Chapter 6
Business Information Sector

Sebastian Wieczorek, Vitaly Kozyura, Wei Wei, Andreas Roth, and Alin Stefanescu

Abstract Enterprise software helps modern corporates to automate their businesses in order to run efficiently and economically. We report a story of successful introduction of formal methods to business software development. This deployment came in three phases: modelling, formal verification and model-based testing. For each phase, we describe a few representative deployment cases in detail, and discuss the problems that we encountered and the decisions that we had to make. The work discussed here was carried out to focus on issues of interest to SAP, the world's leading provider of enterprise software.

6.1 Introduction to the Business Sector

Business application software has nowadays become indispensable to businesses because it provides the backbone and drives their activities through automation. For many areas, including manufacturing, supply chains, sales and human resources, data and services of various organisational units across the entire company need to be consistently integrated. With complex configuration options and business processes, it is no wonder that such software systems are very large and complex. Furthermore, business software constantly evolves and adapts to fast-changing business environments and requirements.

S. Wieczorek (✉) · V. Kozyura · W. Wei · A. Roth
SAP AG, Darmstadt, Germany
e-mail: sebastian.wieczorek@sap.com

V. Kozyura
e-mail: v.kozyura@sap.com

W. Wei
e-mail: wei01.wei@sap.com

A. Roth
e-mail: andreas.roth@sap.com

A. Stefanescu
University of Pitesti, Pitesti, Romania
e-mail: alin.stefanescu@upit.ro

A. Romanovsky, M. Thomas (eds.), *Industrial Deployment of System Engineering Methods*, DOI 10.1007/978-3-642-33170-1_6,

In response to these challenges, two important methodologies, namely *Service-Oriented Architecture* (SOA) and *Model-Based Development* (MBD), have been adopted for business software development in the last decade. The essence of SOA is to break the monolithic structure of large software up into smaller business components, which are then presented as services easily composed to meet increasingly more complex business needs. The development of service-based systems is layered as follows:

- **Development of functional units** that encapsulate a basic piece of computation.
- **Bundling of functional units into components** with the aim of providing reusable composable services.
- **Definition of business processes** by composing services to realise end-to-end business scenarios of enterprises.

MBD documents different aspects of software as models at every development stage, which enables early prototyping and detection of potential errors to avoid a drastically more expensive correction later. A software model retains only the details important to the corresponding design emphasis, and gets rid of any irrelevant information.

Correct functioning of business software is very important because failures could cause great financial losses. Formal methods have attracted growing attention for their capabilities to offer unambiguous semantics and to prove correctness rigorously. Thanks to the rich repository of models already available to us, it becomes less difficult and makes more sense to apply formal methods to the development of business software. It is less difficult because models are usually much smaller compared to the final implementation code, so they can be better handled with formal methods. It makes more sense because, on the one hand, models are used to guide subsequent development, and therefore their correctness is critical. On the other hand, models are perfect for deriving tests, since they capture the essence of how software is supposed to behave. The focus of our deployment of formal methods is on both formal verification and model-based testing.

One important objective during our deployment is to achieve a high degree of automation. Ideally, the application of formal methods should be completely hidden from designers and developers, because in our industry they should not be expected to understand or master formal methods. The high expense of training required is only part of the problem. Manual applications of formal methods, such as manual proofs, are usually very time- and resource-consuming, while the results are often not reusable due to frequent model changes. This would not only further increase the cost of software development, but also interrupt or even delay the development process to an extent that can no longer be tolerated. Therefore, we need to either make reasonable trade-offs (e.g., by sacrificing expressibility of modelling languages in favour of verifiability) or enhance the existing formal methods with improved automation. Finally, hiding formal modelling behind the surface of the existing MBD abstractions not only allows for seamless integration with the current development processes but also makes it easy to re-use the existing model contents.

6.2 Modelling

The early and wide adoption of MBD at SAP resulted in a vast collection of models that cover almost every aspect of software design, from high-level descriptions of business processes to low-level behavioural model for business objects. Some models are described using public standard modelling languages or their variants, and others using completely proprietary languages. We identified two main challenges that we needed to address at the modelling phase.

First, even though compared to implementation code models are small, their size and complexity could still be overwhelming for a formal analysis tool. To overcome this, we decided to compromise by leaving out certain modelling features that are deemed either inessential or too expensive to be analysed using formal methods. We also tried to break up monolithic model structures into smaller components/layers in order to reduce difficulties in verification, as seen in the cases of message choreographies (using layered design) and business processes (using model decomposition).

Second, most industrial modelling languages lack formal semantics. For example, Business Process Model and Notation (BPMN) has many elements with very vague and ambiguous interpretations in its official documentation. Even in case of models whose semantics is more or less clear, we should avoid applying formal analysis techniques directly to models represented in a variety of individual modelling languages. A much better, scalable practice would be to translate these languages into an intermediate language, and use the formal analysis methods on it.

Therefore, we decided to translate all models into a common formal language in which any future formal analysis would be performed. This has the additional advantage of making it possible to formally capture and verify the relations between different models. This is particularly desirable because in this way we are able to guarantee and maintain consistency across different design aspects. In the context of the DEPLOY project, we decided to use Event-B as the common formal language because of its powerful tool support by the Rodin platform.

6.2.1 Message Choreography Modelling

At the beginning of our work, we identified a missing layer in the modelling stack. While there are higher-level models for business processes and lower-level models for business objects, there are no models to describe message protocols for communication between business objects. There was static communication information, such as service interfaces and message formats, scattered throughout various documents and sources. However, there was virtually no documentation about the dynamic aspect of communication, i.e., message sequences that would occur at runtime. Therefore, we decided to come up with message choreography models (MCMs) that would provide all this information in a unified and consistent way. We started with a careful investigation of the state of the art in choreography modelling and matched it with the initial requirements gathered from developers. We

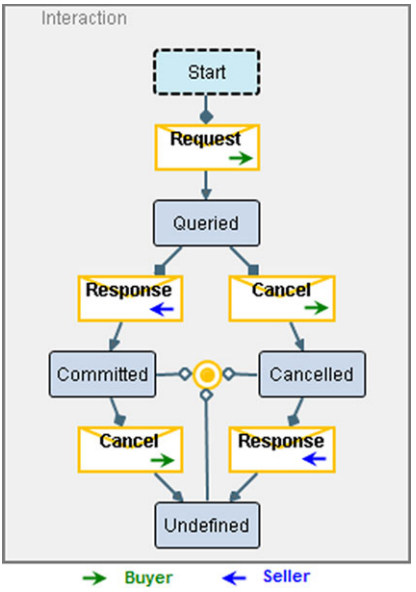

Fig. 6.1 A global choreography model

tried using several existing choreography languages, such as WS-CDL and BPMN-Choreography, to build simple models, and found that they were not suitable for our purposes. Finally, we adapted a proprietary language used internally to describe how business components call each other's services, and replaced business operation calls with message interactions between components. The result is the MCM language [9, 15], as shown in Figs. 6.1 and 6.2.

An MCM consists of a global choreography model that shows all possible sequences of messages exchanged at runtime, and a pair of local partner models that describe how each communication partner sends or receives messages with additional local constraints. In addition, for local partner models, we need to specify a property of the communication channel between partners: whether it is Exactly-Once (no message duplication, no message loss, but no message order guaranteed), Exactly-Once-In-Order (message order guaranteed), or something else. The global choreography model has no information about communication channels, and it is constructed from the viewpoint of an external observer. The decision in favour of the two-layer MCM structure was made to reduce verification complexity. Because the global choreography model is closer to user requirements, we can gain more confidence in their consistency through simulation and testing. Then we only need to verify that the local partner models are consistent with the global choreography models (more on this later) in order to assure that the MCM is consistent with user requirements.

The resulting concept soon gained developers' support, helped by the familiarity of its graphical representation. We built a prototype MCM editor based on SAP's internal platform, NetWeaver Development Studio, which includes a graphical modelling framework. We also used the editor as the basis for implementing various

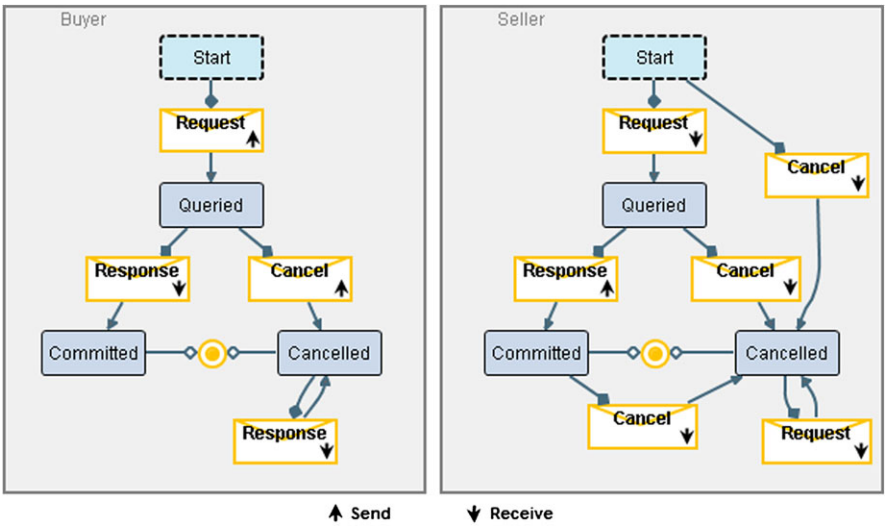

Fig. 6.2 Local partner models

formal verification and model-based testing techniques as plug-ins. Throughout the course of development, we continuously collected feedback from integration and testing architects in the field, thus involving potential users at each development stage. For instance, we extended the basic notion of state to allow for concurrency, which was an advanced feature needed in certain types of scenarios.

For evaluation purposes, we set up four pilots using real-life integration scenarios from the SAP platform, as described in [16]. The creation of each pilot model was conducted in *two guided sessions that lasted about one hour each*. In addition to that, we had *another two hours'* session of refinement and consolidation of the results. After the second session, semi-structured interviews were conducted with the pilot users. The general response was very positive. The participants perceived the possibility of formally describing the design as most beneficial, as it significantly improved communication between distributed development teams of interactive services. Furthermore, full integration of the existing modelling content (e.g., interface and component specifications) into the MCM was appreciated. The graphical modelling approach using a state-based representation was generally perceived as intuitive. The above case study showed that by using the MCM it is possible to model randomly chosen service communications that are part of a real SOA-based product using the MCM. The results suggest that the MCM is expressive enough to capture the relevant service communication.

As mentioned before, MCMs are first translated into Event-B for further formal analysis and test generation. The details of the translation can be found in [13]. Thanks to the state-based nature of MCMs, they are translated into Event-B in a quite straightforward fashion by using state variables and expressing transitions as state variable assignments in event actions. A global choreography model and its local partner models are translated into two separate Event-B machines. For any

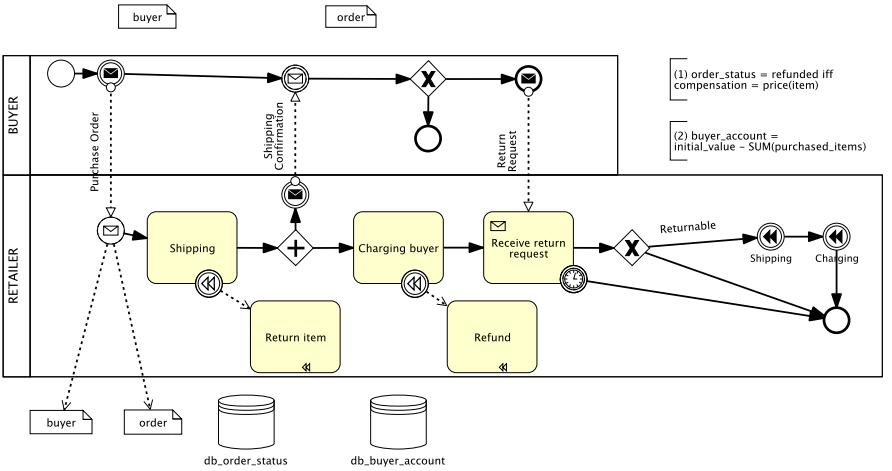

Fig. 6.3 A BPMN model of an online retailer

interaction between two partners in the machine representing the global choreography model, either the sending event or the receiving event of the respective message in the machine representing the local partner models is defined as the refining event of the interaction. These two possible refinement definitions are called *send-view* and *receive-view*. Message channels are modelled either as sets if they are EO channels (i.e. guaranteeing that the receiver gets each message Exactly Once), or as queues if they are EOIO channels (i.e. guaranteeing that the receiver gets each message Exactly Once In Order) The resulting Event-B model preserves the structure of the original MCM, so any verification results or generated test cases can be easily mapped back to their MCM representations.

6.2.2 Business Processes

Currently, business applications are usually built by integrating a broad range of highly configurable software components and services, which can be rapidly tailored to satisfy different and constantly changing business needs. Business process models are used to describe such integration scenarios and their workflows, facilitating an intuitive common understanding of the business logic between customers and developers. In addition to their use as documentation, business process models can also be simulated, analysed and verified to reveal design errors at an early stage of software development. BPMN has become the de facto standard business process modelling language, which is widely adopted in industry. A typical BPMN diagram is shown in Fig. 6.3: two collaboration partners (*BUYER* and *RETAILER*) and the flow of activities, events and messages.

BPMN is specified using natural and graphic languages, and comes without a rigorous semantics definition. Therefore, there are a lot of ambiguities in it that had

to be clarified as we designed the translation into Event-B [1]. Of course, these clarifications were made to meet SAP's specific needs. The translation works for most of the commonly used BPMN features, including comprehensive modelling of control flows, data modelling, compensation, message-based communication, error and exception handling, sub-processes, looping and multi-instance activities. The BPMN features not covered in the translation are most notably choreography and conversations as well as some types of flow objects, including call activities, transactions, conditional events and complex gateways. Some of these are rarely used in practice and would add significant complexity to the model. Others, such as transactions, are very vaguely described in the official BPMN specification and difficult to interpret.

Our translation was guided by three principles. First, the Event-B translation should be **structurally faithful** to the original BPMN model so that anyone with good knowledge of the original model can easily understand the translation. Also, any analysis result that we may obtain from the Event-B translation should be easy to map back to the original model. Second, the translation should be designed to improve **provability**, i.e., it should result in the automatic discharge of as many proof obligations as possible. Finally, we are interested in verifying properties for systems that allow **multiple instances** of the same processes.

We have tried two approaches to breaking down the complexity of the Event-B models that BPMN diagrams are translated into. In the first approach [1], we exclusively used the refinement relationship between machines to gradually add more and more information from a BPMN diagram. We start with a simple Event-B machine that contains only the control flow information of any collaboration partner, for instance, the *BUYER*. Then, we add a second machine that refines the first machine, and contains not only the control flow but also the data flow information of the *BUYER*. Subsequently, we gradually add the control flow and data flow information of the other partner in the machines. In the end, we use a final refining machine to add communication details. This approach has the following advantage: if we only want to verify a property related to the control flow of *BUYER*, then we can verify it in the first machine, which is much easier than verifying it in one that contains a lot of irrelevant information. However, since new information is always added to a refining machine without losing any old information, we still get an "all-in-one" machine, which becomes difficult to apply formal methods to.

Since our goal was separation of concerns, we experimented with another approach, taking advantage of the three model decomposition tools available in Event-B/Rodin [4]. Thus, we used modularisation, one of the decomposition tools mimicking the way that an object-oriented language uses interfaces/encapsulation/method-calls. The idea is to completely separate the local information of each collaboration partner. Each partner has an interface in which several publicly callable methods are exposed. The partner's details are, however, hidden in a series of refining machines invisible to other partners. In the end, we add a global machine to coordinate the interaction between partners. Such decomposition offers a clean separation of concerns. The detailed specification of each partner is replaceable and has no direct impact on the rest of the model, provided that its interface remains unchanged. Because of encapsulation, local behaviour can be completely verified within the

boundaries of the corresponding local partner, without information overload from other parts of the model. Verification of global properties can use local properties as intermediate lemmas. So, a proof procedure can be structured, and the degree of proof reuse is considerably higher.

6.3 Formal Verification

There are two kinds of properties that we focus on. The first kind is consistencies across different modelling layers, in particular, consistency between a global choreography model and its local partner models [5, 9, 15], as well as consistency between the MCM and the implementation models for business objects [7, 14]. The second kind is a selected set of invariants that a model must preserve during runtime, for example absence of deadlocks, absence of inconsumable messages in the MCM [6] and data consistency in business processes [1–3].

We applied both theorem-proving and model-checking approaches to the verification of the above properties. All domain-specific models were first translated into Event-B, and automated provers and the ProB plug-in of the Rodin platform were used to conduct verification. We could not achieve full automation of theorem proving even after several enhancements through various static analysis techniques and proof strategy optimisations. In contrast, model checking did not require much human intervention. However, we often ran into the state explosion problem, because checked models were usually too large for ProB to fully explore. In such cases, we had to manually reduce the explored state space by, for example, setting bounds on model variables. Nevertheless, we could still manage to obtain meaningful results by combining theorem proving and model checking. For instance, we usually applied model checking first with the hope of finding potential errors. We fixed the model accordingly and repeated model checking until no more bugs could be found. After that we started the more difficult and time-consuming proving procedure. Such a strategy can save a lot of time and is quite efficient for finding model errors.

In the following, we elaborate on how we apply theorem proving and improve automation with static analysis techniques.

6.3.1 MCM Verification

As depicted in Fig. 6.4, one major goal of our work was to enforce consistency between requirements and implementation. By introducing the MCM, we were able to divide this complex problem into manageable pieces:

- **Consistency Between Requirements and Choreography Models.** Requirements are not formalised in practice and hence applying formal methods at this level is impossible. Therefore, enforcing consistency between choreography models and requirements is a manual task. We found that the use of model simulation

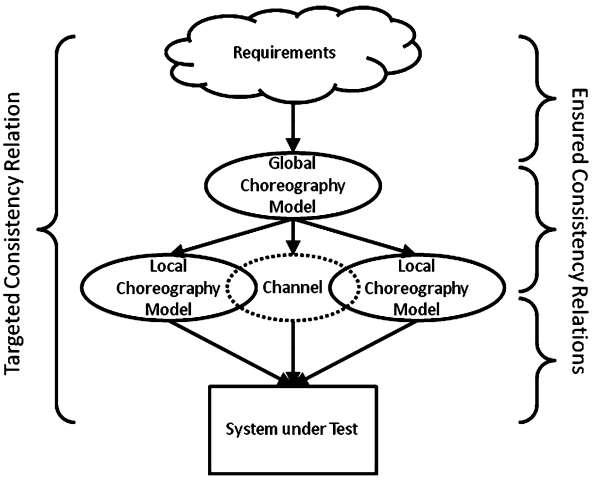

Fig. 6.4 Consistency relations in Choreography Modelling

based on ProB was usually sufficient to achieve high confidence that the choreography model captured what was informally described in the requirements.

- **Consistency Between Global and Local Viewpoints.** There are two possible solutions to enforcing consistency between global and local viewpoints: a generative approach, where the local views are generated from the global ones, and a checking approach, where global and local models are created separately and then verified for consistency with each other. In our work we implemented a mixed approach, which starts by generating local views from global ones but permits user modifications. The necessary consistency checks of manipulated views are realised by automatic transformation and verification, based on Event-B and Rodin.
- **Consistency Between Choreography Models and Implementation.** Service components are usually described with the help of implementation models, by specifying contained attributes (and their types) and state transition diagrams, and describing the effects of actions (such as service calls) on the internal states of components. We aimed to ensure consistency from choreography models to the implementation.

Figure 6.5 shows how we integrated Rodin into the MCM prototype for verification. The integration was made easier by the fact that both Rodin and our editor are Eclipse-based. A developer draws an MCM model using the editor. Within the editor, the translation of the MCM model into an Event-B model can be triggered. The resulting Event-B project is automatically loaded, and the consistency between the global choreography model and the local partner models is also formalised as machine refinements (through gluing invariants). The automated provers of Rodin then try to discharge all outstanding proof obligations (POs). A user can also interact with the provers to manually discharge POs. The ProB model checker can also be used to validate refinement relations.

We noted that in realistic scenarios MCM choreographies were not overly complex. Even the complicated real examples do not exceed 10–15 protocol states and

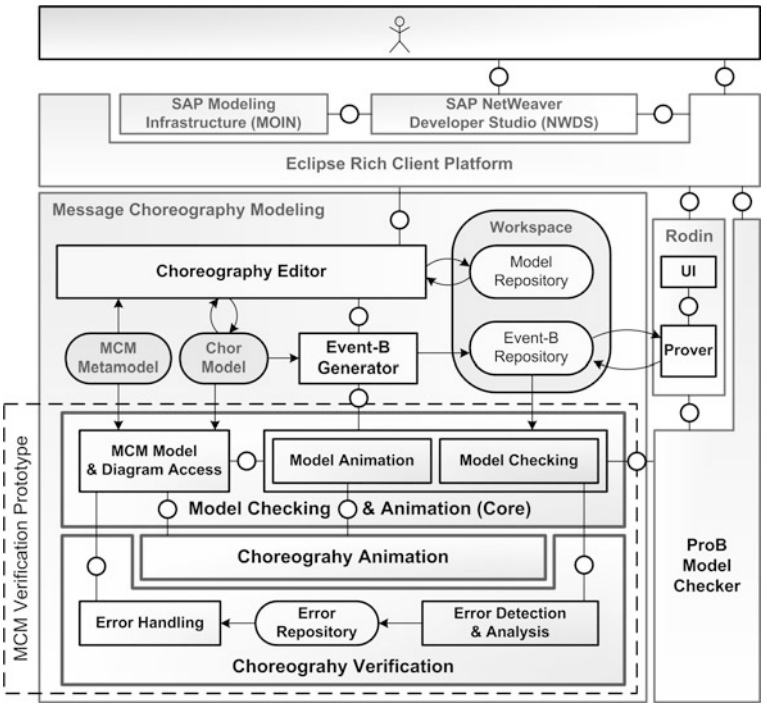

Fig. 6.5 Tool architecture for MCM and verification analysis based on Event-B

about 20 message events. We believe that this is a consequence of a good system architecture design that splits a complex system into manageable parts to be treated separately. However, even at this size of choreographies, subtle communication situations such as message racing and the large state space due to the datatypes of the exchanged messages occurring in a loosely coupled environment fully justify the use of automatic verification and validation techniques based on the MCM and Event-B.

Our experience showed that the automated provers were not able to automatically discharge a large set of proof obligations generated for gluing invariants. To improve automation, we enhanced our tool to automatically discover several kinds of invariants that describe certain dependencies between global states (those in the global choreography models) and local states (those in the local partner models) as well as dependencies between communication channel contents and MCM states [5]. The automated provers were then able to use these invariants as intermediate lemmas to discharge POs. Several experiments with realistic models show that about 300 to 600 POs were generated to verify consistency after invariant generation, and it was possible to automatically discharge up to 70 % of them. It was possible to discharge about 80 % of the remaining POs simply by manually switching to a specific prover (an interesting phenomenon related to how Rodin implements time-outs for

provers). It was possible to prove the rest of the POs manually without too much effort.

6.3.2 Several Remarks on Proof Automation

As illustrated in many cases, automatic discovery and generation of invariants is an important technique for reducing the difficulty of proving a property and increasing the number of automated proofs, a. However, which invariants should be generated depends not only on the types of models being verified, but also largely on specific characteristics of the individual models. For instance, for a business process model, it is important to have invariants specifying control flow dependencies and message flow dependencies [1]. For a business process involving data persistence, it is always helpful to discover additional dependencies between data flow and control flow. It is therefore hard to devise an automated algorithm to discover invariants for arbitrary models.

Another improvement to proof automation was achieved by making use of the Relevance Filter plug-in for the Rodin platform [8]. Using heuristics, the plug-in tries to pick most relevant and useful proof hypotheses from what is usually a very large pool of these. The use of this tool allowed a promising increase in the number of automatically discharged POs.

Inside SAP, we also developed a technique to better present the feedback from automated provers to designers, in case proofs fail to be derived [10]. The basic idea is to visualise the set of those states in the model that are associated with a certain proof step. The visualisation can be helpful in indicating and revealing potential errors in the design. We further enhanced this technique to allow a user to interact with the visualised state. For instance, the current state may ask the user to choose between two nondeterministic branches. By selecting one, the user is actually helping make a proof decision about which part of a disjunction should be focused on in the following proof.

We also discovered that the translation from a modelling language to Event-B can affect the level of proof automation. In proving consistency between the MCM and an implementation model [7], at the beginning we used logic formulas with quantifiers to define semantics for the implementation model. This proved to be ineffective since automated provers have always had great difficulties with quantifiers. Therefore, we decided to replace quantifiers with set operations, which automated provers can deal with better, using powerful simplification tools. After that we saw a large increase in the number of automated proofs.

No matter how much effort we have put into increasing the degree of proof automation, in almost every case there are some POs that require manual proofs. Even though the percentage of undischarged POs is quite low, their actual number is not small, and they are usually difficult to prove. This requires great knowledge and skills in theorem proving, which is not something we can expect from average developers and designers. Formal method experts could be hired to solve the manual

proof problem. However, it would still be a challenge to blend formal verification seamlessly and frictionlessly into software development in such a way that it supports other development activities without restricting or slowing down the whole process.

6.4 Model-Based Testing

It is hard to achieve full confidence in the correctness of software. Therefore, we focus on finding bugs with the help of software models in the last phase of deployment. Software models are very useful in guiding test designs, because they capture the essence of how software is supposed to behave. For instance, MCMs were used to automatically derive conceptual test cases, which can be easily mapped to actual test cases that run on the system under test [9]. According to the pilot users, the automatically generated test suites covered all tests that had been previously created manually. Another advantage of model-based testing is its complete automation. Test designers only need to create an initial test model, which is most of the time very intuitive, because we designed test models to be similar to the domain-specific ones that test designers are familiar with. Model-Based Testing (MBT) proved to be non-intrusive and very productive, replacing what is usually very tedious manual tasks of designing and creating test cases.

In this section, we illustrate how we apply model-based scenario testing to the existing business processes.

6.4.1 Scenario Testing of Business Process

As scenario testing is usually conducted on the user interface, we started working on MBT for graphical user interface (GUI) testing [11, 12, 14], which was a less studied research subject. Figure 6.6 depicts the envisioned testing approach that defined our deployment plan. Scenario testing is carried out when the whole system (or at least a major part of it) is developed and test-ready. The following describes the particular steps of this approach.

1. Based on SAP's expertise in industry's best practice, consultants, key customers and development architects derive business process models for a new product or feature or customer implementation so as to meet the market's or customers' requirements.
2. The created content, which effectively describes the usage scenarios of the new functionality, is used to generate test model skeletons. This step should be made automatic by using model transformation techniques.
3. The test models are then enhanced by test engineers in such a way that they reflect previously defined test goals and pin down the specifics of the concrete software architecture.

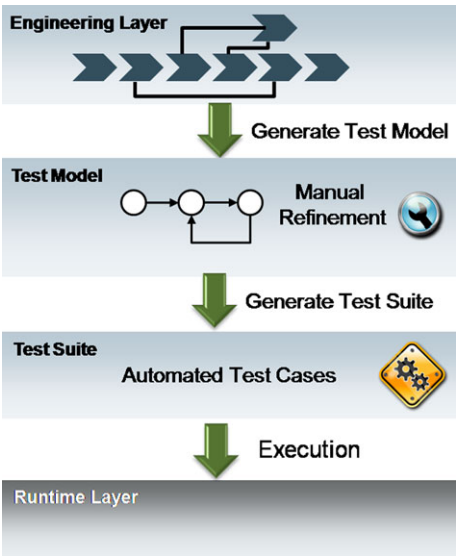

Fig. 6.6 Envisioned testing process

4. Abstract test suites are automatically derived from the test models, using MBT techniques.
5. The abstract test suites are optimised following best industrial practices (e.g., by minimising test case lengths while preserving test coverage). After further concretisations, the optimised suite is automatically executed on the user interface of the system under test.

In order to realise the envisioned deployment, we integrated various components into a testing framework productively used at SAP. Figure 6.7 presents the main blocks of this framework, including our components. The *Test Environment* offers UI-based keyword-driven testing capabilities through a *Scenario Editor*, which allows us to assemble captured test scripts and to visualise the generated executable scenarios (obtained from test cases). The scripts can be recorded through the *Script Recorder* component, which is connected to the System Under Test (SUT) for this purpose. Besides capturing user interactions on the SUT, the *Script Recorder* offers replay functionality, which is utilised for the stepwise execution of scenarios. Together with the *SUT* and the *Back-end Repository*, it assembles the original setup.

We extended the test environment by creating and integrating the *Test Model Editor*, which allows process-based test models to be created and edited. It further enables the triggering of the test generation and the visualisation of the resulting test suite.

In order to mitigate the risk of dependency on one single test generation technology, our goal was to integrate multiple tools and vendors, which we achieved by providing transformations from process-based test models (TMs) to abstract state transition machines (STMs); these can be further transformed into vendor-specific input formats in a straightforward manner. Since there is no standardised intermediate format, we created a proprietary STM. However, we published its concepts [12]

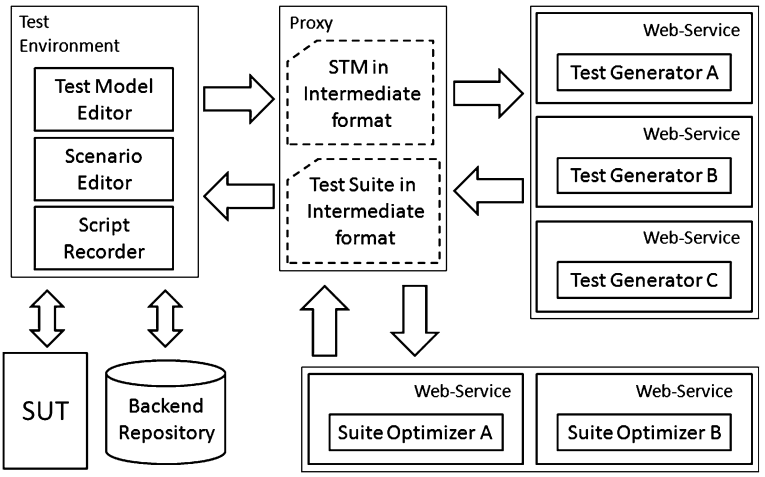

Fig. 6.7 Architecture of the MBT environment

and are active in contributing them to various emerging standardisation initiatives. A proxy has been set up for routing test generation requests in order to obtain a single communication partner, which allows us to add and update generator components without additional configuration of the test environment.

General-purpose MBT tools rely on various strategies to reduce the large initial test suites they produce during test generation. Therefore we decided to offer a unified test suite optimisation independent of the chosen test generator. This further allows us to consider custom requirements for the enterprise software domain. The different optimisation procedures are wrapped in another set of web services and can be used in the following way. After a test is successfully generated, the resulting traces are transformed into an intermediate test suite format and sent to the proxy component, which forwards them to an appropriate *Suite Optimizer*.

Test reduction is implemented on the intermediate format for test suites. Therefore, a further transformation of the results in the *Suite Optimizer* is not necessary. The proxy takes the reduced test suite and routes it back to the *Test Model Editor*, where it will be used to create a concrete test suite, containing executable scenarios.

Besides enabling the seamless integration of the *Test Model Editor*, detaching the test environment from the test generation services has the following advantages:

- **Reuse**: By using a generic input and output format, we are able to hide the complexity of specific model transformations in the input format of concrete test generators, thus making MBT accessible as a service to other potential test environments.
- **Performance**: Decoupling expensive computational functionality like test generation and test suite reduction promises better system performance and does not block front-end users. Replication of the Web services and introduction of load balancing to the proxy further increases scalability.

- **Maintenance**: The service-based decoupling in combination with a proxy also allows us to maintain and upgrade test generation components in a non-intrusive way.

6.4.2 Results

After prototyping, creation and integration of the components described in the previous section, we turned to one of SAP's major product areas in order to negotiate an evaluation strategy. It was agreed to set up a case study with seven development teams, which were asked to apply MBT in their scenario testing activities for a specific internal release. During these activities, the team members were asked to collect requirements and report bugs. At the end of the case study, further interview sessions were carried out with the participants, in order to get their overall assessment of the tool as well as information about their productivity and perceived learning effort. These results were consolidated and presented to the executive board of the product area, which consequently decided to start an unrestricted roll-out to internal development teams. As information about product and development activities needs to be handled with discretion (especially quality-related information), we will report the findings of the case study in a more general way.

Case Study Participants The participants did not have a background in formal methods but knew the basic concepts of business process modelling and were familiar with the concrete business process that they wanted to cover with different scenario tests. They were trained and experienced in the use of the proprietary testing environment, but did not have any knowledge of MBT. At the beginning of the case study, they received a two-hour tutorial on additional modelling and testing concepts necessary for understanding and operating our tool extension, as well as additional documentation and guiding samples. Furthermore, all participants could rely on remote expert support for any tooling, technology or process-related questions. On average, these support activities amounted to about one additional hour per participant.

Requirements Analysis Over the course of the case study, the participants collected 47 different requirements and ranked them based on their importance from 1 (an absolute showstopper) to 5 (nice to have). In the two-month period of evaluation we were able to incorporate all requirements ranked 1 and 2, and most of those ranked 3. The remaining requirements mainly concerned the automation of additional steps in the test generation process, which do not directly relate to test generation and had been manual in the original process, too (e.g., the linking of created test cases with test plans), or even addressed issues in the original test environment. Overall, only one requirement of the remaining concerned the enhancement of the test generation functionality, while the others mainly dealt with usability issues.

Interview Sessions Each participant was interviewed after the case study. All stated that the maturity of the tool improved dramatically during the evaluation phase, and agreed with the conclusion that both the tool and the new testing process were mature enough for wide use within the organisation. Furthermore, it was confirmed that the learning effort was small and the approach quite intuitive. The usability of the test model editor still left room for improvement, but was comparable to other internally used tools. It was noted by many participants that the MBT approach demanded greater care in the script recoding and test data definition activities. However, this was generally perceived as a positive side-effect.

Based on the requirements analysis and interview sessions, the executive board gained enough confidence to decide on a phased roll-out of model-based scenario testing to the whole product area. This roll-out will be accompanied by the handover of responsibility from our research unit to an operations team for the maintenance and further improvement of the tooling that was created in the context of DEPLOY.

6.5 Conclusion

Formal modelling and verification bring many quality assurance advantages to the development of business software. Design documents are complemented with software models that are accurate, executable, analysable, and can be used in deriving test cases directly linked to requirements. Formal validation and verification are not only used to prove correctness, but also to effectively find bugs, which is a nice alternative to traditional testing.

However, given the assumption that formal methods are hidden behind the existing domain-specific modelling abstractions, their success relies on the degree of automation. This is attested by the fact that model-based testing was more widely accepted by developers than formal verification, because MBT is fully automated. Users are not bothered by the technical details of test generation. They construct a test model that looks like a business process, and with a push of the button, test cases are generated and can be immediately run without any further effort. With formal verification, we are still in the process of increasing the degree of automation, to make tool usage and feedback more user-friendly, and to improve tool efficiency when dealing with large software models. Nevertheless, our pilot deployment of MCM verification is very promising and welcomed by software architects and designers. We will continue to work toward a seamless experience of using formal methods in business software development processes.

References

1. Bryans, J., Wei, W.: Formal analysis of BPMN models using Event-B. In: Kowalewski, S., Roveri, M. (eds.) FMICS. Lecture Notes in Computer Science, vol. 6371, pp. 33–49. Springer, Berlin (2010)

2. Bryans, J., Fitzgerald, J., Romanovsky, A., Roth, A.: Formal modelling and analysis of business information applications with fault tolerant middleware. In: ICECCS, pp. 68–77. IEEE Comput. Soc., Los Alamitos (2009)
3. Bryans, J., Fitzgerald, J., Romanovsky, A., Roth, A.: Patterns for modelling time and consistency in business information systems. In: Calinescu, R., Paige, R.F., Kwiatkowska, M.Z. (eds.) ICECCS, pp. 105–114. IEEE Comput. Soc., Los Alamitos (2010)
4. Hoang, T.-S., Iliasov, A., Silva, R., Wei, W.: A survey on Event-B decomposition. In: Automated Verification of Critical Systems AVOCS-2011. Electronic Communications of the EASST, vol. 46 (2012)
5. Kozyura, V., Roth, A.: Generation of gluing invariants for checking local enforceability of message choreographies. In: Jastram, M., Laibinis, L., Lösch, F., Mazzara, M. (eds.) Proceedings of DEPLOY Technical Workshop 2009. Newcastle University, Technical Report (2009)
6. Kozyura, V., Roth, A., Wei, W.: Local enforceability and inconsumable messages in choreography models. In: Proceedings of 4th South-East European Workshop on Formal Methods (SEEFM). IEEE Comput. Soc., Los Alamitos (2009)
7. Kozyura, V., Roth, A., Wieczorek, S., Wei, W.: Checking consistency between message choreographies and their implementation models. Electronic Communications of the EASST, vol. 35 (2010)
8. Röder, J.: Relevance filters for Event-B. Master's thesis, ETH, Zürich (2010)
9. Roth, A., Wieczorek, S., Kozyura, V., Wei, W., Wieczorek, S.: DEPLOY Deliverable D4.1: Report on pilot deployment in business information sector. Technical report, FP7-DEPLOY project EU (2010). http://www.deploy-project.eu/
10. Schur, M.: User interaction in formal verification of service choreography models. Master's thesis, Hochschule Karlsruhe Technik und Wirtschaft (2009)
11. Wieczorek, S., Stefanescu, A.: Improving testing of enterprise systems by model-based testing on graphical user interfaces. In: Sterritt, R., Eames, B., Sprinkle, J. (eds.) ECBS, pp. 352–357. IEEE Comput. Soc., Los Alamitos (2010)
12. Wieczorek, S., Kozyura, V., Schur, M., Roth, A.: Practical model-based testing of user scenarios. In: ICIT12, pp. 306–311. IEEE Comput. Soc., Los Alamitos (2012)
13. Wieczorek, S., Kozyura, V., Roth, A., Leuschel, M., Bendisposto, J., Plagge, D., Schieferdecker, I.: Applying model checking to generate model-based integration tests from choreography models. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TestCom/FATES. Lecture Notes in Computer Science, vol. 5826, pp. 179–194. Springer, Berlin (2009)
14. Wieczorek, S., Kozyura, V., Wei, W., Roth, A.: DEPLOY Deliverable D4.2: Report on enhanced deployment in business information sector. Technical report, FP7-DEPLOY project EU (2011). http://www.deploy-project.eu/
15. Wieczorek, S., Roth, A., Stefanescu, A., Kozyura, V., Charfi, A., Kraft, F.M., Schieferdecker, I.: Viewpoints for modeling choreographies in service-oriented architectures. In: WICSA/ECSA, pp. 11–20. IEEE Comput. Soc., Los Alamitos (2009)
16. Wieczorek, S., Stefanescu, A., Roth, A.: Model-driven service integration testing—a case study. In: QUATIC'10, pp. 292–297. IEEE Comput. Soc., Los Alamitos (2010)