools into industrial practice. Our approach here is to use Event-B in a "lightweight" way [13] in the sense that full formalism is used but is applied to key aspects of a system and with a significant level of tool support. Recent applications of formal verification technology in static analysis and model checking, such as SLAM [14] and ESP [15], have been characterised by the integration of formal analysis with existing development frameworks. Our application differs from some of these in that our focus is on early design stages rather than code verification but we share the need to integrate verification analysis with existing design tools.

The scale and character of the SOA-based business information systems that we consider in this paper, with large numbers of service components engaged in protocols also suggests that the formal modelling and analysis technology must be readily used by a wide range of developers without requiring a sudden revolution in development processes and tools. Our approach therefore aims for a smooth transition from traditional development processes to the use of more formal techniques. 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.

4. 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 application 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 to answering the three questions posed in Section 1 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 4.1 to 4.3 respectively. The studies used two realistic but simplified SOA choreography examples. They represent a large class of business information protocols deployed in industry. 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.

4.1. 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, and the protocol and middleware models therefore contain a local description of each application event. The protocol model describes the effects of the event 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.

The protocol model contains, in effect, two independent state machines, one modelling the buyer and one modelling the seller. All variables are local to one or other of these machines, and all events read and influence the local variables of only one machine. The state variable $last\_s\_o\_rec$ is the last seller offer received by the buyer. *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*. The variables *last_b_o_rec*, *SAgreeStatus*, *SCancelStatus*, and *curr_s_o* are the corresponding local variables in the seller.

If the last offer received by the buyer ($last\_s\_o\_rec$) is the same as the current buyer offer, the buyer believes the parties are in agreement. This is also true, *mutatis mutandis*, for the seller. The important invariants in this model are therefore the ones given in Figure 3. These are local invariants, in the

```
invariants:
  inv1  $BAgreeStatus = Agreement \Rightarrow$
          $curr\_b\_o = last\_s\_o\_rec$
  inv2  $SAgreeStatus = Agreement \Rightarrow$
          $curr\_s\_o = last\_b\_o\_rec$
```

Figure 3. Buyer and Seller invariants

sense that *inv1* involves only variables local to the buyer, and *inv2* applies uses only variables local to the seller. For example, the application-level event of the buyer sending a proposal p to the seller appears in the protocol model as $Buyer\_send$ (see Figure 4). $PROPOSAL$ 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 $PROPOSAL$. This event describes the case where the buyer is sending an offer not in agreement with the last offer from the seller. A separate event, which describes the case where the buyer is in agreement with the seller, additionally sets *BAgreeStatus* to *Agreement*.

```
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 4. Buyer_send in the protocol

EO middleware

Figure 5 shows the EO model invariants. These describe the structure of the middleware. There are two variables $mware\_to\_seller$ and $mware\_to\_buyer$, representing the middleware carrying messages to the seller and buyer respectively. Each is a partial function representing two bags recording the number of proposals in the middleware, but not the order in which they were sent.

```
invariants:
  inv1  $mware\_to\_seller \in PROPOSAL \nrightarrow \mathbb{N}_1$
  inv2  $mware\_to\_buyer \in PROPOSAL \nrightarrow \mathbb{N}_1$
```

Figure 5. EO invariants

In the case where the proposal is not yet in the middleware, the local effects of the buyer sending a proposal are defined in the event $Buyer\_send\_mw$, shown in Figure 6.

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 6. Buyer_send in EO middleware

Combining two events into one retains shared parameters (in this case p), and conjoins guards and actions. In this way, each event in the protocol model is combined with the appropriate event from the middleware model to create an application-level event. This process may be automated by a composition plugin available for the Event-B tool [17].

Property 1 is formulated as two invariants in the combined protocol and EO model. One of these, which covers the case where the buyer and seller agree, is given in Figure 7. For the invariant to hold, there must be no messages in transit.

invariants:
inv20 $(BAgreeStatus = Agreement \wedge$
$SAgreeStatus = Agreement \wedge$
$mware\_to\_buyer = \varnothing \wedge$
$mware\_to\_seller = \varnothing)$
$\Rightarrow curr\_b\_o = curr\_s\_o$

Figure 7. Property 1 in EO

We are required to prove that every event upholds this invariant. This cannot be done when the B2B protocol runs on EO middleware. The proof obligations that fail come from the events associated with the buyer and seller receiving final proposals from the other party. At this stage, the value of the *AgreeStatus* variables changes to *Agreement*, without a guarantee that *curr_b_o* and *curr_s_o* are equal.

A sequence of events which falsifies Property 1 is depicted in Figure 8, in which both the buyer and the seller accept old proposals as representing the current state of the other party. This sequence was anticipated (it was known beforehand that the B2B protocol did not run on EO middleware) and can be identified on the combined model using an Event-B animator (e.g. ProB [18] or AnimB [19]).

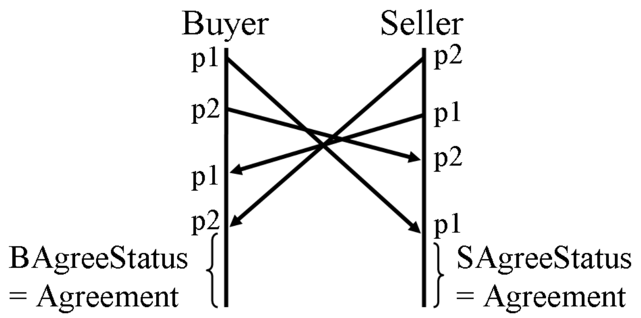

Figure 8. Falsifying Property 1

invariants:

inv1	$mware\_to\_seller \in$ $1 \mathrel{..} bufsize \pfun PROPOSAL$
inv2	$mware\_to\_buyer \in$ $1 \mathrel{..} bufsize \pfun PROPOSAL$
inv3	$b\_rpos \in \mathbb{N}$
inv4	$s\_rpos \in \mathbb{N}$
inv5	$b\_wpos \in \mathbb{N}$
inv6	$s\_wpos \in \mathbb{N}$
inv7	$b\_rpos \leq s\_wpos$
inv8	$s\_rpos \leq b\_wpos$
inv9	$dom(mware\_to\_seller) = 1 \mathrel{..} b\_wpos$
inv10	$dom(mware\_to\_buyer) = 1 \mathrel{..} s\_wpos$

Figure 9. EOIO invariants

EOIO middleware

Figure 9 shows the invariants in the EOIO middleware model. Invariants 1 and 2 fix the middleware structure as a partial function from a range of numeric labels (*1 .. bufsize*) to the carrier set $PROPOSAL$. The variable *bufsize* is defined in the context, and allows us to specify the largest number of messages a middleware can carry. This finiteness restriction allows the model to be animated. The read and write position variables defined in invariants 3 to 6 record the last positions read from and written to by the buyer and seller. Messages are written and read in order, and the read position of one component must not pass the write position of the other component (invariants 7 and 8). The domain of the partial functions is therefore $1 \mathrel{..} *\_wpos$, (where $*$ stands for b or s) and this is asserted by invariants 9 and 10.

The definition of the *buyer_send* event in EOIO is given in Figure 10. A new $PROPOSAL$ can only be added to the middleware if the maximum number of messages (*bufsize*) has not already been reached. When a new proposal is received, it is added to the middleware sequence and the write position is increased by one.

Property 1 is formulated as two invariants in the combined protocol and EOIO model. One of these is given in Figure 11. It differs from the EO formulation because the middleware structure is different. In the case of EOIO middleware, the middleware is empty when the value of each read position variable is equal to the value of the

```
Buyer_send_mw
  any  $p$  where
    $p \notin dom(mware\_to\_seller)$
    $b\_wpos < bufsize$
  then
    $mware\_to\_seller :=$
      $mware\_to\_seller \cup \{b\_wpos + 1 \mapsto p\}$
    $b\_wpos := b\_wpos + 1$
  end
```

Figure 10. Buyer_send in EOIO middleware

corresponding write position variable.

```
invariants:
  $(BAgreeStatus = Agreement \wedge$
  $SAgreeStatus = Agreement \wedge$
  $b\_rpos = s\_wpos \wedge s\_rpos = b\_wpos)$
  $\Rightarrow$
  $last\_b\_o\_rec = last\_s\_o\_rec$
```

Figure 11. Property 1 as an invariant under EOIO

All the proof obligations generated by Rodin for the combined model can now be discharged, guaranteeing that Property 1 holds.

4.2. 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.

The middleware models were integrated with an independently developed model of the B2B protocol. This allowed a clearer identification of the interface between the middleware and the protocol parties, and the development 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 4.1).
- **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.
- **carrier sets** should be instantiated to the messages the protocol parties exchange.

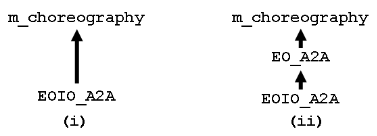

Figure 12. Two refinement techniques

4.3. 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 refinement techniques to produce a new machine containing EOIO middleware, illustrated in Figure 12. 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. This can be seen by considering the definitions of the quantity of a message M in each middleware. The EO middleware representation is an unordered bag named $channel$. In the EO middleware, the quantity of a message

M in the middleware is given as $channel(M)$.

In EOIO_A2A_ONE the middleware representation is a sequence of type

$$1 \mathinner{\ldotp\ldotp} bufsize \rightarrowtail\!\!\!\!\!\!+ MESSAGES$$

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

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

where f and l are the first and last unread messages in mw. While proofs containing the first definition were straightforward for the autoprovers, proofs containing the second definition required manual guidance.

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 link the value of the middleware variables in EOIO_A2A_TWO to the value of the middleware variables in EO_A2A, and all have the form

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

where $M \in MESSAGES$. EOIO_A2A_TWO contains 186 proof obligations, of which 116 were proved automatically. The majority of the invariants that required manual proof were 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.

5. Lessons

This paper has reported our first steps towards answering the three questions posed in Section 1. We have so far conducted three pilot studies in order to assess the feasibility of using verification tools to support the configuration of middleware components for dependable business information applications built on service-oriented architectures, and to understand better the work required to support this in an industrial setting like that of SAP. How have we fared against each of those three basic questions?

First, can state-of-the-art formal modelling technology be used beneficially in early design stages in this context? Our experience, even in Study 1, has shown that it is possible to use the Event-B language and tools to analyse design alternatives in the manner described, assessing the consequences of selecting different middleware configurations, augmenting the engineering judgement and experience that form the basis of such design decisions at present. The conjectures that can already be proved represent a step forward from the level of informal analysis offered by manual analysis and less formal design tools. The level of abstraction in the models allows this analysis to be done at an early stage in the development process.

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.

Our second question asked what level of formal support is possible and whether it is capable of smooth integration with existing practice. In Study 3, we were able to analyse formal Event-B models derived from designs expressed in an existing graphical notation, suggesting that this link could work well. There are some important challenges here. One is reducing the number of failing proofs. Another is communicating information about failing proofs to a user unfamiliar with the formalism. 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.

Our third question asked what advances are needed in formal methods support to make this approach as effective as possible. 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[1], suggesting that this level of automation can be raised substantially [20].

As we discovered in Study 2, it is important to select a

1. The manual proof obligations for EOIO_A2A_TWO in Study 3 took around 6 person-hours to discharge.

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.

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 [21]. In the longer term, we expect that some enhanced support for failing proofs will prove valuable.

6. 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. For example, one interesting possibility lies in using model-checking technology such as ProB [18] to identify a proportion of invalid conjectures before attempts at proof are made. Some of the potential of such a tools coupling has, for example, been explored with the SAL model-checker and PVS theorem prover [22].

We often wish to express reliability as a probability value (for example, the probability of failure on demand), or as the probability of a failure in a time period (as a probability distribution function.) Rodin does not support probabilistic primitives directly, although research on integrating such primitives into Event-B/Rodin is being carried out within the DEPLOY project.

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 [23]) 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 [8].

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] DEPLOY, “http://www.deploy-project.eu/,” accessed 2008-12-08.

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

[3] 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.

[4] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009, to appear. See also http://www.event-b.org.

[5] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin, “An open extensible tool environment for Event-B,” in *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods, ICFEM 2006*, ser. Lecture Notes in Computer Science, Z. Liu and J. He, Eds., vol. 4260. Springer, November 2006, pp. 588–605.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[7] F. Cristian, “Understanding fault-tolerant distributed systems,” *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, February 1991.

[8] D. Powell, “Failure mode assumptions and assumption coverage,” in *Procs. 22nd IEEE Intl. Symp. Fault-Tolerant Computing (FTCS-22)*, June 1992, pp. 386–395.

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

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

[11] L. Lamport, “Lower bounds for asynchronous consensus,” in *Future Directions in Distributed Computing*, ser. Lecture Notes in Computer Science, A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, Eds., vol. 2584. Springer, 2003, pp. 22–23.

[12] 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*, vol. 79, no. 10, pp. 1397–1418, 2006.

[13] D. Jackson and J. Wing, “Lightweight Formal Methods,” *IEEE Computer*, vol. 29, no. 4, pp. 22–23, April 1996.

[14] T. Ball and S. K. Rajamani, “The SLAM project: debugging system software via static analysis,” in *Proceedings of 29th ACM Symposium on Principles of Proramming Languages*, 2002, pp. 1–3.

[15] M. Das, S. Lerner, and M. Seigle, “ESP: Path-sensitive program verification in polynomial time,” *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 57–68, 2002.

[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.

[17] Event-B, “ http://wiki.event-b.org/,” accessed 2009-02-05.

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

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

[20] F. Badeau and A. Amelot, “Using B as a High Level Programming Language in an Industrial Project: Roissy VAL,” in *ZB 2005: Formal Specification and Development in Z and B*, ser. Lecture Notes in Computer Science, H. Treharne, S. King, M. Henson, and S. Schneider, Eds., vol. 3455. Springer, 2005, pp. 334–354.

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

[22] C. George and A. E. Haxthausen, “Specification, proof, and model checking of the Mondex electronic purse using RAISE,” *Formal Asp. Comput.*, vol. 20, no. 1, pp. 101–116, 2008.

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