Integration and Exchangeability of External Security-Critical Web Services in a Model-Driven Approach

Marian Borek(✉), Kurt Stenzel, Kuzman Katkalov, and Wolfgang Reif

Department of Software Engineering, University of Augsburg, Augsburg, Germany
{borek,stenzel,katkalov,reif}@informatik.uni-augsburg.de

Abstract. Model-driven approaches facilitate the development of applications by introducing domain-specific abstractions. Our model-driven approach called SecureMDD supports the domain of security-critical applications that use web services. Because many applications use external web services (i.e. services developed and provided by someone else), the integration of such web services is an important task of a model-driven approach. In this paper we present an approach to integrate and exchange external developed web services that use standard or non-standard cryptographic protocols, in security-critical applications. All necessary information is defined in an abstract way in the application model, which means that no manual changes of the generated code are necessary. We also show how security properties for the whole system including external web services can be defined and proved. For demonstration we use a web shop case study that integrates an external payment service.

1 Introduction

The use of external web services is essential for many applications. For example, a web shop needs to communicate with different external services, such as authentication and authorization services, different payment services or supply chain management services. Therefore, a model-driven approach for developing such applications needs to support the integration of external web services. One way to invoke external web services from a modeled application is to extend the generated code manually. However, our model-driven approach generates from a UML application model runnable code as well as a formal specification for verification of security properties for that application. The manual extension of the generated code would introduce a gap between the formal model and the runnable code, so that the verified properties do not hold necessarily for the running code. Another way is to integrate the external web service into the application model and generate everything from that model. Thereby, everything that is application-specific has to be modeled (e.g., message conversion or security mechanism) in order to be considered by formal verification. Furthermore, it is often necessary to be able to exchange those services against cheaper,

M.A. Jeusfeld and K. Karlapalem (Eds.): ER 2015 Workshops, LNCS 9382, pp. 63–73, 2015.
DOI: 10.1007/978-3-319-25747-1_7

more popular or more efficient ones. The challenge is to make the replacement of external services very easy and minimize verification effort.

Another benefit of the integration of external web services in a model-driven approach is the possibility to extend the approach by application-specific functionality without changing the transformations for code and formal specifications. With this approach also libraries and legacy systems can be integrated by being wrapped inside a web service.

This paper focuses on the integration and exchangeability of external web services in a model-driven approach for security-critical applications by considering different cryptographic mechanisms and discusses the verification of the entire application including the communication with external services.

This paper is structured as follows. Section 2 gives an overview of our model-driven approach and Sect. 3 describes the integration and exchangeability of web services. Section 4 considers assurances and the verification of security properties of the entire application and Sect. 5 explains how external services that use cryptography mechanisms can be integrated and exchanged. Section 6 discusses related work and Sect. 7 concludes this paper.

2 The SecureMDD Approach

SecureMDD is a model-driven approach to develop secure applications. From a UML application model using a predefined UML profile and a platform-independent and domain-specific language (MEL [6,15]), runnable code for different platforms as well as formal specifications are generated (see Fig. 1). One formal specification is used for interactive verification with KIV [4] (see [16,17]) and the other to find vulnerabilities with the model-checker platform AVANTSSAR [1] (see [7]). Additionally, platform-specific models are generated for incremental transformations and better documentation. The approach supports smart cards (implemented in Java Card [21]), user devices like secure

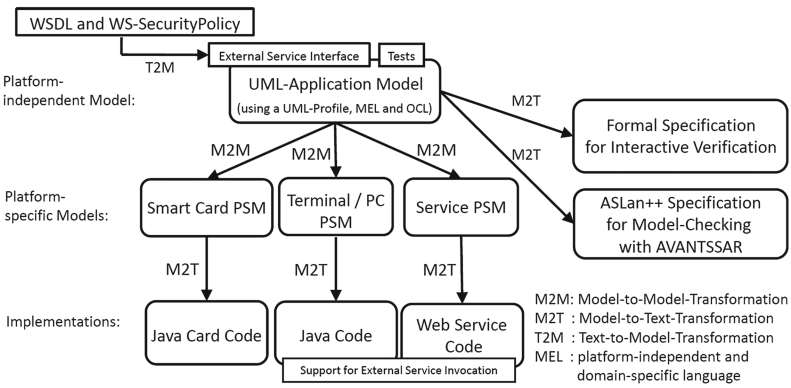

Fig. 1. SecureMDD approach

terminals or home PCs (implemented in Java), web services (also Java) and external web services. The static view of an application is modeled with UML class diagrams and deployment diagrams. The dynamic behavior of system components is modeled in UML activity diagrams with our platform-independent and domain-specific language MEL. Application-specific security properties are expressed with OCL in class diagrams (see [8]) and test cases that generate code for testing the generated application are modeled in UML sequence and activity diagrams (see [12]). The approach is fully tool-supported and all model transformations are implemented. For further information about our approach visit the SecureMDD website[1].

3 Modeling Communication with External Web Services

To communicate with a web service, the client which invokes the service needs to know its public interface. For SOAP web services this interface is defined by a WSDL document, which provides the offered web service functionality, and especially the expected messages in a machine-readable description. Our approach takes a WSDL document and transforms it automatically into an external service interface represented by a UML class diagram that is imported as a module into the UML application model (see Fig. 1). As a result, the external web service and all message data types are included in the application model as classes with operations, attributes and stereotypes. The model abstracts from information like the service address, namespaces and coding algorithm because they are not relevant for modeling an application in a platform-independent way and it is also not relevant for verification of the supported security properties. Because of this abstraction the resulting meta-model for external web services remains simple and it can be used for other web service specification languages like WADL. But the omitted information is available in the generated code as stubs that are generated automatically from the WSDL specification. We use WSDL2Java from Apache Axis2[2] with JiBX[3] as our stubs-generator to bind arbitrary class structures on XML documents. That is important because the data types from external web services differ from the predefined data types in our approach. The transformation from WSDL to UML is done by hyperModel[4] that uses generic XML schema documents as input.

The communication with external web services is mainly described in UML activity diagrams using our platform-independent and domain-specific language called MEL. The external web service is represented by a UML class with the stereotype ≪ExternalService≫ and the external web methods are represented by operations of that class and UML call behavior actions without modeling the behavior. The invocation of a web method is modeled by UML send signal actions and accept event actions that are connected with the UML call behavior actions.

[1] www.isse.de/securemdd.
[2] axis.apache.org/axis2.
[3] jibx.sourceforge.net.
[4] xmlmodeling.com/hypermodel/.

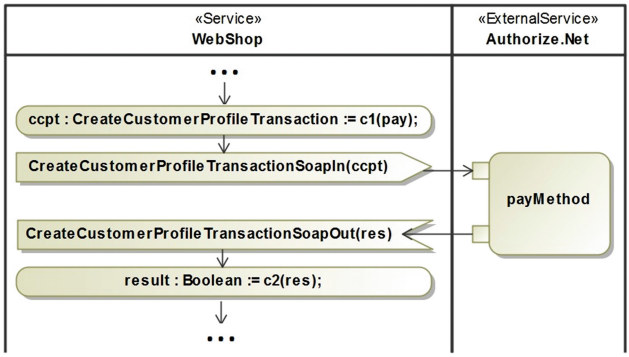

Fig. 2. Invocation of external web service

For conversion of the message data types between the modeled application and the external service, conversion methods have to be defined. This is done by sub activities using MEL. They have to be invoked before sending a message to an external web service and after receiving a message from the external web service.

Figure 2 shows how a modeled web service (*WebShop*) invokes the external service *Authorize.Net* for payment issues. Therefore the conversion methods *c1* and *c2* are used. *c1* converts the data from the modeled application into the required message structure of *Authorize.Net* and *c2* converts the result of *Authorize.Net* back. That means the payload *pay* from type *Pay* and *result* from type *Boolean* belong to the modeled application and *CreateCustomerProfileTransactionSoapIn* and *CreateCustomerProfileTransactionSoapOut* are data types used by *Authorize.Net*.

Figure 3 shows the definition of the conversion method *c1* that converts the *pay* object into *CreateCustomerProfileTransaction*. Some classes can be mapped

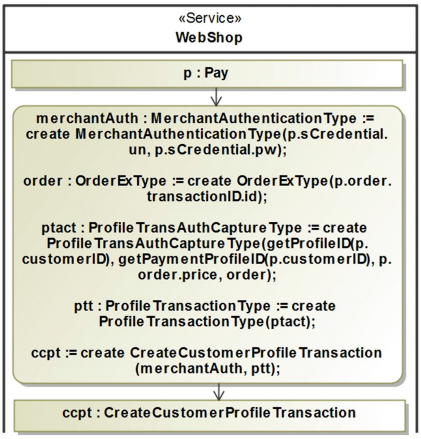

Fig. 3. Convert method *c1*

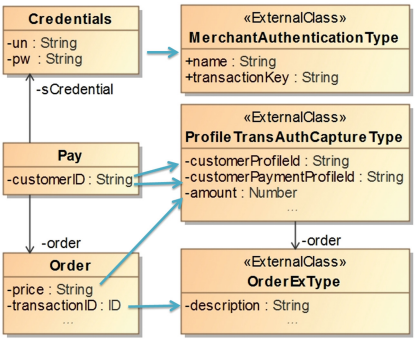

Fig. 4. Converted classes

one-to-one (e.g., *Credentials* and *MerchantAuthenticationType*) other classes consist of merged information from different classes (e.g., *ProfileTransAuthCaptureType* contains attributes from the *Pay* and *Order* classes) and if the output class has more attributes than the input class, the missing information has to emerge from the existing one by duplication or transformation (e.g., *ProfileTransAuthCaptureType* needs a *customerProfileId* and a *customerPaymentProfileId* that can be both extracted from *customerID*) (see Fig. 4). The conversion in Fig. 3 is chosen minimal but the output class *CreateCustomerProfileTransaction* has roughly 100 optional attributes and if all attributes are needed this leads to a very large and error-prone converting method. But because this method must be verified, mistakes that violate specified security properties will be found in contrast to a manually programmed conversion method.

Should the external payment service *Authorize.Net* in Fig. 2 be replaced with a different one, the protocol diagrams have to be changed and the verification of the entire application would need to be redone. To avoid this, the security-critical protocols and the invocation of an external service can be separated. In order to achieve that, we support a proxy pattern. Therefore, a proxy interface that is independent from the external payment service has to be modeled and used in the protocols described by activity diagrams. For each external service a proxy has to be modeled that implements this interface and invokes the external service. Then the external web services can be easily switched by changing the proxy in the class diagram. In our case study the *WebShop* has to invoke a *PayService* proxy interface. To add a concrete payment service like *Authorize.Net*, a new proxy (e.g., *Authorize.NetProxy*) has to be created that inherits from *PayService* and defines the behavior of the *pay* method and the conversion methods.

4 Security Properties and Assurances

For the verification of certain security properties of the entire application, assumptions about the external service are necessary. Those are assured by the service provider. If those assurances are informal, they have to be formalized by the developer. An example for a security property for our web shop is that "only goods that are paid for will be shipped". Obviously, some information about the external payment service method are necessary, e.g., that "if the return value is positive, then the payment was or will be successful". This assumption is specified for the proxy. It represents an abstraction of the external service and manages the conversion between different messages and the invocation of the external service. As a result, the security property is provable independently from the external web service.

Figure 5 shows that a security property uses classes from the client and the proxy and of course the assumptions specified for the proxy. The assurance of the external service uses classes from the external service interface that is generated from the WSDL and the assurance has to be a refinement of the assumption. The security property for the application, the assurance of the external service and the assumption about the proxy are formally defined as OCL constraints

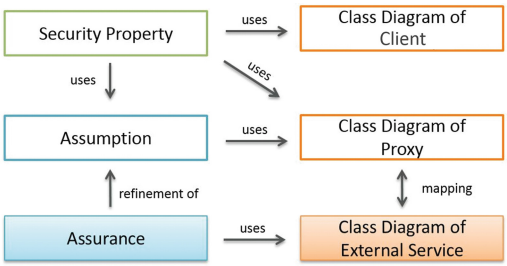

Fig. 5. Relation between security property, assumption and class diagrams

on classes that represent internal states and messages of the application participants. The mapping between messages are handled by the modeled conversion methods that are automatically transformed to executable Java code, and also to formal specifications. The relationship between the internal states is only necessary for verification. Hence, it is not modeled but specified during the verification. Those two mappings make it possible to show the refinement between the external service and the proxy. If the external service is a refinement of the proxy, the security property that holds for the payment method of the proxy holds also for the payment method of the external service. This way, the external services can be exchanged without influence on the security property if the assurances of the new external service are also an refinement of the assumptions of the proxy.

5 Cryptography and External Web Services

There are different ways to secure the communication by cryptography for web services. The simplest and most common way is to use TLS. It is a standard protocol that is independent from any specific web service. But TLS does not fulfill all possible requirements, e.g., end-to-end encryption. WS-SecurityPolicy is a language to describe individual cryptographic protocols for web services. But the design of application-specific security protocols is error-prone and requires verification. Additionally, it is likely that different web services have different WS-SecurityPolicies. This influences the exchangeability of web services. Our approach supports three different ways to secure the modeled functionality using cryptography.

1. The first one is to apply TLS on a connection between two system participants (e.g., web shop and an external payment service). This is modeled using a stereotype that is applied on a UML Communication Path between two UML Nodes in a deployment diagram. Furthermore, the stereotype has two properties to distinguish between mutual authentication and server side authentication. From this model runnable code that uses TLS to secure the communication as well as the key stores and default keys that have to be exchanged during deployment are generated automatically.

2. The second way uses predefined security data types for encryption, signatures, macs, hashes, nonces, keys and predefined operations to create those data types. Because in the past our focus was not exchangeability but ensuring application-specific security properties, there is no strict separation between application logic and cryptography. But for exchangeability this approach is unsuitable.
3. Therefore, the third way to secure the modeled functionality using cryptography in our approach is WS-SecurityPolicy. It applies cryptography directly before sending a message and directly after receiving a message but always independent from application logic. WS-SecurityPolicy is integrated in WSDL so the policies can be automatically extracted from the WSDL and transformed to an abstracted UML representation using stereotypes, classes and attributes. Additionally, it abstracts from WS-SecurityPolicy assertions like *AlgorithmSuite* because it is not used for the formal verification. A WS-SecurityPolicy specification of a web service can contain several alternative policies so the application designer has to choose one that should be used by the client. This is modeled with an attribute of the client or the proxy class. Because reusability makes software more clear, maintainable and reduces errors we mapped WS-SecurityPolicies to the already supported notation that is used for the generation of formal specifications. Therefore, MEL expressions whose behavior is equivalent to the policies are injected inside the modeled activity diagram that invokes the external service. This is done with model-to-model transformations in QVTo [19] and the resulting model is used with our existing generator for formal specifications.

   Figures 6 and 7 show a part of a protocol with injected policy behavior. In the original protocol, without the injected policy, the client collects the payment information (first activity node in Fig. 6) and sends it to the service proxy that invokes the pay method (last activity node in Fig. 7), which handles the conversion and invokes the external service. The regarded WS-SecurityPolicy describes a simple security protocol with symmetric binding and body encryption. The symmetric binding uses a X.509 certificate as protection token that is already exchanged and will be addressed in messages by its thumbprint reference. Hence, the injected part in Fig. 6 (second activity node) generates a symmetric key, stores it in the key store to be able to decrypt an optional response, encrypt the symmetric key with the public key from the X.509 certificate that belongs to the external service, creates the SOAP header including the encrypted symmetric key, encrypts the payment information with the symmetric key and puts it in a SOAP body object. The injected part in Fig. 7 (second activity node) decrypts the symmetric key from the header and uses the symmetric key to decrypt the payment information, that is used to invoke the pay method. The send and receive nodes are modified because the original modeled messages were exchanged with SOAP messages by the transformations. This is all done automatically together with the generation of the required classes. Besides the encrypted symmetric key the real SOAP header contains also algorithm information that can be omitted and token references like the thumbprint of the public key that is not necessary if the formal representation of the external service has only one key pair.

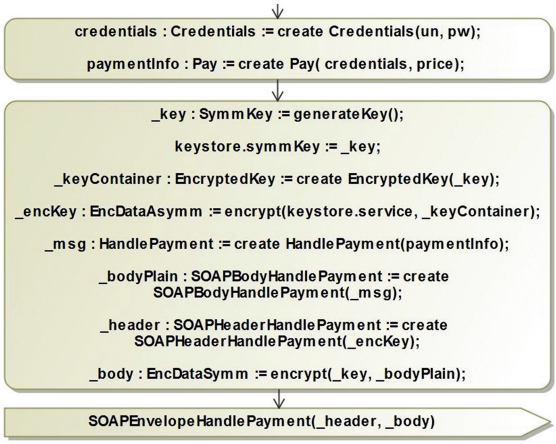

Fig. 6. Send HandlePayment with injected policy behavior

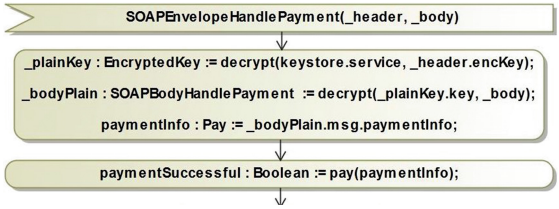

Fig. 7. Receive HandlePayment with injected policy behavior

Because the policy behavior is injected before the pay method (which handle the conversion and invoke the external service) is invoked, replacing external services that both use WS-SecurityPolicy can be done without additional verification if the policy of the replaced external service is a subset of the new one. In this case the new external service ensures the same security properties like the old external service plus some additional ones. This can be checked very fast and automatically during the transformation with QVTo.

6 Related Work

There are many works that consider web services in a model-driven approach. The most related works can be mainly categorized in static web service representation, orchestration and security.

Castro et al. [9] describe web services with a UML meta-model that is very close to the WSDL one. That means that each WSDL element is described by a stereotype with the same name. They also define transformation rules that generate a UML model from a WSDL specification. The advantage of this approach is that the WSDL specification and the generated UML model have the same

information content, but the disadvantage is that the resulted UML model has the same complexity like the WSDL specification. But the aim of a UML model is abstraction. [10,22] describe approaches that transform a WSDL specification in an abstract UML model by predefined rules. This is very close to our WSDL to UML transformation, but because our abstraction level differs from theirs we defined modified abstraction rules that fit better to our approach. They do not describe a model-driven approach that integrates external web services.

Self-Serv [3,5] is a model-driven approach for web service development with focus on orchestration. It uses state machines and generates BPEL based service-skeletons with orchestration logic but without modeling or generating application-specific logic. It does not consider security aspects and it does not integrate existing services. MDD4SOA [13] also illustrates a model-driven approach for web service orchestration. From a UML model code is generated for BPEL, WSDL, Java and the formal language Jolie. Security or the integration of external web services is not considered.

Nakamura et al. [18] enable the model-driven development of WS-SecurityPolicy specifications. They describe standard security properties with stereotypes that are used to select predefined security patterns from a library and apply them on the model. From that model, configuration documents for IBM WebSphere Application Server (WAS) and WS-SecurityPolicy specifications are generated. In contrast we generate an abstract model from the WS-SecurityPolicy of an existing web service that is selected and applied on the client as well as used for formal verification. Menzel et al. [14] also introduce a model-driven approach that uses abstract security patterns to generate XML-based configuration documents for the Apache Rampart-Modul that implements the WS-Security Stack. Jensen et al. [11] is also a model-driven approach that generates WS-BPEL, WSDL and WS-SecurityPolicy specifications from a model. The mentioned works do not integrate external web services and do not transform WS-SecurityPolicies into a formal representation for verification.

Pironti et al. [20] generate verified client-code, which uses an existing TLS-Service but they have to write thousand lines of code for the data conversion manually and without security guarantees. In [2] they explore the verification of systems with external services but they do not generate code, and verifying the security of the application is not part of that work.

We are not aware of a model-driven approach that considers the secure integration and replacement of existing web services in security-critical applications and verifies security properties about the whole application including the external services.

7 Conclusion

The integration and replacement of external security-critical web services in a model-driven approach is a novel and important topic. It enables the model-driven development of realistic applications that use existing code, e.g., services, legacy systems or libraries. In this paper we have shown how to model

the communication with external web services and how runnable code is generated automatically from the model without the necessity of manual changes. We have also discussed how security properties for the modeled application that uses external web services can be verified and how web services that use different cryptographic protocols are handled. An important issue was also the replacement of external web services with minimal verification effort. As a result, we were able to develop a simple web shop with our model-driven approach that integrates the real payment service *Authorize.Net*. Additionally, we are now able to extend our approach by application-specific functionality without changing the transformations for code and formal specifications. This can be done by providing the functionality as a web service and specifying the behavior with OCL. In our opinion this work extends model-driven development with verification significantly and makes the development of real applications that use external components feasible.

References

1. Armando, A., Arsac, W., Avanesov, T., Barletta, M., Calvi, A., Cappai, A., Carbone, R., Chevalier, Y., Compagna, L., Cuéllar, J., Erzse, G., Frau, S., Minea, M., Mödersheim, S., von Oheimb, D., Pellegrino, G., Ponta, S.E., Rocchetto, M., Rusinowitch, M., Torabi Dashti, M., Turuani, M., Viganò, L.: The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 267–282. Springer, Heidelberg (2012)
2. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Proceedings of the 32nd Symposium on Principles of Database Systems, pp. 163–174. ACM (2013)
3. Baïna, K., Benatallah, B., Casati, F., Toumani, F.: Model-driven web service development. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 290–306. Springer, Heidelberg (2004)
4. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, p. 363. Springer, Heidelberg (2000)
5. Benatallah, B., Sheng, Q.Z., Dumas, M.: The self-serv environment for web services composition. Internet Comput. IEEE **7**(1), 40–48 (2003)
6. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Model-driven development of secure service applications. In: 2012 35th Annual IEEE Software Engineering Workshop (SEW), pp. 62–71. IEEE (2012)
7. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Model checking of security-critical applications in a model-driven approach. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 76–90. Springer, Heidelberg (2013)
8. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Security requirements formalized with OCL in a model-driven approach. In: Model-Driven Requirements Engineering Workshop (MoDRE), pp. 65–73. IEEE (2013)
9. de Castro, V., Marcos, E., Vela, B.: Representing wsdl with extended uml. Revista Columbiana de Computation, vol. 5 (2004)

10. Gronmo, R., Skogan, D., Solheim, I., Oldevik, J.: Model-driven web services development. In: 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service, EEE 2004, pp. 42–45. IEEE (2004)
11. Jensen, M., Feja, S.: A security modeling approach for web-service-based business processes. In: 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, ECBS 2009, pp. 340–347. IEEE (2009)
12. Katkalov, K., Moebius, N., Stenzel, K., Borek, M., Reif, W.: Modeling test cases for security protocols with SecureMDD. Comput. Netw. **58**, 99–111 (2013)
13. Mayer, P.: MDD4SOA: model-driven development for service-oriented architectures. Ph.D. thesis, lmu (2010)
14. Menzel, M.: Model-driven security in service-oriented architectures. Ph.D. thesis, Potsdam University (2011). http://opus.kobv.de/ubp/volltexte/2012/5905/
15. Moebius, N., Stenzel, K., Reif, W.: Modeling security-critical applications with UML in the secureMDD approach. Int. J. Adv. Soft. **1**(1), 59–79 (2008)
16. Moebius, N., Stenzel, K., Reif, W.: Generating formal specifications for security-critical applications - a model-driven approach. In: ICSE 2009 Workshop: International Workshop on Software Engineering for Secure Systems (SESS 2009). IEEE/ACM Digital Libary (2009)
17. Moebius, N., Stenzel, K., Reif, W.: Formal verification of application-specific security properties in a model-driven approach. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 166–181. Springer, Heidelberg (2010)
18. Nakamura, Y., Tatsubori, M., Imamura, T., Ono, K.: Model-driven security based on a web services security architecture. In: IEEE International Conference on Services Computing, pp. 7–15. IEEE Press (2005)
19. Nolte, S.: QVT-Operational Mappings: Modellierung mit der Query Views Transformation. Springer, Heidelberg (2009)
20. Pironti, A., Pozza, D., Sisto, R.: Formally-based semi-automatic implementation of an open security protocol. J. Syst. Softw. **85**(4), 835–849 (2012)
21. Sun Microsystems Inc., Java Card 2.2 Specification (2002). http://java.sun.com/products/javacard/
22. Thöne, S., Depke, R., Engels, G.: Process-oriented, flexible composition of web services with UML. In: Olivé, À., Yoshikawa, M., Yu, E.S.K. (eds.) ER 2003. LNCS, vol. 2784, pp. 390–401. Springer, Heidelberg (2003)