Model-Based Development of Security-Critical Systems

Guido Oliver Wimmel

Institut für Informatik
der Technischen Universität München

Model-Based Development of Security-Critical Systems

Guido Oliver Wimmel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Peter Paul Spies
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
	2.	Univ.-Prof. David Basin, Ph.D. Eidgenössische Technische Hochschule Zürich/Schweiz

Die Dissertation wurde am 24.02.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.06.2005 angenommen.

Abstract

The subject of this thesis is the systematic development of secure systems. Security is a complex non-functional requirement affecting all parts of a system at all levels of detail and depending crucially on the assumptions about the system environment. Almost daily, new vulnerabilities are found, often exposing the concerned systems to attacks that result in severe damage.

In this thesis, we describe approaches for the integrated application of model-based development techniques with the aim to improve the security of the developed systems. Of central importance is the integration of security-related aspects into design models and the definition of security-specific development activities based on the extended models. For this purpose, we use a specification language with a formally defined semantics, to eliminate the potential of flaws because of ambiguities in the specifications. An important feature of our work is the provision of tool support, to reduce the necessary effort and experience required from the developer.

As part of our methodology, we present an approach to automatically generate a threat scenario from a model based on the included security-related information, and show how to use threat scenarios to formally verify security requirements by model checking.

Testing is necessary to gain confidence in the security of an implementation, but is not sufficiently integrated into current specification and verification methods for security-critical systems. We describe an approach for security testing based on the extended models. The described approach addresses both the problem of the generation of test sequences likely to discover potential vulnerabilities and the problem of the concretisation of the abstract representations of messages and in particular of cryptography that are employed in current approaches for the specification and verification of security-critical systems to make verification feasible.

We also address top-down oriented development of security-critical systems, where security aspects are first specified in the form of abstract assumptions, which are later realised by applying appropriate security mechanisms. In particular, we elaborate on security mechanisms whose insertion preserves the validity of the security requirements fulfilled with respect to the abstract assumptions. We apply these concepts to the modular specification and analysis of layered security protocols.

The practical applicability of the presented approaches is demonstrated in several case studies from the domain of electronic business and cryptographic protocols.

Acknowledgements

First of all, I thank Prof. Manfred Broy for giving me the opportunity to conduct research in his group and for his encouragement and support of this work. I also thank Prof. David Basin for his interest and for kindly agreeing to act as a second referee.

I thank Dr. Jan Jürjens for the continuing collaboration on a wide variety of topics related to security engineering and for loads of valuable discussions and suggestions.

Thanks also go to the colleagues who I had the pleasure of working with in various research projects during the last few years, Dr. Oscar Slotosch, Alexander Wißpeintner, Dr. Jan Jürjens, Gerhard Popp, Johannes Grünbauer, and Martin Deubler. Particularly, I thank Dr. Oscar Slotosch for his continuous support and further development of the AutoFocus/Quest tool. I am also grateful to Prof. Ruth Breu and Volkmar Lotz for discussions and feedback on early stages of this work. Further thanks go to Axel Heider for developing an implementation of the CEPS purse card.

I thank Dr. Heiko Lötzbeyer and Dr. Alexander Pretschner for introducing me to the topic of model-based test sequence generation.

Many thanks to Dr. Jan Jürjens, Thomas Kuhn, Johannes Grünbauer, Martin Leucker, and Andrew Dang, who took the time to proof-read parts of the thesis and provided valuable criticism on its presentation.

Finally, I am very grateful to my family for their continuing support and motivation, to my friends, and especially to Wu Ji for her enduring love and patience.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation and Problem Statement

Security has become an extremely important issue in software development. The continuing trend towards distributed and mobile systems pervading everyday life and communicating through increasingly interconnected networks considerably raises security risks by opening up many new possibilities for attacks. Recently, the first computer worm has been discovered affecting mobile phones and spreading via the Bluetooth interface [NRT04]. Standardised architectures and connections to open communication networks are to be introduced into automotive systems, where successful attacks may even have life-threatening consequences [Rud03]. The financial damage is already considerable: the current CSI/FBI computer crime and security survey [GLLR04] reports an average loss of $526,000 per respondent in 2003.

Developing secure systems is a difficult task. Security is a complex non-functional requirement affecting all parts of a system at all levels of detail. To secure a system, merely adding mechanisms such as cryptography in some places is not sufficient. Whether a system is secure depends crucially on the complex interplay of its components and on the assumptions about its environment. A single weakness can compromise the security of the entire system. Vulnerabilities are often subtle and hard to detect, as demonstrated by the design flaw in the Needham-Schroeder authentication protocol detected 17 years after the publication of the protocol specification [Low96], or by an implementation flaw allowing local root access, present in the OpenSSH application in various versions between June 2000 and March 2002 [Ope02], just to give two examples. In both cases, the fix merely consists of changing a few characters in the protocol specification, respectively in the C code.

Important measures applied in the development to achieve secure systems are:

Threat modelling and risk analysis The aim of threat modelling and risk analysis is the identification of threats, i.e. possible adversary actions to break a system's security, and the assessment of the risk the threats pose. Risk depends on the likelihood of occurrence of a threat and on the damage caused.

Elaboration of security requirements A set of global security requirements is elaborated forming a global security policy the system must conform to, in presence of the threats that were not ruled out in the risk analysis because the risk was perceived to be small enough to be acceptable.

Application of security mechanisms Security mechanisms applied to ensure that the security requirements are fulfilled include access control functionality, cryptographic protocols, or secure logging.

Formal methods Formal methods provide the possibility to model security requirements and security-related functionality in a mathematically precise way, and to carry out rigorous correctness proofs.

Security testing The purpose of security testing is to gain confidence that an implementation fulfils the stated security requirements and that the security-related functionality has been implemented correctly.

The Common Criteria [Com99], an international set of criteria for security evaluation, demand the use and the documentation of the use of such measures. The effort to be spent on the measures and their documentation depends on a selected evaluation assurance level (EAL).

Because security is an integral part of a computer system, security aspects should be taken into consideration throughout the whole development process, rather than regarded as an add-on during the later stages. A "penetrate and patch" strategy, i.e. designing a system with initially little attention to security aspects and later providing patches when vulnerabilities are discovered, has severe disadvantages [VM01]. For instance, only known vulnerabilities can be fixed, and patches are often not applied or even lead to new problems. Moreover, for certain kinds of systems, distributing and applying patches is inherently difficult or impossible — software stored on the read only memory of a smart card is an example.

Deficits of Current Approaches

Despite the necessity for an integrated consideration of security in software development, current development methodologies lack adequate support for security-specific development activities and products. Besides, even though applying formal methods considerably increases the assurance that a system fulfils its stated security requirements, formal methods are rarely used in practice because their application requires expert knowledge and is costly and time-consuming. There is a need for concrete guidelines for the application of formal models within a development process for security-critical systems and a need for formal modelling concepts for security-related aspects that are understandable for non-experts and can be applied with low additional effort, preferably integrated into a general-purpose specification language used in overall system development. In this context, automation and tool support are also an important requirement, to reduce the development time and to minimise the possibility of errors. Whereas tool support for single activities (most importantly, verification of security protocols) is available, integrated tool support for the diverse activities to be carried out within the development of security-critical systems, such as security verification, application of security mechanisms, and testing, is still an open area of research.

Security-critical systems should be constructed in a top-down oriented way, as suggested by the Common Criteria. A decision at a lower level of abstraction, such as the choice of a cryptographic protocol or an encryption algorithm, generally affects the security of the whole system and thus should be postponed until the security requirements have been broken down into sufficient detail such that it can be justified. Automated and sound techniques for the introduction of security mechanisms into a model based on abstract requirements would be desirable.

Confidence in the security of the implementation of a system can be gained by extensive testing. Testing for vulnerabilities is usually restricted to penetration testing, where a so-called "tiger team" of experts manually tries to break the system, or to the use of tools searching for known vulnerabilities. These approaches are not satisfactory, as they are not systematic and depend largely on the skill of the employed tiger team or on the knowledge encoded into the tool, which does not include application-specific vulnerabilities.

1.2. Aims and Approach

Aims

In this thesis, we argue that through the application of model-based development techniques in the development of security-critical systems one can improve the security of the developed systems and address the above described problems, while keeping the additional effort bearable. The central concept of model-based development are integrated models for the products (e.g., system specifications) and activities of the development process. Our aim is to provide an integrated modelling approach supporting the consideration of security aspects of distributed systems, and to examine how particular security-specific development activities can be carried out based on an integrated model with security-related information. We focus on design activities, in particular generation of threat scenarios describing a system under attack, security verification, testing, and application of security mechanisms, and on communication-related aspects. An important goal is to provide tool support within a general-purpose CASE tool, in order to reduce the necessary effort and experience for the security-related activities and to increase the level of confidence in the results. The applicability of the developed approach should be demonstrated at the example of various case studies of practical relevance.

Methodology and Tool Support

For this purpose, in this work we describe security-related artifacts and activities within a model-based development methodology for the design, verification, and testing of security-critical systems. In the centre of this methodology is a security-enriched model, where security requirements, threats, assumptions, and mechanisms to be applied have been included in the form of annotations, as a result of a security analysis. We explain our concepts based on AutoFocus/Quest [HMR+98], a general-purpose model-based development tool and specification language for the development of distributed and embedded systems. The use of a general-purpose specification language (as opposed to security-specific languages and methods, for instance for security policies or cryptographic protocols) makes it possible to treat security aspects and security-related functionality as an integrated part of the overall system to be developed and to build on existing tool support and experience on the part of the developers. In AutoFocus/Quest, a system is modelled as a network of communicating components, which are connected via channels and whose behaviour is specified by extended finite state machines. The main advantages of AutoFocus/Quest are its simplicity, its clear semantics, and its extensive connections to other tools, e.g. for formal verification using model checking, for model-based test sequence generation, or for code generation. Note that although we selected a particular specification language and tool for demonstration purposes, the general concepts described in this thesis do not depend on this choice. The main prerequisites of our approach are an executable, component-based description technique with an explicitly defined metamodel and a formal semantics, and the availability of verification support, for example by model checking.

AutoFocus/Quest includes a functional language to define and manipulate data stored by and passed between components, which we extended such that cryptographic expressions can be modelled, for example encryptions or hash computations.

Security-Specific Development Activities

In model-based development, security-specific development activities can be given in the form of model transformations. Figure 1.1 depicts an overview over the security-specific development activities addressed in this thesis. Of central importance in our approach is a transformation of a security-enriched model to a threat scenario by adding an intruder with appropriate capabilities. Both security verification and testing are performed based on the threat scenario, such that the malicious environment the system runs in is accounted for. Security verification and test sequence generation can be regarded as model transformations as well, adding truth values of security requirements (and possibly counterexamples) or test sequences to a model.

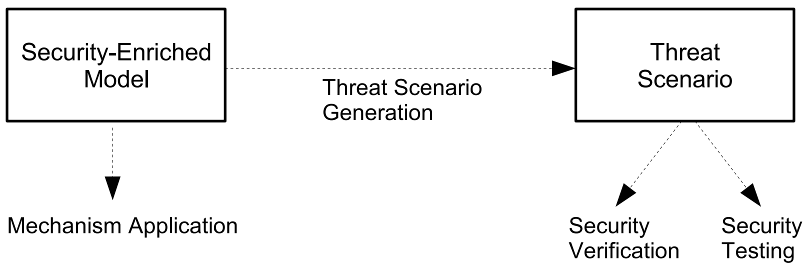


Figure 1.1.: Overview: Security-Specific Development Activities Addressed in this Work

Finally, we consider the application of security mechanisms, by transforming the security-enriched model. In particular, we address transformations that preserve the validity of the security requirements ("security patterns"), such that the security verification need not be repeated. An interesting class of security mechanisms are layered protocols, which are examined in more detail.

1.3. Main Contributions

We summarise the main contributions of this work.

Extended Description Techniques for Model-Based Development of Security-Critical Systems

We describe concepts for the extension of a model-based specification language for distributed systems with security-related aspects, demonstrated at the example of AutoFocus/Quest. The extension allows modelling of security-related aspects at different abstraction levels. In particular, we deal with the specification of security requirements, threats, assumptions, and security mechanisms, and with the use of cryptographic operations and cryptographic data such as keys or random challenges. The extension we provide allows the inclusion of all necessary information for the considered security-specific development activities. It is proven that the developed integration of cryptographic operations and cryptographic data into the used general functional language is sound, in the sense that the assumed abstract properties of the cryptographic operations are correctly reflected. This prevents the possibility to specify unrealistic behaviour. For example, we show that encryption cannot be broken on the abstract level of the functional language.

Threat Scenario Generation

We present an approach for the automated, tool-supported generation of a threat scenario from a security-enriched model, dependent on the annotated threats and assumptions. For this purpose, we provide a generic specification of the intruder behaviour on the semantical level (as a discrete system, i.e. a state transition graph), to abstract from idiosyncrasies of the specification language, and in addition give a concrete specification of the intruder behaviour in AutoFocus/Quest, specialised for model checking.

This two-step approach requires the justification that the generated specialised intruder model is sound and complete with respect to the generic specification. By soundness of the specialised intruder model, we mean that whenever an attack is found using the specialised intruder model, there is a corresponding attack with respect to the generic intruder model. Conversely, we say that a specialised intruder model is complete if all attacks are found using the specialised intruder model, i.e. whenever there is an attack with respect to the generic intruder model, there is a corresponding attack with respect to the specialised intruder model. We explain concepts to carry out corresponding proofs. In particular, we address the fact that the semantics of AutoFocus/Quest is based on synchronous composition of state machines driven by a global clock and that an intruder necessarily introduces delay into the communication. For this reason, completeness can only be achieved for specific classes of systems and properties, where the delay is not relevant. We give such a class, and describe the soundness and completeness proofs.

As the generated threat scenario is given in the same specification language as the original security-enriched model, all features of the used modelling tool can be applied to the threat scenario, such as simulation, verification, or test case generation. Besides, the threat scenario can be easily manually edited to analyse more complex threats than those specified by the security-enriched model.

Security Verification

We show how to verify security requirements based on the generated threat scenario, using the connection of the model checker SMV to the AutoFocus/Quest CASE tool. We demonstrate security verification at several examples, including an example where the threat scenario was manually edited, and provide performance figures for orientation. While verification using model checking could be successfully applied in the given examples, our focus on the integration of security verification into general model-based development with the help of existing verification support, rather than on efficiency considerations.

Security Testing

We adapt methods from classical model-based testing to the domain of security-critical systems.

The derivation of test sequences from a model of a security-critical system for the purpose of detecting possible vulnerabilities is difficult, because security requirements are in general universal properties (as opposed to properties requiring the *existence* of a certain behaviour). Universal properties cannot be used directly for the derivation of test sequences, because they are fulfilled by *all* traces, whereas for testing it is necessary to select a limited number of traces as test sequences. We give strategies to derive test sequences for security testing from a threat scenario by modifying the security requirements or the model.

To be able to execute the tests given by the generated test sequences, the abstract data in the test sequences must be concretised. We give an approach for the concretisation of abstract test data that is based on a test driver performing tests of an implementation with the help of an additional concretisation mapping. Our approach supports the use of cryptography, in particular the verification of results of cryptographic operations. For example, if the necessary key to verify an encrypted message is not available to the test driver, the verification is delayed. In addition, our approach deals with data which was modelled symbolically in the abstract model or was left out to reduce the model's complexity.

Mechanism Application

We show how the application of security mechanisms fits into a top-down oriented model-based process. The general idea is the replacement of abstract assumptions (e.g. confidentiality of a channel) by security mechanisms (e.g. encryption). A commonly used security mechanism is securing the communication between components by the application of a cryptographic protocol. We give a corresponding model transformation for the insertion of such layered protocols into communication channels.

Security Patterns

We define the concept of security patterns, as security mechanisms whose application preserves the validity of the security requirements in the transformed model. We show how to prove that a given security mechanism described by a model transformation is a security pattern. Moreover, we show how such proofs can be partly automated within the AutoFocus/Quest tool, using SAT solving.

This way, the verification task can be modularised into the verification of a model including abstract assumptions and the verification of security mechanisms to be applied to realise the assumptions. The verification of the security mechanisms only needs to be carried out once, and the complex transformed model where the security mechanism has been introduced need not be verified.

Verification of Layered Protocols

We apply the concept of security patterns to the introduction of layered protocols. We describe a number of patterns for modelling common uses of encryption and/or signature and state which abstract assumptions they realise. In a case study, we describe the modular verification of a layered security-critical transaction protocol where communication is tunnelled over a previously established SSL connection.

Tool Support

Tool support for all described development activities has been integrated into the AutoFocus/Quest tool set. In particular, we designed and implemented tool support for threat scenario generation, test sequence generation, test execution using a test driver, application of security mechanisms, and verification of security patterns.

Evaluation Using Case Studies

The presented approaches are evaluated in several case studies. The two main case studies dealt with in this thesis are an application for managing electronic order forms over the Internet (a real-life example taken from a joint research project in cooperation with a major German bank, see [HSG+03, GHJW03, Grü03]) and the Common Electronic Purse Specifications (CEPS) [CEP01], describing the functionality of a globally interoperable electronic purse scheme based on stored-value smart cards.

1.4. Comparison with Related Work

To our knowledge, to date no approach exists for the integrated, formally-based consideration of modelling, security verification, security mechanism application, and security testing within the development of security-critical systems. In the following, we give a short overview over the current state of research in this area and compare it to the results achieved in this thesis. A more detailed account of work related to the topics addressed in this thesis is given at the end of the respective chapters.

Development Processes for Security-Critical Systems

Current process models like the Rational Unified Process [JBR99] or Catalysis [DW99] treat security requirements as non-functional requirements among others and do not provide support for security-specific development activities. In [VWW02] (joint work with M. Vetterling and A. Wißpeintner), we show how to integrate security-specific development activities into a waterfall process to fulfil the requirements stated by the Common Criteria. A top-down oriented approach for the development of security-critical systems is presented in [Eck01]. Integrated modelling, tool support, automation and security testing are not addressed. While not considered in its predecessor, security-relevant activities as described in [Eck01] have also been included into the recently published revision of the German V-Modell [MIE$^+$05], but only at a coarse level of detail. [Pop05] describes a development process for security-critical systems using description techniques of the UML and additional models, to ensure access control properties. In contrast to our approach, [Pop05] is focused on early development phases, in particular on security requirements engineering. Tool support and automation are not part of that work. [Lot97] describes security-specific development activities in the design phase, based on models specified in the formal method Focus, but does not deal with explicit modelling of threats and assumptions and with testing issues.

Security Modelling and Verification

There are many approaches for modelling and verification of security-critical systems, especially focused on the verification of security protocols. For an overview, consider [GSG99, RSG$^+$00]. Usually dedicated specification techniques are employed, for example the languages CAPSL [Mil05], CASPER [Low98] or HLPSL [BMV03] for the specification of security protocols. In contrast, we deal with general security-critical systems, in which cryptographic functionality and security protocols can be utilised. Therefore, we use a general-purpose specification language to facilitate the integration of security verification into general systems development. In addition, in our approach the threat scenario that includes the intruder actions and that is the basis for security verification is a model in the same general-purpose specification language. This facilitates the consideration of complex threats by manually editing the threat scenario. In other approaches, general formal methods are used for security modelling and verification, such as the process algebra CSP [Low96], the guarded command language Murphi [Shm98], or the Focus method [Lot00]. By the description techniques used in our approach, we aim to improve the practical applicability of security modelling and verification and its accessibility to non-experts in formal methods. Besides, the mentioned approaches do not support the automatic generation of a threat scenario. Cryptographic messages are represented by standard constructs in the respective language (such as records). Unlike in our approach, it is left to the designer to ensure that these are manipulated in a sound way, for instance to extract a value from a record representing an encryption only if the respective party owns the correct key.

The SCR method and its tool support have been applied to security-critical systems development in [KAH99]. Treatment of cryptography, security-specific development activities and test case specifications are not addressed. [Jür02, Jür04] describes an extension UMLsec of the UML to include threats and security requirements into UML models, and defines a semantics for the relevant part of the UML and the security extensions. The concept of modelling security-critical systems is similar to our approach. The main difference is that threat scenarios are not represented as explicit UML models. Model checking support has recently become available for UMLsec [JS04], but currently only works for small systems. The author contributed to the work on UMLsec and a UML-based development process for security-critical systems in several areas [PJWB03, BBH$^+$03, JPW03, JLW05]. [LBD02, BDL03] describe an extension of the UML to specify (role based) access control requirements and how they can be transformed to access control mechanisms on the deployment platform.

Model-Based Security Testing

There has been extensive research into model-based testing; [Pet00] contains an annotated bibliography to a wide range of work. To the best of our knowledge, our work on model-based security testing (starting with [JW01c, JW01a, WJ02], joint papers with J. Jürjens) is the first published work using formally generated test sequences for security-critical systems based on the security requirements and formally treating the concretisation of symbolically modelled cryptographic operations and data. [CB03] describes a model-based approach for security functional testing based on models in the specification language SCR using general coverage criteria. The AVA approach [VM01] is designed to allow the assessment of the susceptibility of code against introduced perturbations of the data state, with the help of statistical testing. Testing criteria for the systematic generation of test sequences based on security requirements, treatment of cryptography, and concretisation of test sequences are not considered in these approaches. The PROTOS approach [Kak01] is aimed at testing low-level vulnerabilities such as buffer overflows, by introducing errors into a protocol specification. Again, testing criteria based on the security requirements and treatment of cryptography are not addressed.

The necessary concretisation of test sequences derived from a model to be able to execute the test on the implementation is often neglected in model-based testing approaches. Usually, application-specific scripts are employed for this purpose. In [DBG01], test concretisation is achieved by the definition of a mapping from variable assignments (as parts of a test sequence) to macros that are translated to concrete command sequences to be sent to and received from the implementation. We provide a generic concept for the explicit specification of the concretisation and a test driver that can deal with abstractly specified cryptographic messages.

Application of Security Mechanisms – Layered Protocols

General (informal) work on security patterns includes [YB97, FP01, Sch03]. Here, security patterns are regarded as well-understood solutions to security problems. Stepwise concretisation of secure communication channels is described in [Eck98]. Abstract properties of more complex mechanisms such as the use of the SSL protocol are only informally addressed. [Rud01] describes a top-down oriented design method for cryptographic protocols, which makes use of channels with abstract security properties. Unlike in our approach, the abstract channels map directly to cryptographic primitives. [AFG02] describes how to abstractly model secure channels in a formal calculus (join calculus) and gives a security-preserving translation of such models to a version of the calculus supporting cryptography. [Jür01] addresses secrecy-preserving refinement using a formal model and applies it to a secure channel implemented by a handshake protocol (TLS). [Jür04] contains an example where UMLsec is used for the modelling and analysis of an abstract channel preserving secrecy and of its realisation by encryption. In addition to this, we also deal with the integration of such security-preserving transformations into general systems development and with automation and tool support to verify that a transformation is security-preserving and to apply such a transformation.

[Bro04, HB05] describe a general formal model for layered architectures based on the Focus method.

1.5. Structure of the Thesis

We outline the structure of the remaining part of this thesis.

- **Chapter 2** presents background information on security and the main concepts of model-based development.
- In **Chapter 3**, we outline our model-based development methodology for security-critical systems and describe its artifacts and security-specific development activities. The methodology is divided into the analysis/design phases, including as activities security analysis, threat scenario generation, security verification, and mechanism application, and the implementation/testing phases with coding resp. code generation, and security test sequence generation.

- **Chapter 4** is devoted to concepts for modelling security-critical systems. We introduce the AutoFocus/Quest specification language for distributed systems. The abstract syntax of AutoFocus/Quest is given in the form of a meta model, and its semantics as a mapping of AutoFocus/Quest models to discrete systems (state transition graphs). We show how to extend such models such that security-specific aspects are supported and introduce the concept of a security-enriched model derived from an initial model of a system by incorporating security-specific aspects as a result of a security analysis. We demonstrate the described concepts at the example of two case studies.

- In **Chapter 5**, we show how to automatically generate a threat scenario from a security-enriched model, representing the behaviour of a system in presence of an intruder. We provide both a generic specification of the intruder behaviour in the form of a discrete system and a specialised intruder model in AutoFocus/Quest consisting of communicating finite state machines, suitable for verification of security requirements using model checking. We show that the specialised intruder model for model checking is sound (i.e., does not allow more attacks than the generic intruder model), and give a class of systems and properties for which it is complete (i.e., all attacks are found). The generated threat scenario can be edited to analyse more complex threats than can be specified in the security-enriched model. We demonstrate threat scenario generation and verification at the example of a number of case studies.

- **Chapter 6** is concerned with model-based security testing. We introduce the basic concepts of model-based testing, especially the automated generation of test sequences from test case specifications, and present strategies for the specification of test cases for vulnerability testing, based on a threat scenario generated from a security-enriched model. We describe an approach for testing an implemented system with test sequences generated from a threat scenario, using a test driver that performs translations between the abstract test data and the concrete data sent to and received from the implementation. In particular, we elaborate on the treatment of symbolically modelled data and cryptographic primitives, and of message parts that were omitted to make verification feasible. We apply and evaluate our approach at the example of a case study.

- In **Chapter 7**, we deal with the application of security mechanisms to a security-enriched model, as part of a top-down oriented development methodology. We focus on security mechanisms realising abstract assumptions on channels and give a model transformation for the insertion of a protocol layer providing a security service. We present proof concepts to verify that a model transformation preserves the validity of stated security requirements, and show how to partly automate such proofs with the help of SAT solving. We describe a number of patterns for modelling common uses of encryption and/or signature and state which abstract assumptions they realise. A comprehensive case study concerned with modular verification of a layered security-critical transaction protocol demonstrates the applicability of the described concepts.

- In **Chapter 8**, we conclude with a summary and discussion and give an outlook on future work.

- **Appendix A** contains an overview over frequently used notation.

2. Background: Security and Model-Based Development

This chapter presents some background information on the topics addressed in this thesis. In Section 2.1, we explain basic concepts of security. In Section 2.2, we provide a short overview over model-based development.

2.1. Security

According to the information assurance glossary of the American Committee on National Security Systems (CNSS), information systems security is the

> Protection of information systems against unauthorised access to or modification of information, whether in storage, processing or transit, and against the denial of service to unauthorised users, including those measures necessary to detect, document, and counter such threats. [oNSSC03]

Although to date there is no commonly agreed upon definition of security in information systems (see e.g. [GGK$^+$04, Poh04, Die04] for recent work in the German-speaking community on terminology of security and safety), the above definition highlights the basic characteristics relevant for the work presented in this thesis:

- Security is defined with respect to typical global security requirements, also referred to as protection goals, security services, or security objectives, such as the protection of certain pieces of information against unauthorised access.
- Security must be provided in presence of a malicious environment from which attempts can be performed to access or manipulate a system in an unauthorised way. In formal treatments, the malicious environment is usually modelled in the form of an intruder (also referred to as adversary, attacker or eavesdropper). In contrast to the common treatment in the domain of fault tolerance, the intruder is assumed to be "intelligent", i.e. the intruder can perform complex computations with the data received. For a security verification, the capabilities of the intruder must be stated (e.g. which parts of

the system the intruder can access). It must be justified that the security requirements are fulfilled for *any* possible behaviour of the intruder conforming to the assumptions.

- Properties that are required to hold for any possible behaviour of a system together with its environment are *universal properties.* Therefore, security requirements are universal properties.
- Global security requirements can refer to the intruder, e.g. by stating that the intruder should not be able to obtain certain confidential data.
- Typical security mechanisms are available to enforce global security requirements by detecting, documenting and countering threats. Examples for security mechanisms are cryptography or secure logging.

In the following, we describe the most common global security requirements (after [oNSSC03, Eck01]).

Confidentiality Confidentiality is the assurance that information is not disclosed to unauthorised entities.

Integrity Integrity is the protection against unauthorised modification or destruction of information.

Availability Availability means timely, reliable access to data and information services for authorised users.

Authenticity Authenticity is the validity and credibility of a transmission, message, or originator. Authentication is a mechanism to establish authenticity, e.g. the verification that a message supposedly sent by a particular subject was indeed sent by that subject.

Non-Repudiation Non-repudiation is the assurance that a subject is provided with proof that certain actions have been performed by other subjects, such that the performance of these actions cannot later be denied.

Fair Exchange Fair exchange is a requirement often stated for electronic commerce transactions, specifying that no party should be able to gain any advantage over the other party, even if the protocol stops for any reason. For example, if a buyer has made a payment, he must in return receive either the goods from the seller or a refund.

Initially, security requirements are non-functional requirements, i.e. quality attributes similar to performance, reliability or usability. In the course of system development, the security requirements are concretised to functional requirements specifying particular security mechanisms, such as the use of a cryptographic protocol.

In this thesis, we refer to information systems security simply as "security". For space reasons, a detailed introduction into security and security mechanisms has to be omitted. In particular, we assume familiarity with the basic concepts of cryptography. General literature on security with comprehensive treatments of security concepts and mechanisms includes [Bis03, VM01, Eck01, And01, Sch96].

2.2. Model-Based Development

The basic idea of model-based development techniques is to centre the development effort around building models of different aspects of a system, starting from early stages of development. Models provide abstract means for the specification of system aspects such as structure, communication, behaviour or data. The main advantages of model-based development are platform and language independence, the possibility to offer domain-specific modelling techniques (e.g. for enterprise systems or real-time systems) and a restricted degree of freedom in comparison to programming languages, which reduces the possibilities to introduce errors. Besides, building models at early stages of development contributes to identifying problems at times when they are still comparatively inexpensive to solve. A well-known example of a modelling language available in this context is the UML [OMG03], with the Rational Unified Process (RUP) [Kru00] as a process defined on top of its description techniques.

For the purpose of this work, we use a more specific notion of model-based development, along the lines of [SPHP02]. In [SPHP02], model-based development is defined as a "paradigm for system development that besides the use of domain-specific languages includes explicit and operational descriptions of the relevant entities that occur during development in terms of both product and process". The emphasis is on the fact that the models should be *explicit* and *operational*, and that explicit and operational models of both *product and process* are to be provided. Expliticitness means that models are represented in the form of mathematical entities and relations (respectively as data structures in a tool) based on meta models; operationality makes it possible to simulate models or to generate test cases.

In [SPHP02], models are classified according to a methodical point of view and according to a meta model architecture. From the methodical point of view, the following levels are differentiated:

Process level The entities and relations on the process level describe activities of the development process and their relations. The activities and their relations are defined on top of the elements of the conceptual model.

Conceptual level The entities and relations on the conceptual level describe the elements used to construct development products. Examples for conceptual model elements are components, communication channels or automata. They are visualised by using (possibly graphical) description techniques.

System level The system level (also referred to as semantical level) provides the means to describe the semantics of models at the conceptual level, in the form of a mathematical theory (e.g. predicate logic formulas defined over execution sequences). The semantics is then given by a mapping of model instances at the conceptual level to model instances at the system level.

A three-layer meta model architecture is defined as follows:

Model level Entities and relations on the model level correspond to models of a concrete product or development process. On the conceptual level, this could be a component Webserver.

Meta model level On the meta model level, the abstract syntax of models on the model level is defined, e.g. that a model consists among others of a number of components. Models on the model level are instances of models on the meta model level. Besides, on the meta model level views can be defined for particular subsets of a model, e.g. a structural view or a behavioural view. The representation of such models, e.g. within a CASE tool, is given by the concrete syntax. For instance, components can be depicted as rectangles.

Meta meta model level Finally, on the meta meta model level, means of description for models on the meta model level are defined. Models on the meta meta model level are in general domain-independent.

Here we keep to the terminology of the OMG meta model architecture: the above layers correspond to the MOF layers M1, M2 and M3 respectively. In [SPHP02], the corresponding levels are referred to as “instance level”, “domain level” and “meta level”.

In such a model-based development approach, development activities can be defined as *transformations* of instances of the conceptual model. An example is the introduction of a new component into a model of a component architecture. The development can be guided by *consistency conditions*. For instance, for a model of a component architecture to be executable (which is necessary e.g. to perform simulation runs), the behaviour of all components must have been specified.

The mapping of model instances at the conceptual level to semantical models at the system level enables the use of formal methods to reason about properties of a modelled system in a mathematically precise way. As motivated in the introduction, the use of formal methods is particularly desirable when dealing with security-critical systems because of the subtlety of potential flaws leading to violations of the security requirements.

Finally, *tool support* is of high significance in the context of model-based development, since appropriate tools can offer automation for a large number of model-based development activities.

3. Model-Based Development Methodology for Security-Critical Systems

In this chapter, we describe security-related artifacts and activities within a model-based development methodology for the development of security-critical systems. We focus on design and testing.

This chapter acts as an overview of the security-specific artifacts and activities addressed in more detail in the following chapters and explains their relationships.

3.1. Security-Specific Artifacts and Activities in a Model-Based Development Process

As described in Chapter 2, model-based development relies on explicit and operational descriptions of development artifacts and process activities. Activities can be specified as model transformations, where development artifacts are manipulated to produce new or modified development artifacts. If appropriate tool support is available, the development activities can be fully or partially automated. Model-based development allows an incremental and top-down oriented development strategy, where one starts with an abstract version of a system with limited functionality and adds detail or new functionality in a stepwise manner. Such a methodology provides the basis of our work.

Figure 3.1 depicts security-specific artifacts and activities to be included in a model-based development process to support the development of security-critical systems. Both artifacts and activities relevant in the analysis and design and artifacts relevant in the implementation and testing are considered. In Figure 3.1 it is indicated which of the activities can be carried out automatically (based on additional information included in the model), using the methodology and tool support developed in this thesis, and for which activities automation support can be provided as part of further work. The numbers annotated to the artifacts and activities refer to the chapters in this work where the respective artifact or activity is addressed in more detail. Numbers in brackets indicate that the corresponding artifact or activity is only briefly described. In the following explanations, the artifacts and activities appearing in Figure 3.1 are highlighted using boldface.

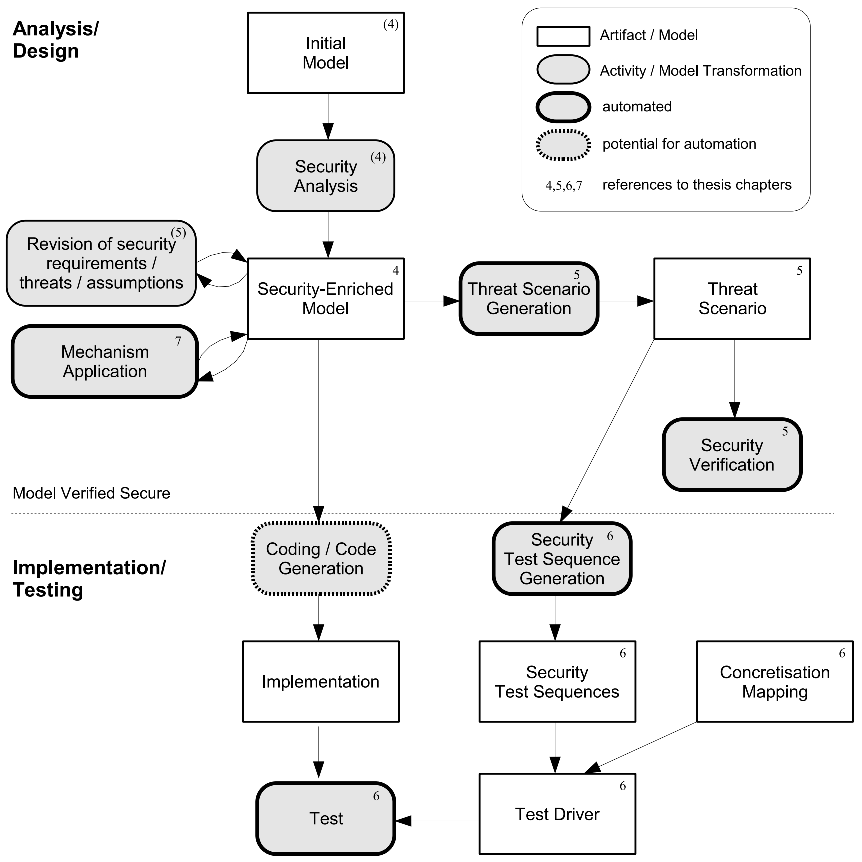


Figure 3.1.: Security-Specific Artifacts and Activities in a Model-Based Development Process

3.2. Analysis and Design

Initial Model The starting point of the methodology is an initial model, describing a coarse system architecture without security-related aspects. To be able to conduct a security analysis, at least enough information must be present to deal with assets and global threats, i.e. at an abstract level, subjects and objects in the system must be identifiable.

Security Analysis During a security analysis, a set of global security requirements is elaborated to form a global security policy the system must conform to. Threats are identified and a risk analysis is performed to assess threats with respect to their likelihood and potential damage. Assumptions are stated that can be taken for granted when determining if the security requirements are fulfilled.

Security-Enriched Model The results of the security analysis, i.e. the global security requirements, the relevant threats, and the assumptions, are integrated into the initial model. This leads to a security-enriched model that forms the basis for the security-specific development activities.

Threat Scenario Generation From a security-enriched model, a **threat scenario** can be generated. A threat scenario is a modification of a model by which an intruder is added who can perform attacks on the system. The capabilities of the intruder depend on the threats and assumptions included in the security-enriched model.

Security Verification Security verification is the verification that the global security requirements are fulfilled with respect to the threat scenario. If the result of the security verification is that a particular security requirement is violated, there is a corresponding attack on the system. Otherwise, the system is secure given the assumptions included in the model.

Revision of Security Requirements / Threats / Assumptions If a security requirement is violated, then the security-enriched model needs to be adjusted. One reason for the violation of a security requirement is that it was stronger than intended or incorrectly formalised. In this case, the security requirement must be revised. To ensure that a security requirement is fulfilled, one can also weaken or omit specified threats, or strengthen or add assumptions. Threats should only be weakened or omitted if indeed the attacks that are no longer considered are not regarded as relevant. On the other hand, newly added or strengthened assumptions can later be enforced by introducing appropriate security mechanisms.

A revision of the security requirements, threats and assumptions must also be carried out if new functionality was added to the security-enriched model as a part of an iterative development.

Mechanism Application Mechanism application is the introduction of security-related functionality into the security-enriched model to ensure that the security requirements are fulfilled and to realise abstract assumptions. An example is the introduction of encryption to realise the assumption that the data transferred via a channel is kept confidential. In general, the security verification must be repeated after the application of a security mechanism, because additional vulnerabilities have been introduced. For the application of particular security mechanisms we refer to as "security patterns", it can be ensured that the validity of the security requirements is preserved.

Initial models, security analysis and security-enriched models are addressed in Chapter 4. Threat scenario generation, security verification and revision of security requirements, threats and assumptions are described in Chapter 5. Finally, mechanism application is dealt with in Chapter 7.

3.3. Implementation and Testing

We refer to a security-enriched model whose threat scenario fulfils the global security requirements as "verified secure". If a model is verified secure, then the implementation and testing activities can be carried out.

Coding / Code Generation The security-enriched model can act as a specification based on which the **implementation** is to be developed. If the used specification language supports code generation, it is also possible to automatically generate the implementation or parts thereof. In this case, the code generator must interpret the security-specific information in the model. In particular, abstract representations of cryptographic operations and cryptographic data in the model must be translated to concrete cryptographic operations (e.g. encryption using a particular encryption algorithm) and concrete cryptographic data (e.g. a key of a fixed length).

Coding and code generation aspects are not addressed in this thesis and left as further work.

Security Test Sequence Generation If reliable code generation is not available or the generated code is not suitable for the environment it is to be deployed in (with respect to programming language, size, performance, completeness, interfaces to other parts of the system, etc.), security testing is indispensable to gain confidence in the security of the implementation.

For security testing, the threat scenario can be fully utilised to derive **security test sequences**. Here, security test sequences are sequences of inputs and outputs to a component under test, derived from an attack trace, i.e. an execution trace of the threat scenario.

As exhaustive testing, i.e. testing using all possible sequences of input data, is not feasible, test sequence generation requires the definition of appropriate testing criteria to specify which test sequences should be generated. In security testing, the aim is to discover implementation flaws that lead to the violation of a security requirement.

Test As in the case of code generation, to execute tests given by generated test sequences, abstract test data must be concretised. For this purpose, we require a **concretisation mapping** to specify how abstract input and output data in the test sequences generated from the threat scenario is to be translated to concrete inputs to be sent to and outputs expected from the implementation. Based on the test sequences and the concretisation mapping, the actual test can be carried out automatically by an appropriate **test driver**.

The generation of security test sequences, the specification of a concretisation mapping and an algorithm for a test driver are described in Chapter 6.

3.4. Related Work

[Lot97] describes security-specific development activities in the design phase related to those given in this chapter, based on models specified in the formal method Focus. Explicit modelling of threats and assumptions is not addressed.

[VWW02] shows how to integrate security-specific development activities into a waterfall process to fulfil the requirements stated by the Common Criteria. A top-down oriented approach for the development of security-critical systems is presented in [Eck01]. Security-relevant activities and documents have also been included in the recently published revision of the German V-Modell [MIE$^+$05], but only at a coarse level of detail: specifically, there is an activity "hazard and security analysis" consisting of the identification of possible hazards and threats (here, both security and safety are addressed), risk analysis, and elaboration of appropriate countermeasures. [BBHP04, Pop05] describe a development process for security-critical systems using description techniques of the UML and additional models, e.g. to specify access control policies. A "micro-process" for security analysis is given, consisting of security requirements elicitation, threat modelling and risk analysis, measures design, and correctness check, and this process is repeated at the different levels of detail (business modelling, system requirements elicitation, application architecture). The approach is focused on early development phases, whereas we concentrate on design models and on automation and tool support.

3.5. Summary

We gave a brief overview over security-related development artifacts and activities within a model-based methodology for the development of security-critical systems, considering both analysis/design and implementation/testing aspects.

In the following chapters, we turn our focus to specific parts of this methodology.

4. Modelling Security-Critical Systems

In this chapter, we present the modelling concepts for security-critical systems that form the foundation of the development activities described in the forthcoming chapters.

According to the model-based development methodology described in Chapter 3, one begins with an initial model described in a specification language appropriate for the application domain. Based on the initial model, a security analysis is performed and the information from the security analysis is incorporated into the model, resulting in a security-enriched model. Thus, security analysis can be seen as a model transformation (which, however, cannot be performed automatically). The security-enriched model is the starting point for security-specific development activities such as threat scenario generation, mechanism application or revision of threats and security requirements. The corresponding part of the methodology is depicted in Figure 4.1.

This chapter is structured as follows. In Section 4.1, we present the syntax and semantics of AutoFocus/Quest, a specification language for distributed/embedded systems, which we are going to use as an example language throughout this work to explain our concepts, and describe ways to specify model transformations. Section 4.2 is dedicated to the security extensions used in the security-enriched model to integrate security aspects. We describe the two main case studies, a bank application and the Common Electronic Purse Specifications in Section 4.3 and Section 4.4. We end with references to related work in Section 4.5 and give a summary in Section 4.6.

4.1. A Specification Language for Distributed Systems

Specification languages play a central role in model-based development: they are used to describe aspects such as structure, behaviour or data of the system under consideration (and possibly of its environment). In this section, we introduce the AutoFocus/Quest specification language, which forms the basis of our model-based approach to the development of security-critical systems.

AutoFocus/Quest [HMR+98] is a general-purpose model-based specification language and tool for distributed, reactive systems. It is mainly aimed at the specification and analysis of embedded software, but if used appropriately, its modelling concepts are suitable for systems consisting of networks of components in general.

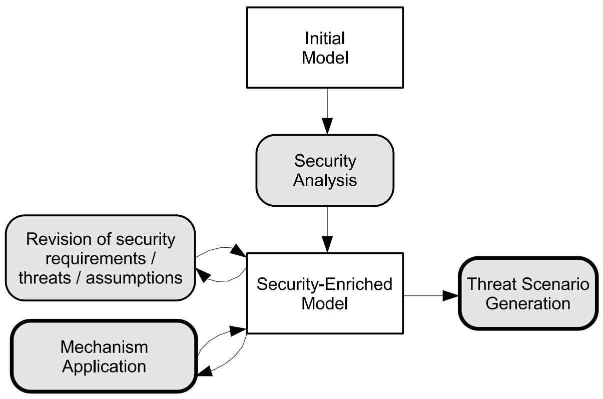

Initial Model
Security Analysis
Revision of security requirements / threats / assumptions
Security-Enriched Model
Threat Scenario Generation
Mechanism Application

Figure 4.1.: Process: Modelling Security-Critical Systems

The formal basis of AutoFocus/Quest are communicating automata, a fairly general formalism which is also part of many other specification languages in its application domain, such as UML-RT or SDL. For the data aspects, AutoFocus/Quest includes a simple functional language named QuestF. Although we had to choose a particular specification language for the presentation of our work, our approach can be fairly easily transferred to other specification languages based on related concepts.

We chose AutoFocus/Quest because of its appropriateness for the considered application domain (distributed systems, in part with embedded components), its conceptual simplicity, and its extensive tool connections. Based on an integrated meta model, the AutoFocus/Quest tool support features simulation, verification, test sequence generation, code generation and an interface for plugins to realise model transformations.

Note that although it turned out that AutoFocus/Quest is well-suited for supporting security-related model aspects and development activities, this was only a minor criterion: the main aim was to utilise a specification language and tool adequate to the application domain and to take advantage of its existing features.

4.1.1. Notation

For formal definitions and formal reasoning, we use standard mathematical notation as far as possible. We write $\emptyset$ for the empty set, $\mathcal{P}(A)$ for the set of subsets of the set A and $\{x : P(x)\}$ for the set of all x such that the predicate $P(x)$ is fulfilled. The symbols $\mathbb{N}$ and $\mathbb{Z}$ stand for the set of natural numbers (including zero) and the set of integers, respectively.

By $f : A_1 \rightarrow A_2$, we specify that f is a function with the functionality $A_1 \rightarrow A_2$. $\mathsf{dom}\, f$ denotes the domain of f (here, A_1), and $\mathsf{rng}\, f := \{f(x) : x \in \mathsf{dom}\, f\}$ denotes its range. We write $\{v_1 \mapsto x_1, \ldots, v_n \mapsto x_n\}$ for the function f with $\mathsf{dom}\, f = \{v_1, \ldots, v_n\}$ and $f(v_j) = x_j$. We write $f|_A$ for the restriction of f to the domain A.

We often work with sequences. We write A^* for the set of finite sequences over a set A, A^+ for the set of non-empty finite sequences over A, and A^∞ for the set of infinite sequences over A. A^ω is defined as $A^\omega := A^* \cup A^\infty$. We write $[x_0, x_1, \ldots, x_n]$ for the sequence consisting of x_0 followed by $x_1, \ldots, x_n$. $\sigma_1 \circ \sigma_2$ denotes the concatenation of the sequences σ_1 and σ_2, $\sigma(i)$ denotes the ith element of σ and σ^i denotes the suffix of σ starting with the ith element. Here, the index of the first element is $i = 0$. If σ is a sequence of functions, we write $\sigma|_A$ for the pointwise application of $|_A$ to the elements of σ.

For a list of frequently used notation, see Appendix A.

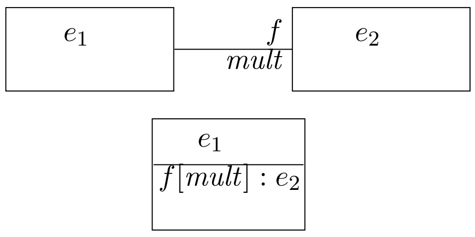
4.1.2. Modelling Formalism

In the following sections, we describe the conceptual meta model representing the abstract syntax of AutoFocus/Quest models. We specify the conceptual meta model of AutoFocus/Quest in the form of a pair ($\mathsf{MEntities}$, $\mathsf{MFunctions}$) consisting of a set $\mathsf{MEntities}$ of names of (meta model) entity sets and a set $\mathsf{MFunctions}$ of names of (meta model) functions with fixed signatures. [1]

As a language to specify the entity sets and functions (i.e., as a meta meta model), we use simplified UML class diagrams. Simplified UML class diagrams also serve as a meta meta model within the model driven architecture of the OMG. The entity set names in $\mathsf{MEntities}$ are given by the class names in the diagram. Association ends and attributes specify function names whose signatures depend on the respective multiplicities, as shown in Figure 4.2. We allow function names to appear more than once if they have the same range: if a diagram defines two functions $f : e_1 \to e_3$ and $f : e_2 \to e_3$, the resulting function specification is $f : (e_1 \cup e_2) \to e_3$. The default multiplicity of attributes is 1. In addition, an entity can only be part of one composition, where compositions are denoted by associations with the symbol ◆——. Finally, a generalisation relationship (denoted by the symbol ◁——) is interpreted as a subset relationship between the entity set corresponding to the subclass and the entity set corresponding to the superclass. Apart from this, all entity sets must be pairwise disjoint.

As an example, the class diagram in Figure 4.3 (a part of the AutoFocus/Quest meta model) specifies two entity set names $\mathsf{Component}$ and $\mathsf{Channel}$, and three functions $\mathsf{subComponents} : \mathsf{Component} \to \mathcal{P}(\mathsf{Component})$, $\mathsf{channels} : \mathsf{Component} \to \mathcal{P}(\mathsf{Channel})$ and $\mathsf{typeof} : \mathsf{Channel} \to \mathsf{Type}$ (the entity Type is specified elsewhere in the diagram).

[1] For simplicity, we use functions instead of general relations and express general relations by a pair of functions if necessary.



$mult$	signature of f
1	$f : e_1 \to e_2$
0..1	$f : e_1 \to e_2$ (partial)
$*$	$f : e_1 \to \mathcal{P}(e_2)$
1..$*$	$f : e_1 \to \mathcal{P}(e_2) \setminus \emptyset$
$*$\{ordered\}	$f : e_1 \to {e_2}^*$
1.. $*$ \{ordered\}	$f : e_1 \to {e_2}^+$

Figure 4.2.: Meta Meta Model: Specification of Meta Model Functions

A **model** $\mathcal{M}$ conforming to a meta model $(\mathsf{MEntities}, \mathsf{MFunctions})$ is an interpretation of the entity set names and function names. $\mathcal{M}$ maps entity set names $e \in \mathsf{MEntities}$ to entity sets $e^{\mathcal{M}}$ and function identifiers $f \in \mathsf{MFunctions}$ to functions $f^{\mathcal{M}}$. In the above example, we obtain a set $\mathsf{Component}^{\mathcal{M}}$ of components, a set $\mathsf{Channel}^{\mathcal{M}}$ of channels, and three functions $\mathsf{subComponents}^{\mathcal{M}}$, $\mathsf{channels}^{\mathcal{M}}$ and $\mathsf{typeof}^{\mathcal{M}}$ that yield the sets of subcomponents and channels of a component and the type of a channel. If it is clear from context that we refer to a fixed model $\mathcal{M}$, for notational convenience we may leave out the superscript $\mathcal{M}$ and use the entity set names and function identifiers and their interpretations interchangeably.

Consistency conditions are formulated as (first-order) predicates over models $\mathcal{M}$. For example, $\forall c \in \mathsf{Component}^{\mathcal{M}} : c \notin \mathsf{subComponents}^{\mathcal{M}}(c)$ is a consistency condition that specifies that a component must not be its own subcomponent. The compositions used in a meta model also state a consistency condition: for all compositions f, f' ($f = f'$ is possible) and all $x \in \mathsf{dom}\, f^{\mathcal{M}}$, there must not exist $x' \in \mathsf{dom}\, f'^{\mathcal{M}}$, $x \neq x'$ such that $f^{\mathcal{M}}(x) \cap f'^{\mathcal{M}}(x') \neq \emptyset$ (where sequences and single values are interpreted as sets). In addition, if $f^{\mathcal{M}}(x)$ is a sequence, it must not contain duplicate elements. As the $\mathsf{subComponents}$ function was represented in Figure 4.3 by a composition, this implies that a $c \in \mathsf{Component}^{\mathcal{M}}$ must only be in the set $\mathsf{subComponents}^{\mathcal{M}}(c')$ for one $c' \in \mathsf{Component}^{\mathcal{M}}$ (i.e., a component cannot be the subcomponent of more than one component).

Lastly, **model transformations** are relations between models, specified as predicates over two models $\mathcal{M}$ (the source model, i.e. the model to be transformed) and $\mathcal{M}'$ (the target model, i.e. the result of the transformation). For example, as part of a transformation, the predicate $\mathsf{Component}^{\mathcal{M}'} = \mathsf{Component}^{\mathcal{M}} \cup \{c\}$ specifies that a component c is added to the set of components.

4.1.3. The Functional Language QuestF

AutoFocus/Quest includes a language to define and manipulate data stored by and passed between components, called QuestF. QuestF is a functional language similar to a subset of Gofer [Jon93].

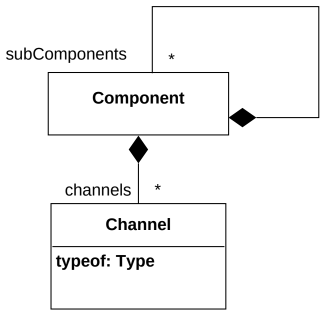


Figure 4.3.: Example Meta Model

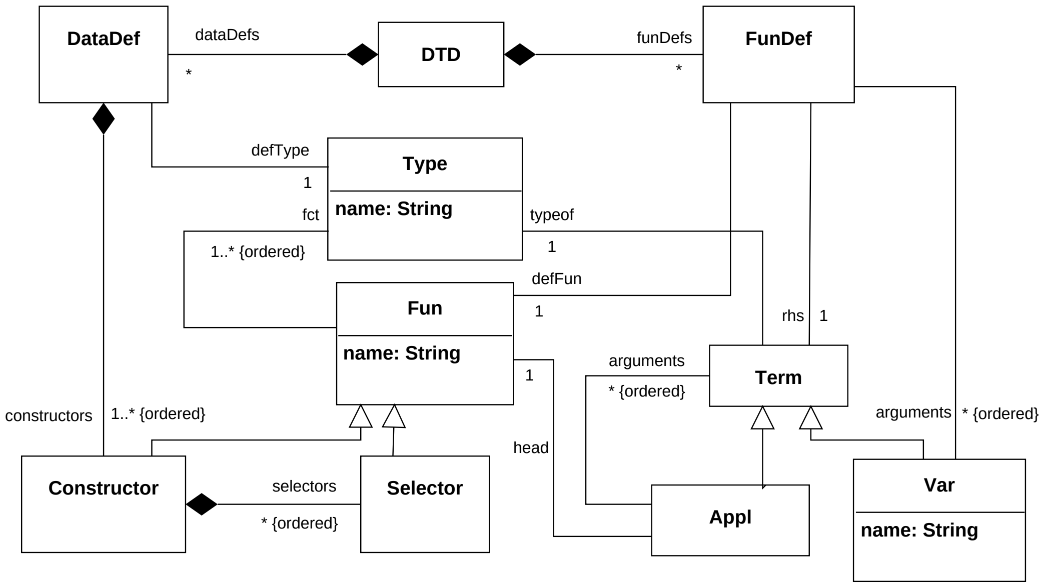


Figure 4.4.: AutoFocus/Quest Meta Model: Abstract Syntax of Language QuestF

A simplified version of the part of the AutoFocus/Quest meta model describing the abstract syntax of QuestF is shown in Figure 4.4. In this thesis, we restrict ourselves to the description of the syntax and semantics of the features relevant for our purposes: hierarchical data types, terms and function definitions.

Data Type Definitions (DTDs)

In QuestF, hierarchical (algebraic) data types are defined by *data type definitions.* A data type definition $d \in \mathsf{DataDef}$ defines a type $\mathsf{defType}(d) \in \mathsf{Type}$ by a non-empty sequence of *constructors* $\mathsf{constructors}(d) \in \mathsf{Constructor}^+$. For each constructor $c \in$

$\mathsf{Constructor}$ there is a (possibly empty) sequence $\mathsf{selectors}(c)$ of *selectors* to extract the arguments of a constructor application.

Constructors and selectors are elements of a set Fun of *functions*. Functions $f \in \mathsf{Fun}$ have functionalities $\mathsf{fct}(f) \in \mathsf{Type}^+$, denoted as $(\mathsf{type}_1, \ldots, \mathsf{type}_n) \rightarrow \mathsf{type}_{n+1}$, where $\mathsf{type}_1, \ldots$, type_n are the argument types and type_{n+1} is the result type.[2]

For $\mathsf{fct}(f) = (\mathsf{type}_1, \ldots, \mathsf{type}_n) \rightarrow \mathsf{type}_{n+1}$, we also write $f : (\mathsf{type}_1, \ldots, \mathsf{type}_n) \rightarrow \mathsf{type}_{n+1}$.

Let $\mathsf{datadef}^k \in \mathsf{DataDef}$ (for $k \in \{1, \ldots, |\mathsf{DataDef}|\}$) be the data type definition defining the type $\mathsf{type}^k = \mathsf{defType}(\mathsf{datadef}^k)$, with $\mathsf{constructors}(\mathsf{datadef}^k) = [\mathsf{C}^k_1, \ldots, \mathsf{C}^k_m]$ and $\mathsf{selectors}(\mathsf{C}^k_i) = [\mathsf{sel}^k_{i1}, \ldots, \mathsf{sel}^k_{in^k_i}]$. For $\mathsf{datadef}^k$ to be consistent, the functionalities of the constructors must be $\mathsf{C}^k_i : (\mathsf{type}^k_{i1}, \ldots, \mathsf{type}^k_{in^k_i}) \rightarrow \mathsf{type}^k$, where type^k_{ij} are the argument types of the constructor C^k_i. The arities n^k_i of the constructors can also be 0. Besides, the selectors must have the functionalities $\mathsf{sel}^k_{ij} : (\mathsf{type}^k) \rightarrow \mathsf{type}^k_{ij}$. For each type $t \in \mathsf{Type}$, there must be only one data type definition $d \in \mathsf{DataDef}$ with $\mathsf{defType}(d) = t$.

In the concrete syntax, the data type definition $\mathsf{datadef}^k$ is written as follows:

$$\begin{array}{ll} \textbf{data}\ \mathsf{type}^k = & \mathsf{C}^k_1(\mathsf{sel}^k_{11} : \mathsf{type}^k_{11}, \ldots, \mathsf{sel}^k_{1n^k_1} : \mathsf{type}^k_{1n^k_1}) \\ & \mid \\ & \ldots \\ & \mid \\ & \mathsf{C}^k_m(\mathsf{sel}^k_{m1} : \mathsf{type}^k_{m1}, \ldots, \mathsf{sel}^k_{mn^k_m} : \mathsf{type}^k_{mn^k_m}); \end{array}$$

Here, types, constructors and selectors are represented by their names, given by the meta model function $\mathsf{name} : (\mathsf{Type} \cup \mathsf{Fun} \cup \mathsf{Var}) \rightarrow \mathsf{String}$, where String is a set of names (represented as strings). Names for the selectors sel^k_{ij} can be omitted in the concrete syntax. If a selector name is omitted, AutoFocus/Quest automatically generates a default one when parsing the data type definition.

Intuitively, a data element of type type^k can have as a value a constructor application $\mathsf{C}^k_i(x_1, \ldots, x_{n^k_i})$, and the selector sel^k_{ij} can be used to extract x_j from such a value.

We assume that one of the data types type^k is Bool, defined as follows:

$$\textbf{data}\ \mathsf{Bool} = \mathsf{True} \mid \mathsf{False};$$

Terms and Evaluation

Figure 4.4 also specifies the abstract syntax of terms in QuestF. A term is either a variable $v \in \mathsf{Var}$, in the concrete syntax represented by its name, or a function

[2]Note that here we make no distinction to the usual mathematical notation for functionalities already used previously on the meta model level.

application $a \in \mathsf{Appl}$ (i.e., $\mathsf{Term} = \mathsf{Var} \cup \mathsf{Appl}$). A function application has a function $f = \mathsf{head}(a) \in \mathsf{Fun}$ as its head and a sequence $\mathsf{arguments}(a) \in \mathsf{Term}^*$ of terms as its arguments and is denoted as $f(t_1, \ldots, t_n)$.

By the function $\mathsf{typeof} : \mathsf{Term} \rightarrow \mathsf{Type}$, types are assigned to terms. Type correctness is defined as follows:

Definition 4.1.1. A term t is *type correct* if $t \in \mathsf{Var}$ or $t = f(t_1, \ldots, t_n)$ where the terms $t_1, \ldots, t_n$ are type correct and $\mathsf{fct}(f) = (\mathsf{typeof}(t_1), \ldots, \mathsf{typeof}(t_n)) \rightarrow \mathsf{typeof}(t)$.

Definition 4.1.2. (subterm, free variables, substitution) We often work with the property of a term t being a subterm of a term t', denoted by $t \trianglelefteq t'$; with its set of free variables $\mathsf{freeVar}(t)$; and with the application of a substitution $[v_1/t_1, \ldots, v_n/t_n]$ to t resulting in the term $t[v_1/t_1, \ldots, v_n/t_n]$ where the variables $v_1, \ldots, v_n$ have been replaced simultaneously by the terms $t_1, \ldots, t_n$. These notations are interpreted in the usual way. Substitutions are treated as sequences of pairs of terms and variables and thus can be concatenated using the operator $\circ$.

A term is evaluated by reducing it to normal form, which only consists of applications of constructor functions. [3]

Definition 4.1.3. A term t *is in normal form* if it is type correct and $\forall s \trianglelefteq t : \exists c \in \mathsf{Constructor}, n \geq 0, t_1, \ldots, t_n \in \mathsf{Term} : s = c(t_1, \ldots, t_n)$. Note that it is possible that $n = 0$, in which case we have a constructor c without arguments. By $\mathsf{Value} := \{t \in \mathsf{Term} : t \text{ in normal form}\} \cup \{\bot\}$ we denote the set of terms in normal form, including a symbol $\bot$ for the empty/error value, and by $\mathsf{Value}(\mathsf{type}^k) := \{t \in \mathsf{Value} : \mathsf{typeof}(t) = \mathsf{type}^k\} \cup \{\bot\}$ we denote the values of type type^k. $\mathsf{Value}(\mathsf{type}^k)$ is the carrier set of the type type^k.

Let $\beta : \mathsf{Var} \rightarrow \mathsf{Value}$ be a valuation of variables, and I be an interpretation of the function symbols $f \in \mathsf{Fun}$, with $\mathsf{I}(f) : \mathsf{Value}(\mathsf{type}_1) \times \ldots \times \mathsf{Value}(\mathsf{type}_n) \rightarrow \mathsf{Value}(\mathsf{type}_{n+1})$ if $\mathsf{fct}(f) = (\mathsf{type}_1, \ldots, \mathsf{type}_n) \rightarrow \mathsf{type}_{n+1}$. The evaluation $\mathsf{eval}_\beta : \mathsf{Term} \rightarrow \mathsf{Value}$ of terms is defined as follows:

$$\begin{array}{lcl} \mathsf{eval}_\beta(v) & = & \beta(v) \quad \text{for } v \in \mathsf{Var} \\ \mathsf{eval}_\beta(f(t_1, \ldots, t_n)) & = & \mathsf{I}(f)(\mathsf{eval}_\beta(t_1), \ldots, \mathsf{eval}_\beta(t_n)) \end{array}$$

Predefined Functions

We assume that for a given DTD, the following functions are predefined:

[3]In terms of algebraic data types, this corresponds to an initial algebra semantics.

- the standard boolean functions $\&\& : (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}$, $|| : (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}$, $=> : (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}$ and $\mathsf{not} : (\mathsf{Bool}) \to \mathsf{Bool}$ with the usual strict interpretations for boolean and, or, implication and not;
- an equality function $==^k$: $(\mathsf{type}^k, \mathsf{type}^k) \to \mathsf{Bool}$ for each type type^k, with $\mathsf{I}(==^k)(x_1, x_2) = \mathsf{True}$ if $x_1 = x_2$ and $\mathsf{I}(==^k)(x_1, x_2) = \mathsf{False}$ otherwise, but always $\mathsf{I}(==^k)(x_1, x_2) =\perp$ if $x_1 =\perp$ or $x_2 =\perp$ (strictness); and
- an immediate-if function $\mathsf{if}^k : (\mathsf{Bool}, \mathsf{type}^k, \mathsf{type}^k) \to \mathsf{type}^k$ for each type type^k, with

  $\mathsf{I}(\mathsf{if}^k)(\mathsf{True}, x_2, x_3) = x_2$, $\mathsf{I}(\mathsf{if}^k)(\mathsf{False}, x_2, x_3) = x_3$ and $\mathsf{I}(\mathsf{if}^k)(\perp, x_2, x_3) =\perp$.

The constructors and selectors specified in the data type definitions have the following interpretations:

- $\mathsf{I}(\mathsf{C}^k_i)(x_1, \ldots, x_{n^k_i}) = \mathsf{C}^k_i(x_1, \ldots, x_{n^k_i})$ if $x_i \neq\perp$ for all $i : 1 \leq i \leq n^k_i$, and $\mathsf{I}(\mathsf{C}^k_i)(x_1, \ldots, x_{n^k_i}) =\perp$ otherwise; and
- $\mathsf{I}(\mathsf{sel}^k_{ij})(x) = x_j$ if $x = \mathsf{C}^k_i(x_1, \ldots, x_j, \ldots, x_{n^k_i})$, and $\mathsf{I}(\mathsf{sel}^k_{ij})(x) =\perp$ otherwise.

Finally, for each constructor C^k_i there is a *discriminator* $\mathsf{is\_C}^k_i \in \mathsf{Fun}$ with functionality $\mathsf{is\_C}^k_i : (\mathsf{type}^k) \to \mathsf{Bool}$ and $\mathsf{I}(\mathsf{is\_C}^k_i)(\mathsf{C}^k_i(x_1, \ldots, x_{n^k_i})) = \mathsf{True}$, $\mathsf{I}(\mathsf{is\_C}^k_i)$ $(\perp) =\perp$ and $\mathsf{I}(\mathsf{is\_C}^k_i)(x) = \mathsf{False}$ otherwise.

For better readability, in the concrete syntax we write &&, ||, => and $==^k$ in infix form, and $\mathsf{if}^k(t_1, t_2, t_3)$ as "if^k t_1 then t_2 else t_3 fi". In addition, as QuestF terms can be statically type checked, we write $t_1 == t_2$ as an abbreviation for $t_1 ==^k t_2$ (where $\mathsf{typeof}(t_1) = \mathsf{typeof}(t_2) = \mathsf{type}^k$). The latter gives us a simple kind of polymorphy (which is in general not supported in QuestF) without cluttering up syntax and semantics too much. In the same way, we write if as an abbreviation for if^k.

Function Definitions

QuestF supports user-defined functions. A function definition $fd \in \mathsf{FunDef}$ defines a function $\mathsf{defFun}(fd) \in \mathsf{Fun}$. It has a sequence of arguments $\mathsf{arguments}(fd) \in \mathsf{Var}^*$ and a right hand side $\mathsf{rhs}(fd) \in \mathsf{Term}$.[4] The concrete syntax of a function definition with $\mathsf{defFun}(fd) = f$, $\mathsf{arguments}(fd) = [v_1, \ldots, v_n]$ and $\mathsf{rhs}(fd) = t_f$ is

$$\textbf{fun } f(v_1, \ldots, v_n) = t_f;$$

[4]Actual QuestF function definitions allow the use of multiple alternatives and pattern matching, which we leave out here for simplification.

$$\begin{array}{lcl}
\textbf{data}\ \mathsf{TAgent} & = & \mathsf{A} \mid \mathsf{B}; \\
\textbf{data}\ \mathsf{TMessage} & = & \mathsf{EmptyTMessage} \\
 & & \mid \mathsf{Request}(\mathsf{getInitiator} : \ \mathsf{TAgent}) \\
 & & \mid \mathsf{Reply}(\mathsf{getResponder} : \ \mathsf{TAgent});
\end{array}$$

Figure 4.5.: Example of QuestF Data Type Definition

For each function definition, $\mathsf{freeVar}(\mathsf{rhs}(fd))$ must be the set $\{v_1, \dots, v_n\}$ of the arguments.

Let $\{v_1 \mapsto x_1, \dots, v_n \mapsto x_n\}$ denote a valuation β with $\beta(v_i) = x_i$ for $1 \leq i \leq n$. The (call by value) semantics of a function definition is then given by

$$\mathsf{I}(f)(x_1, \dots, x_n) = \mathsf{eval}_{\{v_1 \mapsto x_1, \dots, v_n \mapsto x_n\}}(t_f)$$

Each function $f \in \mathsf{Fun}$ must be either a predefined function or it must be defined via a function definition. The entity set DTD integrates data type and function definitions and corresponds to a textual specification document. We allow only one such specification to be present, which globally defines the data types and functions used in a model. That is, $|\mathsf{DTD}| = 1$.

Example 4.1.4. Figure 4.5 shows the definition of a simple data type $\mathsf{TMessage}$ for messages, based on a data type TAgent for agent names. A possible value of type $\mathsf{TMessage}$ would be $\mathsf{Request}(\mathsf{A})$. $\mathsf{eval}_\beta(\mathsf{is\_Request}(\mathsf{Request}(\mathsf{A})))$ yields True and the result of $\mathsf{eval}_\beta(\mathsf{getInitiator}(\mathsf{Request}(\mathsf{A})))$ is A.

Integer Type

The language QuestF also includes an integer type Int. If needed in a model, we assume we have an additional type definition

$$\textbf{data}\ \mathsf{Int} = \mathsf{Zero} \mid \mathsf{Succ}(\mathsf{pred} : \mathsf{Int});$$

for (positive) integers, and corresponding function definitions such as

$$\textbf{fun}\ + (x, y) = \mathsf{if}\ \mathsf{is\_Zero}(x)\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{Succ}(+(\mathsf{pred}(x), y))\ \mathsf{fi};$$

We write “0” as an abbreviation for Zero, “1” as an abbreviation for $\mathsf{Succ}(\mathsf{Zero})$, etc. Note that integers will not play a central role in our models of security-critical systems. In particular, we represent security-critical data such as cryptographic keys or random challenges symbolically rather than by integer values.

4.1.4. Structural and Behavioural Views

Based on the underlying functional language, we can specify the structure and behaviour of the system under consideration.

Like the abstract syntax for data type and function definitions, the abstract syntax to specify structural and behavioural aspects of a system is part of the AutoFocus/Quest meta model. We refer to such parts as *views* on a model. Each view has its own concrete syntax. The fact that these views are tightly coupled through the meta model facilitates ensuring their mutual consistency and defining operations on the model such as transformations or the export to external tools for verification or test case generation.

Figure 4.6 shows a simplified version of the part of the AutoFocus/Quest meta model relevant for the structural and behavioural views. In the following, we describe the views and their concrete syntax in more detail.

System Structure Diagrams (SSDs)

The system structure diagram (SSD) view specifies the interfaces and the architecture of a system. An SSD is similar to a data flow diagram and consists of named components with input and output ports for receiving and sending messages. Figure 4.7 shows the part of the AutoFocus/Quest meta model relevant for the SSD view. A component $c \in \mathsf{Component}$ is either *atomic* ($\mathsf{subComponents}(c) = \emptyset$), or it consists of a number of subcomponents and of directed channels connecting the ports of the subcomponents to each other or to the ports of c (i.e., to the environment of c). A port can be the destination port $\mathsf{destP}(ch)$ of at most one channel ch, such that the received message is unambiguously defined. Besides, to simplify the formulation of model transformations on channels, we also require that a port can be the source port $\mathsf{sourceP}(ch)$ of at most one channel ch. If necessary, multicast (which can be specified in the original AutoFocus/Quest by connecting more than one channel to the same source port) can be explicitly modelled by introducing an additional component receiving a message and sending multiple copies. Ports and channels are typed, with the consistency condition that $\forall ch \in \mathsf{Channel} : \mathsf{typeof}(ch) = \mathsf{typeof}(\mathsf{sourceP}(ch)) = \mathsf{typeof}(\mathsf{destP}(ch))$. We allow ports to be used as variables in terms. Therefore, the set Port is included in the set Var.

Example 4.1.5. Figure 4.8 shows a simple AutoFocus/Quest SSD as it is depicted in the concrete syntax used by the AutoFocus/Quest SSD editor. It consists of two subcomponents A and B that have an output and input port each, named $\mathsf{A.Req}$ and $\mathsf{A.Rep}$, respectively $\mathsf{B.Rep}$ and $\mathsf{B.Req}$, which are connected via two channels.

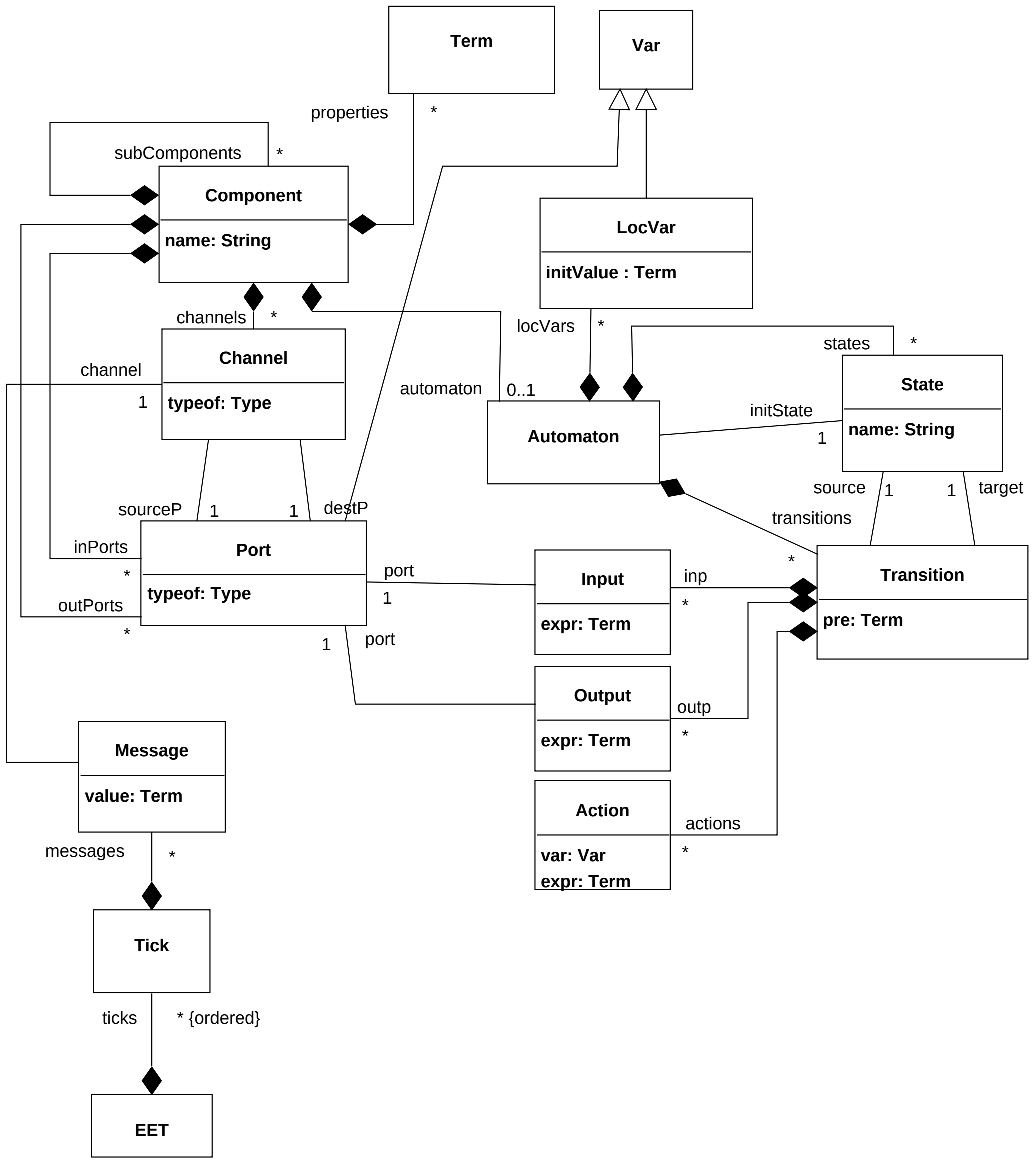


Figure 4.6.: AutoFocus/Quest Meta Model: Structural and Behavioural Views

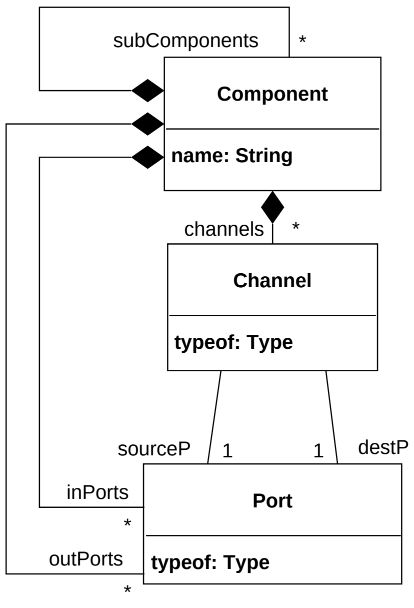


Figure 4.7.: AutoFocus/Quest Meta Model: SSD View

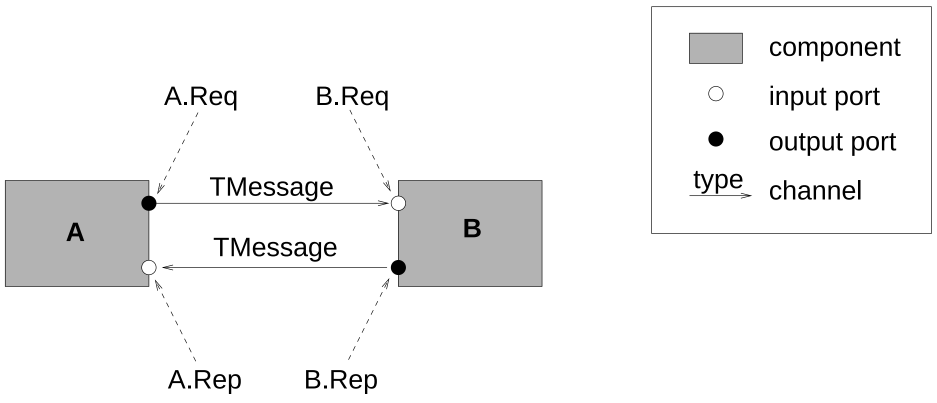


Figure 4.8.: Example AutoFocus/Quest System Structure Diagram

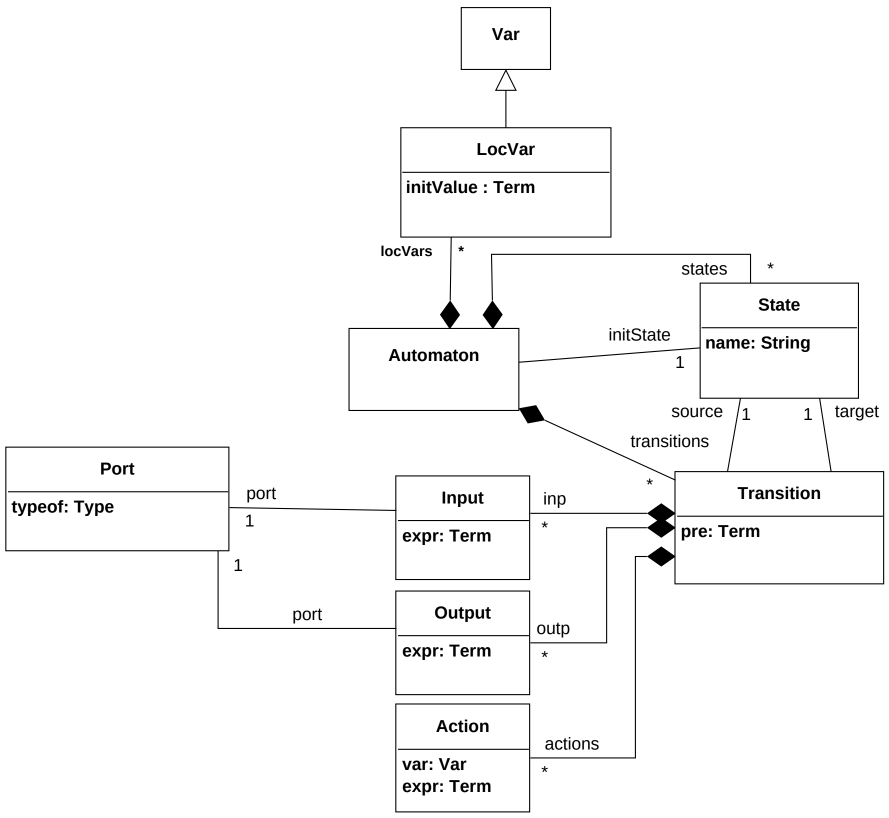


Figure 4.9.: AutoFocus/Quest Meta Model: STD View

State Transition Diagrams

The behaviour of components can be specified in AutoFocus/Quest by the State Transition Diagram (STD) view. STDs define extended finite state machines, which have a data state as well as a control state. In AutoFocus/Quest, STDs can also be hierarchical (i.e., states can have substates). However, we will not make use of this feature.

Figure 4.9 shows the part of the AutoFocus/Quest meta model relevant for the STD view. An STD $aut \in \mathsf{Automaton}$ (where Automaton is the entity set for STDs), consists of a set $\mathsf{states}(aut) \subseteq \mathsf{State}$ of named control states, one of which is the initial state $\mathsf{initState}(aut)$, of a set $\mathsf{locVars}(aut) \subseteq \mathsf{LocVar}$ of local variables (together forming the data state), and of a set $\mathsf{transitions}(aut)$ of transitions. Local variables $v \in \mathsf{LocVar}$ are variables (i.e., $\mathsf{LocVar} \subseteq \mathsf{Var}$) for which an initial value $\mathsf{initValue}(v)$ is specified. STDs can be assigned to components via the function $\mathsf{automaton} : \mathsf{Component} \rightarrow \mathsf{Automaton}$. Local variables cannot be shared between components (expressed by the composition association in the diagram).

Each transition $tr \in \mathsf{transitions}(aut)$ has the following attributes:

- source and target state $\mathsf{source}(tr)$, $\mathsf{target}(tr) \in \mathsf{State}$;
- a boolean expression $\mathsf{pre}(tr) \in \mathsf{Term}$, the *precondition* (guard) for firing tr;
- $\mathsf{inp}(tr) \in \mathcal{P}(\mathsf{Input})$, a set of *input expressions*, where each input expression $i \in \mathsf{Input}$ consists of a port $\mathsf{port}(i)$ and a term $\mathsf{expr}(i)$ consisting only of variables and constructors: a *pattern* that must match the value read on $\mathsf{port}(i)$;
- $\mathsf{outp}(tr) \in \mathcal{P}(\mathsf{Output})$, a set of *output expressions*, where each output expression $o \in \mathsf{Output}$ consists of a port $\mathsf{port}(o)$ and a term $\mathsf{expr}(o)$ whose value is output to $\mathsf{port}(o)$ when tr is fired; and
- $\mathsf{actions}(tr) \in \mathsf{Action}$, a set of *actions*, where each action $a \in \mathsf{Action}$ consists of a variable $\mathsf{var}(a)$ and a term $\mathsf{expr}(a)$ whose value is assigned to $\mathsf{var}(a)$ when tr is fired.

Only local variables and the free variables in the input expressions may appear as free variables in the precondition and in the terms of the output expressions and actions. For convenience of notation, we sometimes write input expressions i and output expressions o as pairs $(\mathsf{port}(i), \mathsf{expr}(i))$ and $(\mathsf{port}(o), \mathsf{expr}(o))$, and actions a as pairs $(\mathsf{var}(a), \mathsf{expr}(a))$. Likewise, we write transitions $tr \in \mathsf{Transition}$ as 6-tuples $tr = (\mathsf{source}(tr), \mathsf{pre}(tr), \mathsf{inp}(tr), \mathsf{outp}(tr), \mathsf{actions}(tr), \mathsf{target}(tr))$.

Example 4.1.6. Figure 4.10 shows a simple AutoFocus/Quest STD that could specify the behaviour of the component B depicted in Figure 4.8. In the concrete syntax of STDs, states are depicted by ellipses and the initial state is marked with a black dot. The transitions are annotated with "$\mathsf{pre}(tr) : \mathsf{inp}(tr) : \mathsf{outp}(tr) : \mathsf{actions}(tr)$", where $\mathsf{inp}(tr)$ is denoted as "$p_1?t_1; p_2?t_2; \ldots$", $\mathsf{outp}(tr)$ as "$p_1!t_1; p_2!t_2; \ldots$", and $\mathsf{actions}(tr)$ as "$v_1 = t_1; v_2 = t_2; \ldots$" with $p_i \in \mathsf{Port}$, $v_i \in \mathsf{Var}$, $t_i \in \mathsf{Term}$. Local variables with their types and initial values are shown in an additional box, in the form "$v : \mathsf{typeof}(v) = \mathsf{initValue}(v)$". A component B with the behaviour specified in Figure 4.10 would accept a request message of the form Request(x), store it in the local variable stored_req (which is of type TMessage and has the initial value EmptyTMessage) and return a reply message Reply(B).

Extended Event Traces

Extended Event Traces (EETs) in AutoFocus/Quest represent system runs, similarly to message sequence charts [ITU96]. AutoFocus/Quest EETs include concepts for hierarchy and repetition. For our purposes (depicting test sequences and

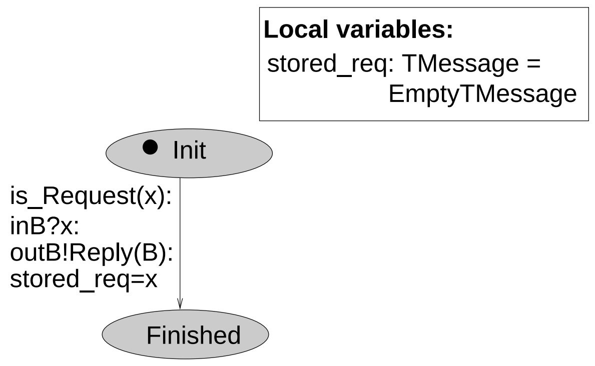


Figure 4.10.: Example AutoFocus/Quest State Transition Diagram

attack scenarios), it suffices to view an EET as a sequence of messages exchanged between the components via their channels.

The meta model part relevant for the definition of EETs is shown in Figure 4.11. An EET $e \in \mathsf{EET}$ is a sequence $\mathsf{ticks}(e)$ of ticks. Each tick consists of a set of messages $m \in \mathsf{Message}$, which are valuations of channels.

For convenience of notation, we also denote an EET directly as a sequence of channel valuations $\beta_1, \beta_2, \ldots$ with $\beta_i : \mathsf{Channel} \rightarrow \mathsf{Value}$.

Example 4.1.7. Figure 4.12 shows an EET for the system in Figure 4.8, where component A sends a Request(A) message to B and B sends the corresponding reply. Here, the arrows denote the messages and the ticks are separated by dashed lines.

Properties

Finally, one can assign a set of properties $\mathsf{properties}(c) \in \mathcal{P}(\mathsf{Term})$ to a component c, which can be used for verification and test sequence generation.

In AutoFocus/Quest, properties are formulated in temporal logic (see e.g. [Eme90] for a general introduction). AutoFocus/Quest allows to specify both Computation Tree Logic (CTL) and Linear-Time Temporal Logic (LTL) properties. In this thesis, we will only make use of LTL properties, respectively of predicates on traces (which LTL properties can be translated to). Properties assigned to components are specified in AutoFocus/Quest with the help of a separate property editor.

For simplicity reasons, there is no additional entity for temporal logic properties in AutoFocus/Quest. Instead, special terms are used:

Definition 4.1.8. For the formulation of temporal logic properties we require that the set Fun includes a set of *LTL operators* $\mathsf{LTLOp} = \{\mathsf{X}\ ,\ \mathsf{U}\ , \mathsf{F}\ , \mathsf{G}\ \}$. The operators

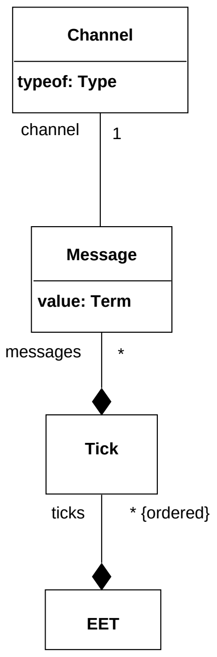


Figure 4.11.: AutoFocus/Quest Meta Model: EET View

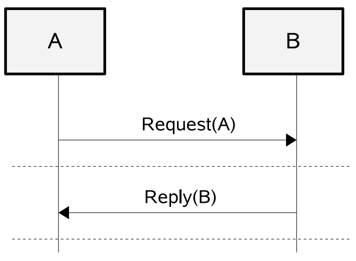


Figure 4.12.: Example AutoFocus/Quest Extended Event Trace

X , F and G have the functionalities $(\mathsf{Bool}) \to \mathsf{Bool}$, and the operator U has the functionality $(\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}$. A term $t \in \mathsf{Term}$ *is an LTL term* if it contains an LTL operator.

If $t \in \mathsf{Term}$ is an LTL term, we require that for all its subterms $f(t_1, \ldots, t_n) \trianglelefteq t$ that are LTL terms, either f is an LTL operator, or $\exists n \geq 1 : \mathsf{fct}(f) = \mathsf{Bool}^n$ (i.e., LTL terms can only be connected by boolean functions).

LTL terms cannot be interpreted via eval_β. The only way LTL terms may appear in a model is via the association $\mathsf{properties}(c)$. In particular, the transition attributes of State Transition Diagrams and the messages in Extended Event Traces must not contain LTL terms. In any case, we require that for all $pr \in \mathsf{properties}(c)$, $\mathsf{typeof}(pr) = \mathsf{Bool}$. Besides, as free variables in pr only ports, local variables or the special variable state_{aut} referring to the control state of the automaton aut can occur. The usage of state_{aut} is explained below.

The temporal operators have the following meanings:

- $pr_1 \; \mathsf{U} \; pr_2$ ("pr_1 until pr_2"): pr_2 holds in some state in an execution sequence, and until that state, pr_1 holds,
- $\mathsf{X} \; pr$ ("nexttime pr"): pr holds in the immediate successor state,
- $\mathsf{F} \; pr$ ("future pr"): pr holds in some state, and
- $\mathsf{G} \; pr$ ("globally pr"): pr holds in all states.

Example 4.1.9. A possible property of the system specified in part by the SSD in Figure 4.8 is

$$\mathsf{F} \; (\mathsf{A.Req} == \mathsf{Request(A)})$$

stating that in all executions, the component A will eventually issue a request $\mathsf{Request(A)}$ via port $\mathsf{A.Req}$.

4.1.5. Semantics of AutoFocus/Quest Models

In this section, we give an operational semantics for AutoFocus/Quest models.

Our semantics is based on synchronous composition using a global clock and message buffers that hold a message for exactly one time tick:

- At each execution step, a transition is fired in each of the automata in the model at the same time, depending on the values of the local variables and of the input ports of the component the automaton is assigned to.
- The values written to the output ports are available to the connected components at the following execution step.

This execution model was chosen in the project Quest for the integration of AutoFocus/Quest with various simulation and verification tools for the following reasons:

- It is the natural way of describing hardware oriented embedded systems, the main application area of AutoFocus/Quest.
- The state space has a fixed size (when using finite data types), which facilitates the use of automated verification techniques such as model checking.

The used execution model is also general enough for the specification of arbitrary networks of communicating components: for example, if necessary, buffer components can be introduced to allow communication with arbitrary delay. Synchronous composition is commonly used in the analysis of cryptographic protocols as well [Pfi98] and therefore provides a good basis for the treatment of security-critical systems.

In particular, our definition of the semantics of an AutoFocus/Quest model is based on the translation to state transition graphs (which we will call discrete systems in this work) presented in [WLPS00], which was derived from [HSE97, PS99]. It is formulated in terms of the abstract syntax presented in the previous sections.

Definition 4.1.10. A *discrete system* $\mathcal{D} = (V, I, T)$ consists of:

- V, a set of *state variables*;
- $I(\beta)$, an *initial condition*, given as a predicate on valuations β of V, that characterises the initial states; and
- $T(\beta, \beta')$, a *transition relation*, given as a predicate on a pair β, β' of valuations of V.

A *computation* of $\mathcal{D}$ is an infinite sequence $\sigma = [\beta_0, \beta_1, \beta_2, \ldots]$ of valuations satisfying $\Psi_{\mathcal{D}}(\sigma)$, where

$$\Psi_{(V,I,T)}(\sigma) = I(\beta_0) \wedge \forall i \geq 0 : T(\beta_i, \beta_{i+1})$$

Semantics of Atomic Components

The semantics of an atomic component $c \in \mathsf{Component}^{\mathcal{M}}$ in a model $\mathcal{M}$ can only be defined if it has an associated automaton $aut = \mathsf{automaton}^{\mathcal{M}}(c)$. To be able to represent the control state as a valuation of a variable as well, we assume that for each automaton aut there is an implicit data type declaration

$$\textbf{data}\ \mathsf{states}_{aut} = s_1 \mid \ldots \mid s_n\ ;$$

with $\{s_1, \ldots, s_n\} = \mathsf{states}^{\mathcal{M}}(aut)$, and a variable $\mathsf{state}_{aut} \in \mathsf{Var}^{\mathcal{M}}$ with that type. The semantics of c is then given as a discrete system $[\![c]\!]_{\mathcal{M}} = (V_{\mathcal{M};c}, I_{\mathcal{M};c}, T_{\mathcal{M};c})$ with

$$\begin{array}{lcl}
V_{\mathcal{M};c} & = & \mathsf{inPorts}^{\mathcal{M}}(c) \cup \mathsf{outPorts}^{\mathcal{M}}(c) \cup \mathsf{locVars}^{\mathcal{M}}(aut) \cup \{\mathsf{state}_{aut}\} \\
I_{\mathcal{M};c}(\beta) & = & \beta(\mathsf{state}_{aut}) = \mathsf{initState}^{\mathcal{M}}(aut) \wedge \\
 & & \bigwedge_{x \in \mathsf{locVars}^{\mathcal{M}}(aut)} \beta(x) = \mathsf{initValue}^{\mathcal{M}}(x) \wedge \\
 & & \bigwedge_{x \in \mathsf{outPorts}^{\mathcal{M}}(c)} \beta(x) = \perp \\
T_{\mathcal{M};c}(\beta, \beta') & = & \bigvee_{tr \in \mathsf{transitions}^{\mathcal{M}}(aut)} T_{\mathcal{M};c;tr}(\beta, \beta') \ \vee\ T_{\mathcal{M};c;\mathsf{idle}}(\beta, \beta')
\end{array}$$

Thus, the (global) state of c is a valuation of its ports and of the local variables and the control state of *aut*. Initially, *aut* is in its initial control state, the local variables have their initial values and the output ports are empty. Empty ports are represented by $\perp$. The transition relation is a disjunction of transition relations $T_{\mathcal{M};c;tr}$ for the transitions of *aut* and an “idle” transition, as explained in the following.

For the definition of $T_{\mathcal{M};c;tr}$, we first consider the input patterns. We define a pattern matching condition $\mathsf{pmCond}(p, t)$, which is a set of terms that evaluates to True if a term p matches the pattern t, and a substitution $\mathsf{pmSubst}(p, t)$ that maps each free variable in t to a term yielding the value of this variable in the matching:

$$\begin{array}{lcl}
\mathsf{pmCond}(p, t) & = & \emptyset, \text{ if } t \in \mathsf{Var} \\
\mathsf{pmCond}(p, \mathsf{C}_i^k(t_1, \ldots, t_{n_i^k})) & = & \{\mathsf{is\_C}_i^k(p)\} \cup \bigcup_{1 \leq j \leq n_i^k} \mathsf{pmCond}(\mathsf{sel}_{ij}^k(p), t_j) \\
\mathsf{pmSubst}(p, t) & = & [t/p], \text{ if } t \in \mathsf{Var} \\
\mathsf{pmSubst}(p, \mathsf{C}_i^k(t_1, \ldots, t_{n_i^k})) & = & \mathsf{pmSubst}(\mathsf{sel}_{i1}^k(p), t_1) \circ \ldots \circ \mathsf{pmSubst}(\mathsf{sel}_{in_i^k}^k(p), t_{n_i^k})
\end{array}$$

For example, for the pattern $t = \mathsf{Request}(x)$ with the above definition we have $\mathsf{pmCond}(p, t) = \{\mathsf{is\_Request}(p)\}$ and $\mathsf{pmSubst}(p, t) = [x/\mathsf{getInitiator}(p)]$. Thus, the term $\mathsf{Request(A)}$ matches the pattern t and x is substituted by A (the result of $\mathsf{getInitiator(Request(A))}$) in the matching. For a transition tr, we define $\mathsf{pmCond}(tr) = \bigcup_{(p,t) \in \mathsf{inp}^{\mathcal{M}}(tr)} \mathsf{pmCond}(p, t)$ as the union of the pattern matching conditions for its input patterns, and likewise $\mathsf{pmSubst}(tr)$ as the concatenation of the corresponding substitutions.

$T_{\mathcal{M};c;tr}$ is then given by:

$$
\begin{array}{lr}
T_{\mathcal{M};c;tr}(\beta,\beta') = & \\
\quad \beta(\mathsf{state}_{aut}) = \mathsf{source}^{\mathcal{M}}(tr) \wedge & (1) \\
\quad \beta'(\mathsf{state}_{aut}) = \mathsf{target}^{\mathcal{M}}(tr) \wedge & \\
\quad \forall (p,t) \in \mathsf{inp}^{\mathcal{M}}(tr) & \\
\qquad \beta(p) \neq \perp \ \wedge \forall t \in \mathsf{pmCond}(tr) : \mathsf{eval}_{\beta}(t) = \mathsf{True} \wedge & (2) \\
\qquad \mathsf{eval}_{\beta}(\mathsf{pre}^{\mathcal{M}}(tr)(\mathsf{pmSubst}(tr))) = \mathsf{True} \wedge & (3) \\
\quad \forall (p,t) \in \mathsf{outp}^{\mathcal{M}}(tr) : \beta'(p) = \mathsf{eval}_{\beta}(t(\mathsf{pmSubst}(tr))) \wedge & (4) \\
\quad \forall (v,t) \in \mathsf{actions}^{\mathcal{M}}(tr) : \beta'(v) = \mathsf{eval}_{\beta}(t(\mathsf{pmSubst}(tr))) \wedge & \\
\quad \forall p \in \mathsf{outPorts}^{\mathcal{M}}(c) \setminus \{p : (p,t) \in \mathsf{outp}^{\mathcal{M}}(tr)\} : \beta'(p) = \perp \ \wedge & (5) \\
\quad \forall v \in \mathsf{locVars}^{\mathcal{M}}(aut) \setminus \{v : (v,t) \in \mathsf{actions}^{\mathcal{M}}(tr)\} : \beta'(v) = \beta(v) & (6)
\end{array}
$$

The above definition specifies that when a transition fires, the current control state must be the source state and upon firing, it must change to the destination state (1). For the input patterns, values must be available and the pattern matching conditions must evaluate to True (2), as must the precondition (where pattern variables are substituted by their actual values) (3). The output ports and local variables are set accordingly, again taking into account the pattern substitutions (4). Finally, the ports not mentioned in the output expressions are cleared (5) and the local variables not mentioned in the output expressions remain unchanged (6).

Finally, when no transition can be fired, the system can take an "idle" step, in which all output ports are cleared and all local variables are left unchanged. The idle step is given by

$$
\begin{array}{rcl}
T_{\mathcal{M};c;\mathsf{idle}}(\beta,\beta') & = & \neg\exists\beta'' : \bigvee_{tr\in\mathsf{transitions}^{\mathcal{M}}(aut)} T_{c,tr}(\beta,\beta'') \wedge \\
& & \beta'(\mathsf{state}_{aut}) = \beta(\mathsf{state}_{aut}) \wedge \\
& & \bigwedge_{p\in\mathsf{outPorts}^{\mathcal{M}}(c)} \beta'(p) = \perp \ \wedge \\
& & \bigwedge_{v\in\mathsf{locVars}^{\mathcal{M}}(aut)} \beta'(v) = \beta(v)
\end{array}
$$

Semantics of Composed Components

If the component $c \in \mathsf{Component}^{\mathcal{M}}$ has subcomponents, then we define its semantics in terms of the semantics of its subcomponents. Therefore, each subcomponent of c must either be composed of subcomponents as well, or if it is atomic, it must have an associated automaton. We ignore automata possibly associated to composed components.

The (global) state of a component consists of the states of its subcomponents and of the values on its channels and ports. The subcomponents execute in lock-step: at each execution step, a transition is fired in each of the subcomponents at the same time. Thus, the transition relation is the conjunction of the transition relations of the subcomponents. Communication takes place via the channels. For this reason, the values of source and destination port of the channels are equated for the initial state and for the successor state in the transition relation.

Thus, the semantics of a composed component c is $[\![c]\!]_{\mathcal{M}} = (V_{\mathcal{M};c}, I_{\mathcal{M};c}, T_{\mathcal{M};c})$, where

$$\begin{array}{lcl}
V_{\mathcal{M};c} & = & \bigcup_{c' \in \mathsf{subComponents}^{\mathcal{M}}(c)} V_{\mathcal{M};c'} \cup \\
 & & \mathsf{inPorts}^{\mathcal{M}}(c) \cup \mathsf{outPorts}^{\mathcal{M}}(c) \cup \mathsf{channels}^{\mathcal{M}}(c) \\
I_{\mathcal{M};c}(\beta) & = & \bigwedge_{c' \in \mathsf{subComponents}^{\mathcal{M}}(c)} I_{\mathcal{M};c'} \wedge \\
 & & \bigwedge_{ch \in \mathsf{channels}^{\mathcal{M}}(c)} (\beta(\mathsf{sourceP}^{\mathcal{M}}(ch)) = \beta(\mathsf{destP}^{\mathcal{M}}(ch)) = \beta(ch)) \\
T_{\mathcal{M};c}(\beta, \beta') & = & \bigwedge_{c' \in \mathsf{subComponents}^{\mathcal{M}}(c)} T_{\mathcal{M};c'} \wedge \\
 & & \bigwedge_{ch \in \mathsf{channels}^{\mathcal{M}}(c)} (\beta'(\mathsf{sourceP}^{\mathcal{M}}(ch)) = \beta'(\mathsf{destP}^{\mathcal{M}}(ch)) = \beta'(ch))
\end{array}$$

Semantics of Properties

We define the semantics of a property pr as a predicate $[\![pr]\!](\sigma)$ on sequences $\sigma = [\beta_0, \beta_1, \beta_2, \ldots]$ of valuations:

$$\begin{array}{lcl}
[\![pr]\!](\sigma) & = & \mathsf{eval}_{\sigma(0)}(pr), \text{ if } pr \text{ is not an LTL term} \\
[\![f(pr_1, \ldots, pr_n)]\!](\sigma) & = & \mathsf{I}(f)([\![pr_1]\!](\sigma), \ldots, [\![pr_n]\!](\sigma)) \\
[\![\mathsf{X}\ pr]\!](\sigma) & = & [\![pr]\!](\sigma^1) \\
[\![pr_1\ \mathsf{U}\ pr_2]\!](\sigma) & = & \exists j : [\![pr_2]\!](\sigma^j) \wedge \forall i : 0 \leq i < j : [\![pr_1]\!](\sigma^i) \\
[\![\mathsf{F}\ pr]\!](\sigma) & = & [\![\mathsf{True}\ \mathsf{U}\ pr]\!](\sigma) \\
[\![\mathsf{G}\ pr]\!](\sigma) & = & [\![\mathsf{not}(\mathsf{F}\ \mathsf{not}(pr))]\!](\sigma)
\end{array}$$

With $\sigma(i)$, we denote the i-th step in the sequence (i.e., β_i), and with σ^j, we denote the suffix of σ starting from the index j (i.e., $[\beta_j, \beta_{j+1}, \ldots]$). We say that a property pr is fulfilled with respect to a discrete system $\mathcal{D}$ if it is fulfilled for all its computations, i.e. $\forall \sigma : \Psi_{\mathcal{D}}(\sigma) \Rightarrow [\![pr]\!](\sigma)$.

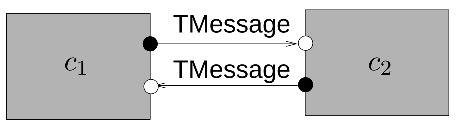
4.1.6. Model Transformations in AutoFocus/Quest

As explained in Section 4.1.2, we define model transformations as predicates relating two models $\mathcal{M}$ and $\mathcal{M}'$. Model transformations for AutoFocus/Quest models refer to the entities and functions defined in the AutoFocus/Quest meta model.

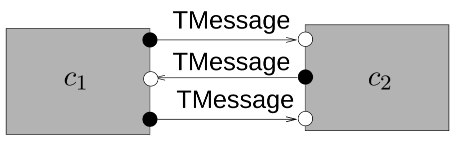
Figure 4.13 gives an example: here, a new channel from the component c_1 to the component c_2 is added.

Such a transformation $\mathsf{AddChannel}(c_1, c_2)(\mathcal{M}, \mathcal{M}')$ can be specified as follows:

- $\mathsf{outPorts}^{\mathcal{M}'}(c_1) = \mathsf{outPorts}^{\mathcal{M}}(c_1) \cup \{p\}$, where $p \notin \mathsf{Port}^{\mathcal{M}}$ is a new port
- $\mathsf{inPorts}^{\mathcal{M}'}(c_2) = \mathsf{inPorts}^{\mathcal{M}}(c_2) \cup \{p'\}$, where $p' \notin \mathsf{Port}^{\mathcal{M}}, p' \neq p$ is a new port



(a) Source Model $\mathcal{M}$



(b) Target Model $\mathcal{M}'$

Figure 4.13.: Model Transformation Example

- $\mathsf{Port}^{\mathcal{M}'} = \mathsf{Port}^{\mathcal{M}} \cup \{p, p'\}$
- $\mathsf{Channel}^{\mathcal{M}'} = \mathsf{Channel}^{\mathcal{M}} \cup \{ch\}$, where $ch \notin \mathsf{Channel}^{\mathcal{M}}$
- $\mathsf{sourceP}^{\mathcal{M}'}(ch) = p$, $\mathsf{destP}^{\mathcal{M}'}(ch) = p'$, $\mathsf{typeof}^{\mathcal{M}'}(ch) = \mathsf{TMessage}$

We assume that all entity sets and functions not mentioned stay unchanged (e.g. $\mathsf{Component}^{\mathcal{M}'} = \mathsf{Component}^{\mathcal{M}}$), and for the arguments not mentioned, the functions keep their values (e.g. $\forall c \neq c_1 : \mathsf{outPorts}^{\mathcal{M}'}(c) = \mathsf{outPorts}^{\mathcal{M}}(c)$).

Defining Model Transformations in the AutoFocus/Quest Tool

According to a classification given in [SK03], architectural approaches for defining model transformations in a tool can be divided into three categories: direct model manipulation, intermediate representation, and transformation language support. The AutoFocus/Quest tool framework supports all three approaches:

Direct Model Transformation: the tool offers access to its internal model representation, which can be manipulated using a general-purpose programming language. In AutoFocus/Quest, the internal representation of a model is given by a graph of Java objects that are instances of meta model classes. The meta model classes are generated from a textual description of the meta model and offer APIs for their manipulation. AutoFocus/Quest includes a plugin mechanism that makes it possible to make model transformations programmed in Java available to the user via new menu items.

```
AddChannel($c_1, c_2$) :=
    new $p$ : Port.
        new $p'$ : Port.
            new $ch$ : Channel.
                result $p \in$ outPorts($c_1$) $\wedge$
                result $p' \in$ inPorts($c_2$) $\wedge$
                result sourceP($ch$) $= p \wedge$
                result destP($ch$) $= p' \wedge$
                result typeof($ch$) $=$ TMessage
```

Figure 4.14.: ODL Description for AddChannel Transformation

Intermediate Representation: the tool offers export to and import from a model representation in a standard form, which can be transformed by an external tool. AutoFocus/Quest offers both an interface to representations in XML (which can be transformed using existing XML tools such as XSLT) and to a proprietary representation called QML (Quest Markup Language).

Transformation Language Support: the tool offers a language for explicitly expressing transformations. For this purpose, the language ODL (Operation Definition Language) was integrated into AutoFocus/Quest [Sch01]. ODL is an extension of first order predicate logic by additional operators to introduce new model elements and to specify the result of a transformation (called new and result). In ODL, the above transformation for adding a channel between two components would be described as shown in Figure 4.14 (slightly adjusted to our formalism). Other approaches that fall into this category include visual transformation languages e.g. based on graph grammars, which are however not supported by AutoFocus/Quest.

All these approaches have their advantages and drawbacks, which we cannot address here for space reasons — [SK03] gives a general overview.

To avoid having to require knowledge from the reader of syntax and semantics of a specific transformation language, for describing model transformations in this thesis we stick to the general specification described above by logic predicates between the entities and relations of source and target model, where a precise description is necessary.

Prototypical implementations of the transformations that appear in this thesis have been implemented in Java. However, any of the other approaches could have been used as well.

4.2. Security Extensions

In this section, we describe how AutoFocus/Quest is extended to support the modelling of security-critical systems. For this purpose, we define means to integrate security aspects into a model, based on a generic model extension mechanism. In particular, we consider global security requirements that form the security policy of a system, threats against the system and assumptions restricting the impact of the threats, and security mechanisms responsible for enforcing the security policy by countering the threats and realising the assumptions. Besides, we extend the QuestF language with cryptographic data types and operations and with means to specify "secret" data items not guessable by an intruder.

As communicating systems are the primary application area of AutoFocus/ Quest, the main focus is on communication security aspects. An application of AutoFocus/Quest to access control is presented in [DGJW04] (joint work with M. Deubler, J. Grünbauer, and J. Jürjens), which could be formulated in the original AutoFocus/Quest syntax without making use of extensions.

We refer to a model with integrated security aspects as *security-enriched model.* The additional information contained in the security-enriched model is obtained by a security analysis. For our purposes, we assume that this information is already available at a sufficiently detailed level. For more information on approaches to perform security analyses, the reader is referred to [Eck01].

4.2.1. Model Extension Mechanism

To be able to include threats, assumptions and security mechanisms in the models, we define an extension mechanism that allows us to add arbitrary annotations to the model elements. The extension mechanism is a simplified variant of the extension mechanisms of the UML [OMG03], which are based on *stereotypes* to define new kinds of model elements based on existing ones, *tag definitions*, i.e. definitions of additional meta attributes of model elements, and *constraints* that must be obeyed by the model elements which they are attached to. Tag definitions and constraints are usually part of a stereotype.

As for our purposes we do not need to define new kinds of model elements, but rather add annotations to existing model elements, we resort to a slightly simpler concept: we omit the stereotypes and just include tag definitions and the corresponding tags in our meta model.

We will give constraints on our use of the tags, but we define these constraints globally. A consistent annotated model must fulfil all defined constraints.

The extension mechanism is depicted in Figure 4.15 as an extension of the AutoFocus/Quest meta model shown in Figure 4.4 and Figure 4.6. All model entities are subsumed under an entity ModelElement. An arbitrary number of tags

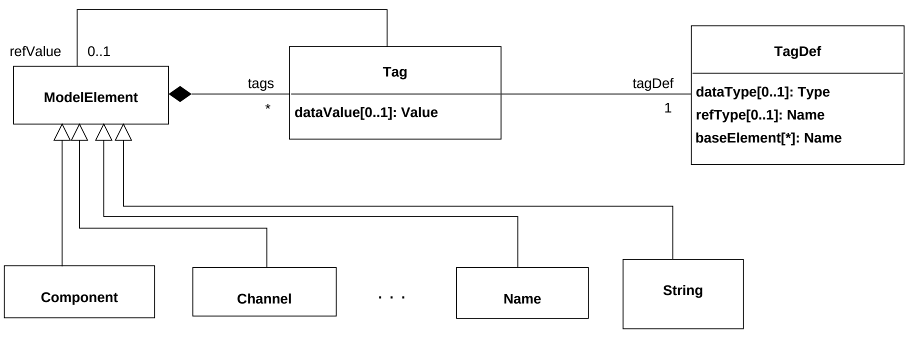


Figure 4.15.: Model Extension Mechanism

$\mathsf{tags}(m) \in \mathcal{P}(\mathsf{Tag})$ can be attached to any model element $m \in \mathsf{ModelElement}$. Each tag t is related to its tag definition $td = \mathsf{tagDef}(t)$, which defines a number of base elements $\mathsf{baseElement}(td) \subseteq \mathsf{Name}$ and either a data type $\mathsf{dataType}(td) \in \mathsf{Type}$ or a reference type $\mathsf{refType}(td) \in \mathsf{Name}$. Here, Name is the set of names of the original entity sets, i.e. $\mathsf{Name} = \{\text{'}\mathsf{Component}\text{'}, \text{'}\mathsf{Channel}\text{'}, \ldots\}$. Correspondingly, the tag t itself either has a data value $\mathsf{dataValue}(t)$ or a reference value $\mathsf{refValue}(t)$ which can refer to another model element.

The following global constraints must hold if the extension mechanism is used:

- A tag definition must either define a data type or a reference type, but not both.
- A tag t must respect its tag definition: it must either contain a data value of the corresponding type if the tag definition defines a data type, or a reference value to a corresponding model element if the tag definition defines a reference type. Moreover, a tag can only be attached to a model element belonging to one of the entity sets given in $\mathsf{baseElement}(\mathsf{tagDef}(t))$.

For convenience of notation, we also write tags t as pairs $(\mathsf{tagDef}(t), \mathsf{dataValue}(t))$ or $(\mathsf{tagDef}(t), \mathsf{refValue}(t))$. If $\mathsf{dataType}(\mathsf{tagDef}(t)) = \mathsf{Bool}$, we just write $\mathsf{tagDef}(t)$ instead of $(\mathsf{tagDef}(t), \mathsf{True})$.

Example 4.2.1. Figure 4.16 shows an annotated version of the system structure diagram in Figure 4.8. In the concrete syntax of the editor, tags (td, v) are denoted by $\{td = v\}$. Again, True can be omitted. In Figure 4.8, both channels between A and B have a public tag (with the data value True). As we will see later, this specifies that the channels are subject to attacks.

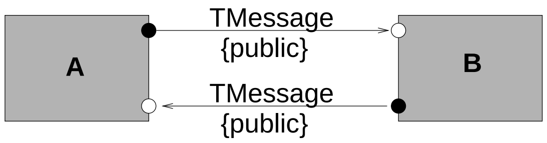


Figure 4.16.: Annotated AutoFocus/Quest System Structure Diagram

Security-Related Annotations

The main characteristic of security-critical systems is that the *global security requirements* must be fulfilled in the presence of *threats*. Threats can be restricted by *assumptions*, which are taken for granted when a security verification is performed. To fulfil its security requirements and assumptions, a system must include *security mechanisms*, which can be realised using *cryptography*.

In the following sections, we show how global security requirements, threats, assumptions, mechanisms and cryptography can be added to a system specification, using the presented model extension mechanism. We do this by defining security annotations that are attached to the model elements.

At this point, with the exception of the cryptography-related annotations these annotations should be regarded as comments. They form the basis for threat scenario generation and verification, security testing, and mechanism application, which will be described in Chapters 5, 6 and 7.

The security annotations defined in this thesis are given by a set $\mathsf{SecTagDef} \subseteq \mathsf{TagDef}$ of tag definitions, as specified in Table 4.1. The role of the annotation (requirement, threat, assumption, security mechanism) is also given in this table.

4.2.2. Global Security Requirements

The global security requirements reflect the global security policy of a system and are the result of a first threat/risk analysis [Eck01]. Commonly occurring global security requirements and their general characteristics were described in Section 2.1. Obtaining and refining security requirements is a very important issue, but out of scope of this work — we assume that as a result of such a requirements engineering process we already have security requirements detailed enough such that they can be expressed in terms of the (formal) model of the system. Note that the validity of the security requirements depends on the specified threats and assumptions. The security requirements must therefore be verified with respect to the threat scenario, which includes the intruder behaviour (see Chapter 5).

Table 4.1.: Security-Related Tag Definitions

$t \in$ SecTagDef	dataType(t)	refType(t)	baseElement(t)	Role
SecRequirement	Bool	—	{Term}	Requirement
critical	Bool	—	{Component, Channel, Transition, State}	Threat
public	Bool	—	{Component, Channel}	Threat
replace	Bool	—	{Component}	Threat
node	Bool	—	{Component}	Assumption
secret	Bool	—	{Channel}	Assumption
auth	Bool	—	{Channel}	Assumption
avail	Bool	—	{Channel}	Assumption
integrity	Bool	—	{Channel}	Assumption
noreplay	Bool	—	{Channel}	Assumption
protocol	—	Component	{Channel}	Mechanism
protoChannel	String	—	{Channel, Port}	Mechanism
protoInst	String	—	{Channel}	Mechanism
Encr	Bool	—	{Constructor}	Cryptography
Sign	Bool	—	{Constructor}	Cryptography
Hash	Bool	—	{Constructor}	Cryptography
Mac	Bool	—	{Constructor}	Cryptography
Key	Bool	—	{Type}	Cryptography

Global Security Requirements in AutoFocus/Quest

To specify security requirements in AutoFocus/Quest models, we use the temporal logic properties that can be attached to components c via properties(c) (see Section 4.1.4). As described in Section 4.1.5, to be fulfilled with respect to a system, a temporal logic property must be fulfilled for all its computations. This corresponds to the universal nature of security requirements. We annotate security requirements $pr \in$ Term with a boolean tag SecRequirement $\in$ tags(pr).

Property Patterns

The correct formal specification of security requirements in temporal logic is often difficult, as is interpreting such specifications. To alleviate this problem, we take advantage of the fact that in formal specifications of common security requirements, such as confidentiality or authentication, often similar patterns occur, e.g. the requirement that something should *never* happen.

[DAC99] defines patterns for frequently used temporal logic expressions. Based on this work, we introduce the following patterns for temporal logic expressions that occur in the security requirements we consider, as abbreviations of more complex formulas in temporal logic:

- never(pr) (where $pr \in$ PropTerm) specifies that pr is not fulfilled at any step in an execution sequence. It is an abbreviation for not(F pr).
- precedes(pr_1, pr_2) specifies that if pr_2 is fulfilled in some step of an execution sequence, pr_1 is fulfilled in the same or an earlier step. It is an abbreviation for (not(pr_2) U pr_1) || G not(pr_2).
- leadsto(pr_1, pr_2) specifies that if pr_1 is fulfilled in some step of an execution sequence, pr_2 is fulfilled in the same or some later step. It is an abbreviation for G (pr_1 => F pr_2).

Referring to Intruder Knowledge

In the specification of security requirements, it is often necessary to refer to the intruder knowledge (for instance, the intruder must not learn a certain value). For this purpose, we define a special function learnedIntruder^k : (type^k) $\rightarrow$ Bool for each type type^k. learnedIntruder$^k(t)$ evaluates to True at a particular point in execution if the value of t is contained in the intruder knowledge. As usual, we abbreviate learnedIntruder^k to learnedIntruder if the required type can be statically determined. The learnedIntruder^k functions must only appear in security requirements. Their interpretation depends on the threat scenario (see Chapter 5).

Patterns for Common Global Security Requirements

In the following, we show how instances of the common security requirements listed in Section 2.1 can be formalised with the help of the above described property patterns and give examples with respect to the simple system depicted in Figure 4.8 on page 38.

- **Confidentiality** requirements can be specified by patterns of the form never(learnedIntruder(x)), where $x \in$ Value. This means that the intruder should not be able to learn x in any possible computation.

  For instance, in the example system, never(learnedIntruder(Reply(B))) specifies that the intruder will never learn the message Reply(B).
- **Authenticity** can be specified by patterns of the form precedes(pr_1, pr_2), where pr_1 and pr_2 do not contain LTL operators, pr_1 refers to the local variables and ports of a component c_1, and pr_2 refers to the local variables and ports of another component c_2. In other words, if c_2 has reached a state

characterised by pr_2, c_2 can validly assume that c_1 has reached a state characterised by pr_1 (for instance, c_1 has indeed sent a particular message). This is similar to Lowe's definitions of authentication with respect to authentication protocols [Low97], which state that an authentication specification is guaranteed to an initiator if, whenever the initiator completed a run of the protocol, apparently with a certain responder, then the responder has previously been running the protocol and the runs of the initiator and the responder correspond to each other in a way dependent on the kind of authentication specification. In our case, c_2 plays the role of the initiator, to whom authentication is guaranteed, and c_1 plays the role of the responder. [Low97] defines various kinds of authentication specifications with different strengths, which can be reflected by pr_1 and pr_2 (excluding agreement, which cannot be formalised by a precedes pattern as it requires a one-to-one relationship between the two runs).

As an example, $\mathsf{precedes}(\mathsf{A.Req} == \mathsf{Request}(\mathsf{A}), \mathsf{B.Rep} == \mathsf{Reply}(\mathsf{B}))$ specifies that B only sends a reply message towards A if A (and not another party or the intruder) has sent a request message towards B. This corresponds to the simplest kind of authentication defined in [Low97], aliveness: there is no guarantee that A indeed intended to communicate with B, or that the request and reply messages correspond to each other. These aspects are not considered in the simple example. To allow for the specification of stronger requirements, the necessary information must be included in the model. For instance, in local variables or messages it can be explicitly stated who a particular party intended to communicate with. Examples for more complex requirements are given within the case studies (see Sections 4.3, 4.4, and 5.4).

- **Integrity** can be specified in an analogous way to authentication: here, we interpret pr_2 as a certain effect and pr_1 as the legal action that causes this effect. Thus, $\mathsf{precedes}(pr_1, pr_2)$ states that the effect pr_2 can only be caused by the action pr_1 and not by unauthorised manipulation by the intruder.

  For example, $\mathsf{precedes}(\mathsf{A.Req} == \mathsf{Request}(\mathsf{A}), \mathsf{B.Req} == \mathsf{Request}(\mathsf{A}))$ specifies that if B receives the request message Request(A), this message must actually have been sent by A (which is not true if the intruder can modify the message during the transmission). In a similar way, manipulation of data stored within a component can be addressed, e.g. by specifying that certain data can only be changed as the result of particular legal write actions.

- **Non-repudiation** can also be formulated using the precedes pattern: here, pr_2 acts as proof for c_2 that c_1 must have been in a state characterised by pr_1. In this context, one often also considers threats where c_2 tries to obtain the proof by manipulation. This can be modelled by allowing c_2 to cooperate with the intruder, e.g. by supplying the intruder with the secrets that c_2 has access to.

- **Availability** can be specified by patterns of the form $\mathsf{leadsto}(pr_1, pr_2)$, where pr_1 and pr_2 are again terms that refer to the ports and local variables of components c_1 and c_2 and do not contain LTL operators. pr_1 and pr_2 are interpreted as a legal action and its intended effect. $\mathsf{leadsto}(pr_1, pr_2)$ states that the effect is always caused when the action is performed and in particular cannot be prevented by the intruder.

  For instance, $\mathsf{leadsto}(\mathsf{A.Req} == \mathsf{Request(A)}, \mathsf{B.Rep} == \mathsf{Reply(B)})$ specifies that if A sends a request, B will eventually reply.

- **Fair Exchange** can be specified by patterns of the form $\mathsf{leadsto}(pr_1, pr_2)$ or $\mathsf{precedes}(pr_1, pr_2)$, depending on the point of view and the temporal relationship of the states characterised by pr_1 and pr_2. Again, pr_1 is a term without LTL operators referring to the local variables and ports of c_1 and pr_2 is a term without LTL operators referring to the local variables and ports of c_2.

  From the point of view of c_1, $\mathsf{leadsto}(pr_1, pr_2)$ specifies that whenever c_1 is in a state characterised by pr_1, eventually c_2 reaches a state characterised by pr_2, which ensures that c_2 does not gain an advantage over c_1. Conversely, from the point of view of c_2, $\mathsf{precedes}(pr_1, pr_2)$ specifies that whenever c_2 reaches a state characterised by pr_2, then c_1 must have reached a state characterised by pr_1 such that c_1 does not gain an advantage over c_2.

  An example would be a transaction where money is transferred between two electronic purses. Firstly, when money is withdrawn from the sending purse, it must always be credited to the receiving purse. Secondly, when money is credited to the receiving purse, it must always have been withdrawn from the sending purse. If withdrawal and credit of money are specified by pr_1 and pr_2, these properties are given as $\mathsf{leadsto}(pr_1, pr_2)$ and $\mathsf{precedes}(pr_1, pr_2)$ respectively.

Note that the above description is intended as a guideline, not as a complete list of formalisations of security requirements. Variations or more complex versions of the described properties may occur. The exact security requirements closely depend on the system under consideration and must be the result of a careful requirements engineering process. Besides, we focus on security requirements that can be formalised as trace properties. The treatment of security requirements that cannot be formalised as trace properties, such as non-interference, would require an extension of the specification formalism. [RSG$^+$00] describes the formalisation of security requirements in more detail (in the context of security protocols modelled using the formal method CSP).

4.2.3. Threats

Critical System Parts

The critical annotation specifies the security-critical parts of a system, i.e. the parts that are assumed to be relevant for the system's security (because security-critical information is processed or transmitted). We use the critical annotation as an indication given by the designer that (formal or informal) security verification and testing should focus on the annotated parts of the model.

In the SSD view, the critical annotation can be attached to components and channels. In the STD view, it can be attached to states and transitions. For the purpose of security analysis and testing, the system can be sliced to comprise only the critical system parts.

The use of the critical annotation involves the following consistency conditions:

- If a channel or a component carries any other security annotations, it is assumed to be critical and must also be annotated with the critical annotation.
- If a state or a transition is annotated with critical, the component associated with the corresponding automaton must be annotated with critical as well, because security-critical information is processed in this component.
- A component that has a subcomponent annotated with critical must also be annotated with critical.
- Security requirements must only refer to ports of components annotated with critical or to local variables or states of automata associated with components annotated with critical.

In the following, we will mainly concentrate on the security-critical parts of a system. Therefore, we will omit the critical tag if it is clear that it should be present (i.e. because already other security annotations are attached to the corresponding model element).

Public Components and Channels

The public annotation describes the access possibilities of an intruder to components and channels of a system:

- If a channel *ch* is annotated with public, then *ch* is a channel whose messages can be accessed and manipulated by the intruder. Otherwise, *ch* behaves like an ordinary channel in AutoFocus/Quest (i.e., it is a dedicated connection between two ports, which cannot be influenced).

- If a component c is annotated with public, it can be replaced by an intruder having access to all secrets contained in c (e.g., because the intruder can access and manipulate the program).

We do not mark states or transitions with public, assuming that an intruder can either manipulate the complete behaviour of a component or not manipulate it at all.

Replaceable Components

In addition to full access to components and channels, it is also possible that the intruder acts in place of a component c and communicates with the rest of the system, without having access to the secrets of c. As an example, an intruder could try to act in place of a honest client component in an Internet-based system, or could construct a fake smart card communicating with a card reader instead of the real one.

The possibility of such an attack is modelled by annotating the component c with replace. This attack is less powerful than annotating all channels connected to c with public, because in the replace case, the intruder cannot obtain the outputs of a honest c-component.

4.2.4. Assumptions

Assumptions can be attached to the model elements to restrict the capabilities of the intruder. For the security analysis, it is then sufficient to consider the restricted capabilities.

As our main focus is communication aspects of distributed systems, we concentrate on assumptions on channels.

Encapsulated Components

An encapsulated component is a component whose internals cannot be accessed or manipulated by an intruder, such as for example a smart card. We model this by the annotation node on components.

The annotation of a component c with node implies the following consistency condition:

- The channels and subcomponents of c must not be annotated with public or replace. Besides, all subcomponents of c must be annotated with node as well.

Secret Channels

Data sent on a channel annotated with secret is assumed to be only readable by the component connected to the channel via its destination port. In particular, it cannot be read by the intruder.

Authentic Channels

Data received via a channel annotated with auth must have been sent by the component connected to the channel via its source port. In particular, the intruder cannot write own data on the channel (but he can insert replays of data that has already been sent).

Integrity-Preserving Channels

Data received via a channel annotated with integrity must have been sent before by a honest component (which does not necessarily have to be connected to the channel). This is a generalisation of the auth annotation.

Channels Ensuring Availability

If a channel is annotated with avail, it can be assumed that a message sent out via the channel will eventually be delivered to the component connected to the channel via its destination port.

Replay-Protected Channels

If a channel is annotated with noreplay, it is assumed to be protected from replays. This indicates that the intruder cannot repeat a message already sent on the channel, unless the intruder can generate the message himself. The noreplay annotation is only meaningful in combination with secret. On a channel annotated with secret but not with noreplay, the intruder cannot read the transferred data, but he can repeat it. This is for example the case if the transferred data is encrypted on the lower level.

4.2.5. Security Mechanisms

Security mechanisms are functions of a system that are introduced to ensure that the security requirements are fulfilled by countering the threats and realising the assumptions. Examples for security mechanisms are encryption or electronic signatures, access control components, the use of cryptographic protocols or the use of hardware such as smart cards.

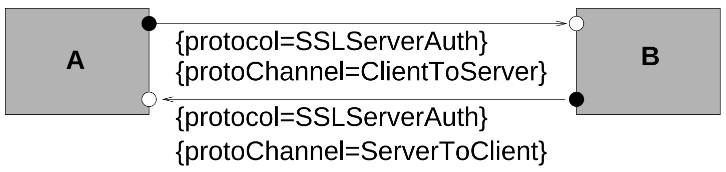


Figure 4.17.: System Structure Diagram with protocol Annotation

Following our focus on communication aspects, we introduce annotations for security mechanisms on channels.

Layered Protocols

Security mechanisms on channels can often be represented in the form of layered protocols: the communication between two components is "tunnelled", which means that the messages are transferred by a lower-level protocol. Basic security mechanisms such as the encryption of messages sent over a channel can be modelled in this way, but also more complex ones such as the tunnelling of the messages over a previously established SSL connection.

We specify the use of a layered protocol by the protocol annotation, which contains the name of a component c as its value. If $(\mathsf{protocol}, c) \in \mathsf{tags}(ch)$ for a channel ch, the messages sent via ch are assumed to be tunnelled using the protocol realised by the component c. We refer to such channels as *tunnelled channels.*

Example 4.2.2. Figure 4.17 shows two components A and B communicating via the SSL protocol with server authentication, realised by the SSLServerAuth component.

The component c that specifies the tunnelling protocol is an ordinary AutoFocus/Quest component with special annotations. Such a *protocol component* contains pairs of input and output ports, which we call *protocol channels.* A message that is sent to the source port of a protocol channel is processed by the protocol and then delivered to the destination port of the protocol channel. The protocol channels are identified by annotating the corresponding input and output ports with $(\mathsf{protoChannel}, n)$, where $n \in \mathsf{String}$ is a name identifying a particular protocol channel.

If a protocol component has more than one protocol channel, in addition to the protocol annotation it must be specified on the tunnelled channel which of them is to be used. This is done by adding the annotation $(\mathsf{protoChannel}, n)$ to the corresponding channel on the upper layer.

Finally, it is possible that more than one instance of the same tunnelling protocol is to be used between two components. If the protocol has more than one protocol

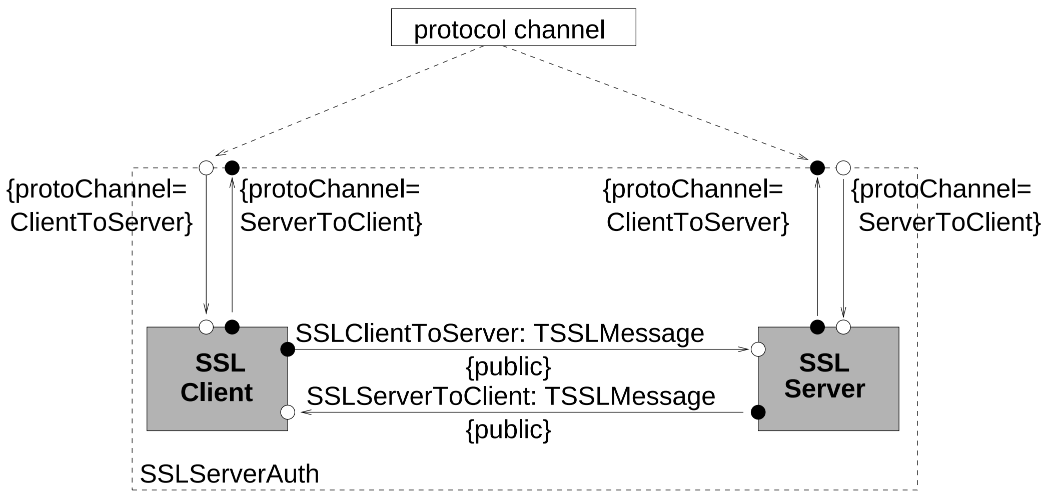


Figure 4.18.: System Structure Diagram for Protocol Component SSLServerAuth

channel, the mapping of the channels to the protocol instances must be specified using the annotation (protoInstance, n). Channels with the same (protoInstance, n) tag are assumed to use the same protocol instance.

Example 4.2.3. Figure 4.18 shows the system structure diagram for the protocol component SSLServerAuth with two protocol channels ClientToServer and ServerToClient. The protoChannel annotations in Figure 4.17 specify how the channels between the two components A and B are mapped to the protocol channels. If a second instance of the SSLServerAuth protocol should be used between A and B, another pair of channels with the same tags must be added and both pairs must be distinguished by different protoInstance tags, e.g. (protoInstance, 'Connection1') and (protoInstance, 'Connection2'). In addition to the structure, the behaviour of SSLClient and SSLServer must be specified. A complete model of the protocol component SSLServerAuth is described in Section 7.4.2.

The use of the annotations protocol, protoChannel and protoInstance involves the following constraints:

- The annotations protoChannel and protoInstance can only be used on a channel if the annotation protocol is present.
- If (protoChannel, n) $\in$ tags(ch) and (protocol, c) $\in$ tags(ch) for a channel ch, then the component c must have an input port p_i and an output port p_o with (protoChannel, n) $\in$ tags(p_i) and (protoChannel, n) $\in$ tags(p_o) and typeof(ch) = typeof(p_i) = typeof(p_o).

- There must not be two channels between the same components with the same protoInstance and protoChannel tag.
- In a protocol component, the annotation $(\mathsf{protoChannel}, n)$ must be attached to exactly one input and one output port.

4.2.6. Cryptography and Secrets

Cryptography and secrets are central mechanisms used in security-critical systems for the protection against attacks. As cryptography and secrets affect the data part of a system specification, we integrate their treatment into the functional language underlying our system model. For this purpose, we modify the predefined functions of particular hierarchical data types in order to reflect the properties of the respective cryptographic operations.

We specify cryptography within data type definitions by annotating certain constructors with special tags. Based on the annotations, we interpret a constructor application as a cryptographic operation if the constructor is annotated with a corresponding tag. In this work, we use the tag definitions Encr, Sign, Hash and Mac to model encryptions, signatures, hash computations or computations of message authentication codes.

For convenience, we write C_{td} for the set of constructors annotated with td (for $td \in \{\mathsf{Encr}, \mathsf{Sign}, \mathsf{Hash}, \mathsf{Mac}\}$). The sets $\mathsf{C}_{\mathsf{Encr}}$, $\mathsf{C}_{\mathsf{Sign}}$, $\mathsf{C}_{\mathsf{Hash}}$ and $\mathsf{C}_{\mathsf{Mac}}$ must be disjoint. In the concrete syntax, we assume that a constructor with a name containing Encr belongs to $\mathsf{C}_{\mathsf{Encr}}$, and constructors with names containing Sign, Hash or Mac belong to $\mathsf{C}_{\mathsf{Sign}}$, $\mathsf{C}_{\mathsf{Hash}}$ or $\mathsf{C}_{\mathsf{Mac}}$ respectively.

In the following, we first describe the meaning of the different annotations and then justify that the desired properties of the respective cryptographic operations are correctly reflected.

Encryptions

If $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}}$, the term $\mathsf{C}_i^k(t_1, t_2, \ldots, t_{n_i^k})$ (where $n_i^k > 1$) is supposed to represent the encryption of $(t_2, \ldots, t_{n_i^k})$ using t_1 as a key.

As usual in formal treatments of security-critical systems, we assume that cryptography is perfect. This means that it is impossible to extract the encryption key from an encrypted message and to decrypt the message without the correct decryption key. Modern cryptosystems can fulfil these assumptions with a high probability if they are used in the appropriate way and key lengths are chosen large enough to prevent guessing. We also assume that cryptograms can be compared (i.e., it can be verified whether a message has the same content and was encrypted with the same key as another message), and that successful decryption can be detected.

If other properties of cryptosystems need to be taken into account, the definitions given in this section must be modified or extended.

We incorporate these assumptions by adding the decryption key as another parameter to the selectors sel_{ij}^k for $j > 1$ and to the discriminator $\mathsf{is\_C}_i^k$. Intuitively, $\mathsf{sel}_{ij}^k(s,t)$ means "decrypt t using s as a key and extract the j-th argument if possible", and $\mathsf{is\_C}_i^k(s,t)$ means "check if t is an encrypted term of the form $\mathsf{C}_i^k(\dots)$ that can be decrypted with s". Besides, the selector sel_{i1}^k is omitted (so the key cannot be extracted from an encrypted message).

Signatures

If $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Sign}}$, the term $\mathsf{C}_i^k(t_1, t_2, \dots, t_{n_i^k})$ (where $n_i^k > 1$) is supposed to represent a signature with appendix of $(t_2, \dots, t_{n_i^k})$ using t_1 as a key, i.e. $t_2, \dots, t_{n_i^k}$ in clear together with an encrypted hash of $t_2, \dots, t_{n_i^k}$ using t_1 as a key.

We assume that it is impossible to extract the key from a signature, but that it can be verified if a signature was created with a particular key k using the inverse key. The data $t_2, \dots, t_{n_i^k}$ can be extracted from the signature (as it was transmitted in clear). Signatures cannot be forged, i.e. it is only possible to create a signature with knowledge of the corresponding signature key (here, t_1).

We incorporate these properties by adding the decryption key as another parameter to the discriminator $\mathsf{is\_C}_i^k$ as in the encryption case (here, for signature verification) and by omitting the selector sel_{i1}^k. The selectors sel_{ij}^k for $j > 1$ are defined as usual, for extraction of the data from the signature.

Hashes

If $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Hash}}$, the term $\mathsf{C}_i^k(t_1, \dots, t_{n_i^k})$ is supposed to represent a cryptographic hash of $(t_1, \dots, t_{n_i^k})$.

Cryptographic hashes must be one-way functions (i.e., it must be practically impossible to re-compute the hashed values from a hash). We assume perfect hash functions, and thus omit all selector and discriminator functions for C_i^k. Besides, it must be practically impossible to find a message M' with the same hash as a given message M (second preimage resistance), or a pair of messages (M, M') with the same hash (collision resistance). Hash functions that fulfil both properties are called "strong hash functions", hash functions that are only second preimage resistant are called "weak hash functions".

Message Authentication Codes

If $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Mac}}$, the term $\mathsf{C}_i^k(t_1, t_2, \dots, t_{n_i^k})$ is supposed to represent a message authentication code (MAC) of $(t_2, \dots, t_{n_i^k})$ using t_1 as a key.

We assume that it is only possible to create a MAC with knowledge of the key and the data to be authenticated and that it is impossible to retrieve the key or the data from a MAC. Thus, from the abstract viewpoint of the cryptographic extension to QuestF, message authentication codes have the same properties as hashes. In fact, one of the most secure MAC algorithms according to [FS03], the HMAC algorithm, is based on this principle — i.e., computing a hash of data concatenated with the key (HMAC uses double hashing to prevent more subtle attacks on the algorithmic level).

Predefined Functions for Cryptographic Terms

We assume that for each type type^k used as a encryption or signature key (i.e., as the first argument of a constructor $c \in \mathsf{C}_{\mathsf{Encr}} \cup \mathsf{C}_{\mathsf{Sign}}$), a function $\mathsf{inv}^k : (\mathsf{type}^k) \to \mathsf{type}^k$ has been defined mapping a key to its inverse. For any $x \in \mathsf{Value}(\mathsf{type}^k)$, $\mathsf{I}(\mathsf{inv}^k)(x) = x$ must hold if x is assumed to be a symmetric key, and $\mathsf{I}(\mathsf{inv}^k)(\mathsf{I}(\mathsf{inv}^k)(x)) = x$ must hold if x is assumed to be an asymmetric key. As with $==^k$ and if^k, we abbreviate $\mathsf{inv}^k(t)$ by $\mathsf{inv}(t)$ (where $\mathsf{typeof}(t) = \mathsf{type}^k$).

The predefined selector and discriminator functions given in Section 4.1.3 change as follows:

- The interpretation of the selector function sel^k_{i1} is not defined for $\mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Encr}} \cup \mathsf{C}_{\mathsf{Sign}}$, and the interpretations of the selector functions sel^k_{ij} are not defined for $C^k_i \in \mathsf{C}_{\mathsf{Hash}} \cup \mathsf{C}_{\mathsf{Mac}}$.
- The selector functions sel^k_{ij} for $j > 1, \mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Encr}}$ have the functionalities $\mathsf{fct}(\mathsf{sel}^k_{ij}) = (\mathsf{type}^k_{i1}, \mathsf{type}^k) \to \mathsf{type}^k_{ij}$, with $\mathsf{I}(\mathsf{sel}^k_{ij})(x, C^k_i(x_1, \ldots, x_{n^k_i})) = x_j$ if $\mathsf{eval}_\beta(x_1 == \mathsf{inv}(x)) = \mathsf{True}$ and $\perp$ otherwise. The selector functions sel^k_{ij} for $j > 1, \mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Sign}}$ are defined as in Section 4.1.3.
- The discriminator functions $\mathsf{is\_C}^k_i$ for $\mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Encr}} \cup \mathsf{C}_{\mathsf{Sign}}$ have the functionalities $\mathsf{fct}(\mathsf{is\_C}^k_i) = (\mathsf{type}^k_{i1}, \mathsf{type}^k)\ \mathsf{Bool}$, with $\mathsf{I}(\mathsf{is\_C}^k_i)(x, \mathsf{C}^k_i(x_1, \ldots, x_{n^k_i})) = \mathsf{True}$ if $\mathsf{eval}_\beta(x_1 == \mathsf{inv}(x)) = \mathsf{True}$, $\mathsf{I}(\mathsf{is\_C}^k_i)(\perp) = \perp$ and $\mathsf{I}(\mathsf{is\_C}^k_i)(x) = \mathsf{False}$ otherwise. For $\mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Hash}} \cup \mathsf{C}_{\mathsf{Mac}}$, no discriminator function $\mathsf{is\_C}^k_i$ is defined.

Example

Example 4.2.4. Figure 4.19 shows data type definitions for a message type to transmit encrypted signed data. Here, terms $\mathsf{PK}(x)$ and $\mathsf{SK}(x)$ of type TKey should represent the public and secret key of the agent x, respectively. We assume that inv is defined such that its interpretation is $\mathsf{I}(\mathsf{inv})(\mathsf{PK}(x)) = \mathsf{SK}(x)$ and $\mathsf{I}(\mathsf{inv})(x) = x$ otherwise.

Assume that t is a term of type $\mathsf{TMessage}$. Then,

```
data TAgent = A | B | I;
data TKey = PK(TAgent) |  SK(TAgent) | Data;

data TSig = Sign(TKey, getSigData: TKey);

data TMessage = EncrSig(TKey, getSig: TSig);
```

Figure 4.19.: DTD for Cryptographic Messages

- getSig(SK(B), t) can be used to extract the signature s from a term t of type TMessage, where t must be of the form t = EncrSig(PK(B), s) — here, t is interpreted as an encryption of s with the public key PK(B);

- is_Sign(PK(A), s) can be used to check if the signature s of type TSig has been signed with the secret key SK(A); and

- getSigData(s) is the signed data. In our example, the signed data is a value of type TKey.

Pattern Matching

Pattern matching can also be used for cryptographic data types, with only slight changes in the definitions of the pattern matching condition pmCond and the substitution pmSubst as given in Section 4.1.5

Assume $\mathsf{C}_i^k(t_1, t_2, \ldots, t_{n_i^k})$ is a pattern with $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}} \cup \mathsf{C}_{\mathsf{Sign}}$. In this case, we require t_1 not to contain variables (otherwise, the key could be extracted via matching) and use t_1 as the decryption/signature verification key in the matching condition. If the t_j for $j > 1$ contain variables, the appropriate selectors (with the key as a parameter in case of encryptions) must be used in the substitution.

If $\mathsf{C}_i^k(t_1, \ldots, t_{n_i^k})$ is a pattern with $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Hash}} \cup \mathsf{C}_{\mathsf{Mac}}$, none of the t_j must contain variables. For the matching condition, the equality operator can be used.

Thus, the definitions of pmCond and pmSubst change as follows:

$$\begin{array}{rcl}
\mathsf{pmCond}(p, \mathsf{C}_i^k(t_1, \ldots, t_{n_i^k})) & = & \{\mathsf{is\_C}_i^k(t_1, p)\} \cup \\
 & & \bigcup_{2 \le j \le n_i^k} \mathsf{pmCond}(\mathsf{sel}_{ij}^k(t_1, p), t_j), \text{ for } \mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}} \\
\mathsf{pmCond}(p, \mathsf{C}_i^k(t_1, \ldots, t_{n_i^k})) & = & \{\mathsf{is\_C}_i^k(t_1, p)\} \cup \\
 & & \bigcup_{2 \le j \le n_i^k} \mathsf{pmCond}(\mathsf{sel}_{ij}^k(p), t_j), \text{ for } \mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Sign}} \\
\mathsf{pmCond}(p, \mathsf{C}_i^k(t_1, \ldots, t_{n_i^k})) & = & (p == C_i^k(t_1, \ldots, t_{n_i^k})), \text{ for } \mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Hash}} \cup \mathsf{C}_{\mathsf{Mac}} \\
\mathsf{pmSubst}(p, \mathsf{C}_i^k(t_1, \ldots, t_{n_i^k})) & = & \mathsf{pmSubst}(\mathsf{sel}_{i2}^k(t_1, p), t_1) \circ \ldots \circ \\
 & & \mathsf{pmSubst}(\mathsf{sel}_{in_i^k}^k(t_1, p), t_{n_i^k}), \text{ for } \mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}}
\end{array}$$

Example 4.2.5. If $t = \mathsf{EncrSig}(\mathsf{SK}(\mathsf{B}), x)$ is a pattern of the type $\mathsf{TMessage}$ defined in Figure 4.19, then we have $\mathsf{pmCond}(p, t) = \{\mathsf{is\_EncrSig}(\mathsf{SK}(\mathsf{B}), p)\}$ and $\mathsf{pmSubst}(p, t) = [x/\mathsf{getSig}(\mathsf{SK}(\mathsf{B}), p)]$. The term $p = \mathsf{EncrSig}(\mathsf{PK}(\mathsf{B}), \mathsf{Sign}(\mathsf{SK}(\mathsf{A}), \mathsf{Data}))$ matches t, and x is substituted by $\mathsf{Sign}(\mathsf{SK}(\mathsf{A}), \mathsf{Data})$ in the matching.

Translation to Standard AutoFocus/Quest Models

To be able to use the existing simulation, verification and test sequence generation functionality of the tool, we provided a translation from models using the cryptographic extensions of the functional language QuestF to models using only the standard features of QuestF.

The basic idea of this translation is to add function definitions for the selectors and discriminators whose functionality and semantics has been changed and to replace the actual selectors and discriminators by the newly defined functions.

For example, the following function definition would be generated for a discriminator belonging to a constructor in $\mathsf{C}_{\mathsf{Encr}}$:

$$\mathsf{fun}\ \mathsf{\_is\_C}_i^k(k, t) = \mathsf{is\_C}_i^k(t)\ \&\&\ (\mathsf{sel}_{i1}^k(t) == \mathsf{inv}(k));$$

In the translation, all occurrences of $\mathsf{is\_C}_i^k$ in the model are replaced by $\mathsf{\_is\_C}_i^k$. Besides, pattern matching is eliminated by adding pmCond to the precondition of a transition and applying the substitution $\mathsf{pmSubst}$ to the precondition, the output patterns and the actions.

With this translation, in the following we can work with the extended functional language without the necessity to make any further changes to the tool.

Secrets and Intruder Knowledge

When modelling security-critical systems, often some data is assumed to be guessable by the intruder (for example, agent names), whereas other data is not (for example, keys or secure random numbers). We need to reflect this in our model to be able to determine which data the intruder can generate.

For this purpose, we annotate types whose values are assumed to be non-guessable with Key. The only way an intruder can produce a value of such a type is if it is part of his initial knowledge, or if he learned it by analysing messages he received. We write $\mathsf{T}_{\mathsf{Key}}$ for the set of types annotated with Key. In the concrete syntax, a type with a name containing Key is assumed to belong to $\mathsf{T}_{\mathsf{Key}}$. To allow the intruder to generate messages with empty/illegal values in place of value of a non-guessable type, we require that a type $\mathsf{type}^k \in \mathsf{T}_{\mathsf{Key}}$ has a constructor Empty^k, i.e. $\mathsf{Empty}^k \in \mathsf{constructors}(d)$ for the data type definition d with $\mathsf{defType}(d) = \mathsf{type}^k$. Keys are always modelled symbolically and not as integer values, i.e. $\mathsf{Int} \notin \mathsf{T}_{\mathsf{Key}}$. The mapping of symbolically modelled keys to concrete byte sequences is described in Section 6.4.

We model the initial intruder knowledge by a set of functions $\mathsf{knowsIntruder}^k : (\mathsf{type}^k) \rightarrow \mathsf{Bool}$ for each $\mathsf{type}^k \in \mathsf{T}_{\mathsf{Key}}$. The intruder is assumed to initially know a key $x \in \mathsf{Value}(\mathsf{type}^k)$ if $\mathsf{I}(\mathsf{knowsIntruder}^k)(x) = \mathsf{True}$. As usual, we abbreviate $\mathsf{knowsIntruder}^k(t)$ by $\mathsf{knowsIntruder}(t)$ for $\mathsf{typeof}(t) = \mathsf{type}^k$.

Derivable Values

Definition 4.2.6. Let $H \subseteq \mathsf{Value}$ be a set of values. We define a set $\mathsf{derivable}(H)$, the set of values that can be generated by an intruder with the knowledge of H, as follows:

$$\mathsf{derivable}(H) = \{\mathsf{eval}_\beta(t) : t \in \mathsf{Term} \wedge \mathsf{nosecrets}(t) \wedge \forall v \in \mathsf{freeVar}(t) : \beta(v) \in H\}$$

The terms t must not contain selectors, constructors or discriminators of the types in $\mathsf{T}_{\mathsf{Key}}$, with the exception of the Empty^k constructors. This is specified by $\mathsf{nosecrets}(t)$:

$$\begin{aligned}\mathsf{nosecrets}(t) \;\; = \;\; & \forall t' : t' \trianglelefteq t \;\wedge\; t' \in \mathsf{Appl} \;\wedge\; \mathsf{typeof}(t') \in \mathsf{T}_{\mathsf{Key}} \wedge \\ & \quad \exists d \in \mathsf{DataDef} : \mathsf{defType}(d) = \mathsf{typeof}(t') \Rightarrow \\ & \quad\quad \forall c \in \mathsf{constructors}(d) \setminus \{\mathsf{Empty}^k\} : \\ & \quad\quad\quad \mathsf{head}(t') \neq c \;\wedge\; \mathsf{head}(t') \neq \mathsf{is\_}c \;\wedge \\ & \quad\quad\quad\quad \forall s \in \mathsf{selectors}(c) : \; \mathsf{head}(t') \neq s\end{aligned}$$

Thus, $\mathsf{derivable}(H)$ is the set of values obtained by evaluating any possible terms that refer to the elements of H, but that do not contain functions belonging to non-guessable types.

Example 4.2.7. Consider again the data type definitions depicted in Section 4.19. Assume that the initial intruder knowledge is $H = \{\mathsf{SK}(\mathsf{I})\} \cup \{\mathsf{PK}(x) : x \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}\}$. Then,
$\mathsf{derivable}(H) = H \cup \{\bot, \mathsf{A}, \mathsf{B}, \mathsf{C}\} \cup S \cup M$, where
$S = \{\mathsf{Sign}(x, y) : x, y \in H\}$ and $M = \{\mathsf{EncrSig}(x, y) : x \in H \wedge y \in S\}$.

Formal Justification

We formally justify that our integration of cryptography into the QuestF language is sound, in the sense that it fulfils the desired abstract properties stated above. This prevents the possibility that a specification contains unrealistic behaviour which cannot be implemented. We focus on abstract terms representing encryptions. In particular, we show that an argument cannot be extracted from an encryption without knowledge of the key.

Definition 4.2.8. Let $t \in \mathsf{Term} \cup \{\bot\}$, $s \in \mathsf{Term}$ and $x \in \mathsf{Value}$. We define a predicate $\mathsf{encapsulated}(t,s,x)$ stating that x appears in t only as a subterm of s and that for all subterms t' of t with $\mathsf{head}(t') = \mathsf{head}(x)$ which are outside of s it must be the case that $t' \in \mathsf{Value}$:

$$\begin{array}{lcl}
\mathsf{encapsulated}(\bot,s,x) & = & \mathsf{True}\\
\mathsf{encapsulated}(v,s,x) & = & \mathsf{True}, \text{ for } v \in \mathsf{Var}\\
\mathsf{encapsulated}(s,s,x) & = & \mathsf{True}\\
\mathsf{encapsulated}(f(t_1,\ldots,t_n) \neq s,s,x) & = & (\mathsf{head}(x) = f \Rightarrow f(t_1,\ldots,t_n) \in \mathsf{Value}) \wedge\\
 & & f(t_1,\ldots,t_n) \neq x \wedge\\
 & & \mathsf{encapsulated}(t_1,s,x) \wedge \ldots \wedge\\
 & & \mathsf{encapsulated}(t_n,s,x)
\end{array}$$

In the following, we interpret x as a value we want to extract from an encryption s using the term t. $\mathsf{encapsulated}(t,s,x)$ ensures that x cannot be produced without accessing its occurrence in s. In particular, by the condition on the subterms t' with $\mathsf{head}(t') = \mathsf{head}(x)$, it is ruled out that x is produced outside of s by applying the constructor of x to appropriately computed arguments: t' must be already in normal form and is not further evaluated.

Lemma 4.2.9. *If $x \neq s$ and $\mathsf{encapsulated}(t,s,x)$, then $t \neq x$.*

Lemma 4.2.10. *If $\mathsf{encapsulated}(f(t_1,\ldots,t_n),s,x)$ and $f(t_1,\ldots,t_n) \neq s$, then we have $\mathsf{encapsulated}(t_i,s,x)$ for all $i \in \{1,\ldots,n\}$.*

Proof Lemmas 4.2.9 and 4.2.10 follow directly from the definition of encapsulated. □

Theorem 4.2.11. *Let $s = \mathsf{C}_i^k(x_1,x_2,\ldots,x_{n_i^k}) \in \mathsf{Value}$ be an encryption, i.e. $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}}$, t a term, β a valuation, $j \geq 2$ and x_1^{-1} be an abbreviation for the inverse key $\mathsf{eval}_\beta(\mathsf{inv}^k(x_1))$. Suppose that $\mathsf{encapsulated}(t,s,x_1^{-1})$ and $\mathsf{encapsulated}(t,s,x_j)$, and likewise $\mathsf{encapsulated}(\beta(v),s,x_1^{-1})$ and $\mathsf{encapsulated}(\beta(v),s,x_j)$ for all $v \in \mathsf{Var}$ and $\mathsf{encapsulated}(t^f,s,x_1^{-1})$ and $\mathsf{encapsulated}(t^f,s,x_j)$ for all right hand sides of function definitions for functions f. Then $\mathsf{eval}_\beta(t) \neq x_j$.*

This means that it is impossible to extract x_j from an encryption s without the correct decryption key x_1^{-1} using any term t, provided that x_j does not appear outside of the encryption.

In addition, if we assume instead $\mathsf{encapsulated}(t, s, x_1)$ *and* $\mathsf{encapsulated}(\beta(v), s, x_1)$ *for all* $v \in \mathsf{freeVar}(t)$ *and* $\mathsf{encapsulated}(t^f, s, x_1)$ *for all right hand sides of function definitions for functions f, then* $\mathsf{eval}_\beta(t) \neq x_1$*. I.e., it is impossible to extract the encryption key x_1 from an encrypted message s.*

Proof To prove the first part of the above statement, we prove the stronger consequence that $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$ hold, by structural induction on t. $\mathsf{eval}_\beta(t) \neq x_j$ follows by Lemma 4.2.9.

Suppose t is a variable. Then, $\mathsf{eval}_\beta(t) = \beta(t)$, and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$ because of the assumption on β.

Suppose $t = s$. Then, $\mathsf{eval}_\beta(t) = s$, as $s \in \mathsf{Value}$, and thus we can conclude that $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$ hold, because of the definition of $\mathsf{encapsulated}$.

Suppose $\mathsf{head}(t) \in \{==, \&\&, ||, =>, \mathsf{not}, \mathsf{is\_C}_{i'}^{k'}\}$. Then, $\mathsf{eval}_\beta(t)$ is either True, False or $\perp$. $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$ are true in all those cases.

Suppose $t = \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3\ \mathsf{fi}$. If $\mathsf{eval}_\beta(t_1) = \perp$, $\mathsf{eval}_\beta(t) = \perp$. Otherwise, without loss of generality, we assume that $\mathsf{eval}_\beta(t_1) = \mathsf{True}$. By Lemma 4.2.10, we have $\mathsf{encapsulated}(t_2, s, x_1^{-1})$ and $\mathsf{encapsulated}(t_2, s, x_j)$. Thus, by the induction hypothesis $\mathsf{encapsulated}(\mathsf{eval}_\beta(t_2), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t_2), s, x_j)$ hold, and thus $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$, because $\mathsf{eval}_\beta(t) = \mathsf{eval}_\beta(t_2)$.

Suppose $t = \mathsf{C}_{i'}^{k'}(t_1, \ldots, t_{n_{i'}^{k'}}) \neq s$. As $t \neq s$, by Lemma 4.2.10 and the induction hypothesis we have $\mathsf{encapsulated}(\mathsf{eval}_\beta(t_l), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t_l), s, x_j)$ for $l \in \{1, \ldots, n_{i'}^{k'}\}$. If $\mathsf{head}(x_j) \neq \mathsf{C}_{i'}^{k'}$, then $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$ must hold, because of the definitions of $\mathsf{C}_{i'}^{k'}$ and $\mathsf{encapsulated}$. Otherwise, it must be the case that $t \in \mathsf{Value}$ (because $\mathsf{encapsulated}(t, s, x_j)$). Thus, $\mathsf{eval}_\beta(t) = t$ and therefore $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$. In the same way, $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_1^{-1})$ can be derived.

Suppose $t = \mathsf{sel}_{i'j'}^{k'}(t_1)$ with $\mathsf{C}_{i'}^{k'} \notin \mathsf{C}_{\mathsf{Encr}}$ (i.e., $\mathsf{sel}_{i'j'}^{k'}$ is an "ordinary" selector that does not belong to a cryptographic constructor). Again we can conclude that $\mathsf{encapsulated}(\mathsf{eval}_\beta(t_1), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t_1), s, x_j)$. If we require that $\mathsf{eval}_\beta(t)$ should not be $\perp$, $\mathsf{eval}_\beta(t_1)$ must be of the form $\mathsf{C}_{i'}^{k'}(y_1, \ldots, y_{n_i^k})$. Besides, as $\mathsf{C}_{i'}^{k'} \notin \mathsf{C}_{\mathsf{Encr}}$, $\mathsf{eval}_\beta(t_1) \neq s$. Thus, $\mathsf{encapsulated}(y_l, s, x_1^{-1})$ and $\mathsf{encapsulated}(y_l, s, x_j)$ (by Lemma 4.2.10). As $\mathsf{eval}_\beta(t) = y_j$, we conclude that $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$.

Suppose $t = \mathsf{sel}_{i'j'}^{k'}(t_1, t_2)$ with $\mathsf{C}_{i'}^{k'} \in \mathsf{C}_{\mathsf{Encr}}$. If we require that $\mathsf{eval}_\beta(t)$ should not be $\bot$, $\mathsf{eval}_\beta(t_2)$ must be of the form $\mathsf{C}_{i'}^{k'}(y_1, \ldots, y_{n_i^k})$ and $\mathsf{eval}_\beta(y_1 == \mathsf{inv}(t_1)) = \mathsf{True}$ (cf. definition of cryptographic selectors). If $\mathsf{eval}_\beta(t_2) \neq s$ we can continue as in the above case for $\mathsf{C}_{i'}^{k'} \notin \mathsf{C}_{\mathsf{Encr}}$. On the other hand, we can conclude that $\mathsf{eval}_\beta(t_1) \neq x_1^{-1}$ (by the induction hypothesis). Because of the properties of inv, it follows that $\mathsf{eval}_\beta(x_1 == \mathsf{inv}(t_1)) == \mathsf{False}$, and thus, if $\mathsf{eval}_\beta(t_2) = s$, $\mathsf{eval}_\beta(y_1 == \mathsf{inv}(t_1)) == \mathsf{False}$ (because $y_1 = x_1$) and $\mathsf{eval}_\beta(t) = \bot$.

Finally, suppose that $t = f(t_1, \ldots, t_n)$, with f being a user-defined function. From the assumption, we have $\mathsf{encapsulated}(\mathsf{eval}_\beta(t_l), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t_l), s, x_j)$. Let β' be the valuation $[v_1 \leftarrow \mathsf{eval}_\beta(t_1), \ldots, v_n \leftarrow \mathsf{eval}_\beta(t_n)]$. β' and t^f (the right hand side of the function definition for f) fulfil the assumptions in Theorem 4.2.11 and $\mathsf{eval}_\beta(t) = \mathsf{eval}_{\beta'}(t^f)$. Thus, $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_1^{-1})$ and $\mathsf{encapsulated}(\mathsf{eval}_\beta(t), s, x_j)$.

As we required that each function symbol must be either predefined or defined via a function definition, there are no other cases. The second part of Theorem 4.2.11 can be proved in a similar way. □

The other properties of encryption mentioned above are fairly straightforward. Cryptograms can be compared via the == operator, and successful decryption can be detected via the discriminator function $\mathsf{is\_C}_\mathsf{k}^\mathsf{i}$ that evaluates to True if applied to an encrypted term with the correct key.

Note that the assumption on x_j is not fulfilled if $\mathsf{typeof}(x_j) = \mathsf{Int}$, because the constructors Zero and Succ appear in the function definitions for functions on integers. This reflects the fact that in the model, in contrast to symbolically represented data, integers can always be "guessed".

Theorem 4.2.12. *Let $H \subseteq \mathsf{Value}$ be a set of values, $s = \mathsf{C}_i^k(x_1, x_2, \ldots, x_{n_i^k}) \in H$ be an encryption, i.e. $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}}$, $j \geq 2$ and x_1^{-1} be an abbreviation for the inverse key $\mathsf{eval}_\beta(\mathsf{inv}^k(x_1))$. Suppose that $\mathsf{typeof}(x_1) \in \mathsf{T}_{\mathsf{Key}}$, $\mathsf{typeof}(x_j) \in \mathsf{T}_{\mathsf{Key}}$, $x_1 \neq \mathsf{Empty}^k$, $x_j \neq \mathsf{Empty}^k$, and $\mathsf{encapsulated}(t^f, s, x_1^{-1})$ and $\mathsf{encapsulated}(t^f, s, x_j)$ for all right hand sides of function definitions for functions f. Also suppose that there is no $x \in H \setminus \{s\}$ with $x_1^{-1} \trianglelefteq x$ or $x_j \trianglelefteq x$. Then, $x_j \notin \mathsf{derivable}(H)$.*

I.e., from a set H containing an encryption, one cannot derive an argument of the encryption, provided that the argument or the key are not guessable and cannot be derived from other values in H or from function definitions.

Proof From the assumptions in Theorem 4.2.12, we conclude $\mathsf{encapsulated}(t, s, x_1^{-1})$ and $\mathsf{encapsulated}(t, s, x_j)$ for the terms t appearing in the definition of $\mathsf{derivable}$, because constructors of types in $\mathsf{T}_{\mathsf{Key}}$ cannot occur in t. Besides, $\mathsf{encapsulated}(\beta(v), s, x_1^{-1})$ and $\mathsf{encapsulated}(\beta(v), s, x_j)$ hold for all $v \in \mathsf{Var}$: either because $\beta(v) = s$ or because x_1^{-1} and x_j do not occur in $H \setminus \{s\}$. Therefore, Theorem 4.2.11 can be applied, and thus $\mathsf{eval}_\beta(t) \neq x_j$. Therefore, $x_j \notin \mathsf{derivable}(H)$ because of the definition of $\mathsf{derivable}$. □

Theorem 4.2.13. *Let* $s = \mathsf{C}_i^k(x_1, x_2, \ldots, x_{n_i^k}) \in \mathsf{Value}$ *be a signature, i.e.* $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Sign}}$*,* t *a term,* β *a valuation such that* $\mathsf{encapsulated}(t, s, x_1)$ *and* $\mathsf{encapsulated}(\beta(v), s, x_1)$ *for all* $v \in \mathsf{Var}$ *and* $\mathsf{encapsulated}(t^f, s, x_1)$ *for all right hand sides of function definitions for functions* f*. Then* $\mathsf{eval}_\beta(t) \neq x_1$ *(it is impossible to extract the signature key from a signature).*

Theorem 4.2.14. *Let* $s = \mathsf{C}_i^k(x_1, x_2, \ldots, x_{n_i^k}) \in \mathsf{Value}$ *be a hash or MAC, i.e.* $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Hash}} \cup \mathsf{C}_{\mathsf{Mac}}$*,* t *a term,* β *a valuation and* $j \in \{1, \ldots, n_i^k\}$ *such that* $\mathsf{encapsulated}(t, s, x_j)$ *and* $\mathsf{encapsulated}(\beta(v), s, x_j)$ *for all* $v \in \mathsf{Var}$ *and* $\mathsf{encapsulated}(t^f, s, x_j)$ *for all right hand sides of function definitions for functions* f*. Then* $\mathsf{eval}_\beta(t) \neq x_j$ *(it is impossible to extract the data from a hash, respectively the data and the key from a MAC).*

The proofs of Theorem 4.2.13 and Theorem 4.2.14 are similar to the proof of Theorem 4.2.11. Besides, analogous theorems as Theorem 4.2.12 can be formulated to provide a connection to the definition of derivable. The fact that a signature can only be verified using the discriminator $\mathsf{is\_C}_i^k$ together with the correct inverse key follows directly from the defined semantics of $\mathsf{is\_C}_i^k$. The only way to create a new term of the form $\mathsf{C}_i^k(t_1, \ldots, t_{n_i^k})$ with $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Sign}}$ (a signature) is by application of the constructor C_i^k, as selectors can only extract existing arguments from a constructor term. In the application of C_i^k, the key t_1 must be provided as the first argument. Therefore it is not possible to create a signature without knowledge of the key. Finally, $\mathsf{C}_i^k(t_1, \ldots, t_{n_i^k}) == \mathsf{C}_i^k(t'_1, \ldots, t'_{n_i^k})$ generally only evaluates to True if $\forall i : \mathsf{eval}_\beta(t_i) = \mathsf{eval}_\beta(t'_i)$. Thus, in the particular case that $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Hash}} \cup \mathsf{C}_{\mathsf{Mac}}$, there are no two different pairs $(t_1, \ldots, t_{n_i^k})$, $(t'_1, \ldots, t'_{n_i^k})$ of messages with the same hash (or MAC, in which case t_1 is interpreted as the key).

4.3. Case Study I: A Bank Application

AutoFocus/Quest was used in a number of case studies to model, verify and test security-critical systems. In this section, we present an AutoFocus/Quest model of a part of an application for managing electronic order forms, developed during a research project in cooperation with a major German bank. This case study will serve as a running example for verification using threat scenario generation and for the treatment of layered protocols. It was published in [GHJW03, Grü03]. The version presented in this thesis was adapted to make use of the AutoFocus/Quest extensions for cryptographic data types, threat specification, protocol layering, and the application of automatic threat scenario generation.

Initial Model The basic structure of the system is shown in Figure 4.20, as an AutoFocus/Quest SSD. The system can be divided into three parts: the client's

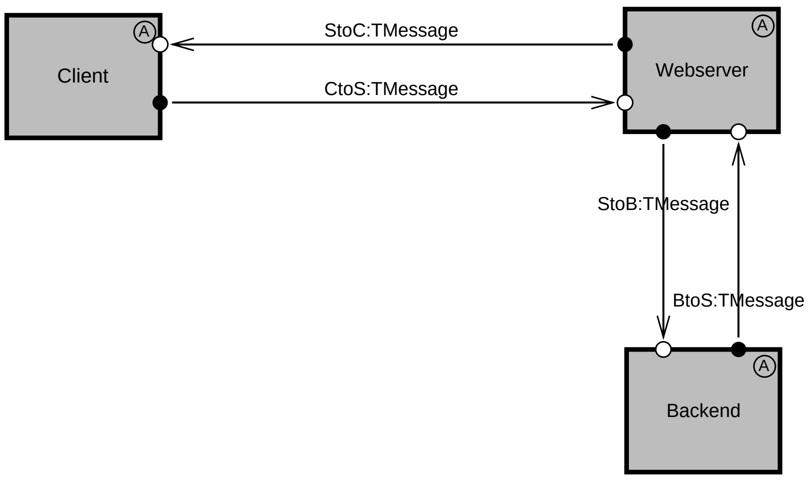


Figure 4.20.: Bank Application: System Structure Diagram

computer with a Web browser and a smart card to securely generate electronic signatures (component Client in the model), the Web server / application server authenticating and handling the client's requests (component Webserver), and a back-end system with data bases where the actual forms and customer data are stored (component Backend).

Obtaining and signing electronic order forms is carried out as follows. The client connects to the Web server using her browser. After she has been authenticated successfully, she can request an order form (e.g. for an interest report), which is sent to her pre-filled with her data from the back-end system. The client digitally signs the document using her smart card and sends it back to the server. After verification of the signature, the server sends an acknowledgement and the connection is closed.

The EET in Figure 4.21 shows the messages exchanged during such a transaction, in terms of the abstract AutoFocus/Quest model. To keep verification feasible, data such as keys, identification numbers and messages containing these elements was modelled symbolically using hierarchical data types, as introduced in Section 4.2.6. The corresponding data type definitions are depicted in Figure 4.22. For testing purposes, the abstract messages (data type TMessage) must be mapped to implementation-level messages. We will present an approach for this in Section 6.4. Finally, the STDs specifying the behaviour of the components are shown in Figure 4.23.

In the function definitions, as the name of a function inv^k we take the name of the type type^k prefixed with "inv". For convenience of notation, we make use of alternatives and pattern matching. Such function definitions can be translated to function definitions without alternatives and pattern matching, as described in

Section 4.1.3, with the help of chains of applications of if ... then ... else ... fi. For example, the definition of invTAKey in Figure 4.22 can also be written as

```
fun invTAKey(x) = if is_PK(x) then SK(PKSel1(x)) else
                     if is_SK(x) then PK(SKSel1(x)) else
                        x
                     fi
                  fi;
```

Here, PKSel1 and SKSel1 are the default names automatically generated for the selectors of PK and SK.

Likewise, we write Empty^k as the name of type^k prefixed with "Empty". For instance, EmptyTAKey stands for an empty/illegal value for an asymmetric key.

Flow of a Transaction In the model, the Client component begins the transaction by sending a ClientHello message to the Webserver component. On reception of the message, the Webserver chooses a nonce NonceS, stores it in its local variable localNonceServer, and sends a message NonceID(NonceS) to the client. The client sends back a message Data(Sign(SK(C), NonceS), SignCert(SK(CA), C, GIDC, PK(C))) containing the received nonce signed with her public key SK(C) and her certificate, signed by the certification authority CA. The certificate contains the client's identity C, a global identification number GIDC which references her data on the backend system and her public key. The Webserver checks the certificate and the signature of the nonce. If the checks have been successful, the client is authenticated (transition from state NonceSent to state ClientAuthenticated in the corresponding STD). Note that here we use the cryptographic versions of the discriminators for signature verification, e.g. is_Sign(PK(CA), cert) for verifying the certificate cert using the certification authority's public key PK(CA).

The Webserver then stores the global ID and sends it together with an empty form to the Backend (message DataFormGID(EmptyTForm, GIDC)). Here, the global ID is stored as well and the form is filled with the client's data CDataBC. The filled form is then sent back via the Webserver to the client (message DataForm(Form(CDataBC))). The Client signs the form with the private key SK(C)(message SignCData(SK(C), CDataBC), SignCert(SK(CA), C, GIDC, PK(C)))). This message is forwarded by the Webserver to the Backend. The Backend checks the signature of the received data object and the certificate. The received global ID and the signed data object are compared with the ones stored. On success, an order is generated, an acknowledgement is sent to the client (message Form(AcknowledgementC)), and the transaction is finished.

Note that for simplicity, the model consists of just one honest client, specifies only one execution of a transaction and does not include error handling. The model can be easily extended to reflect more complex scenarios.

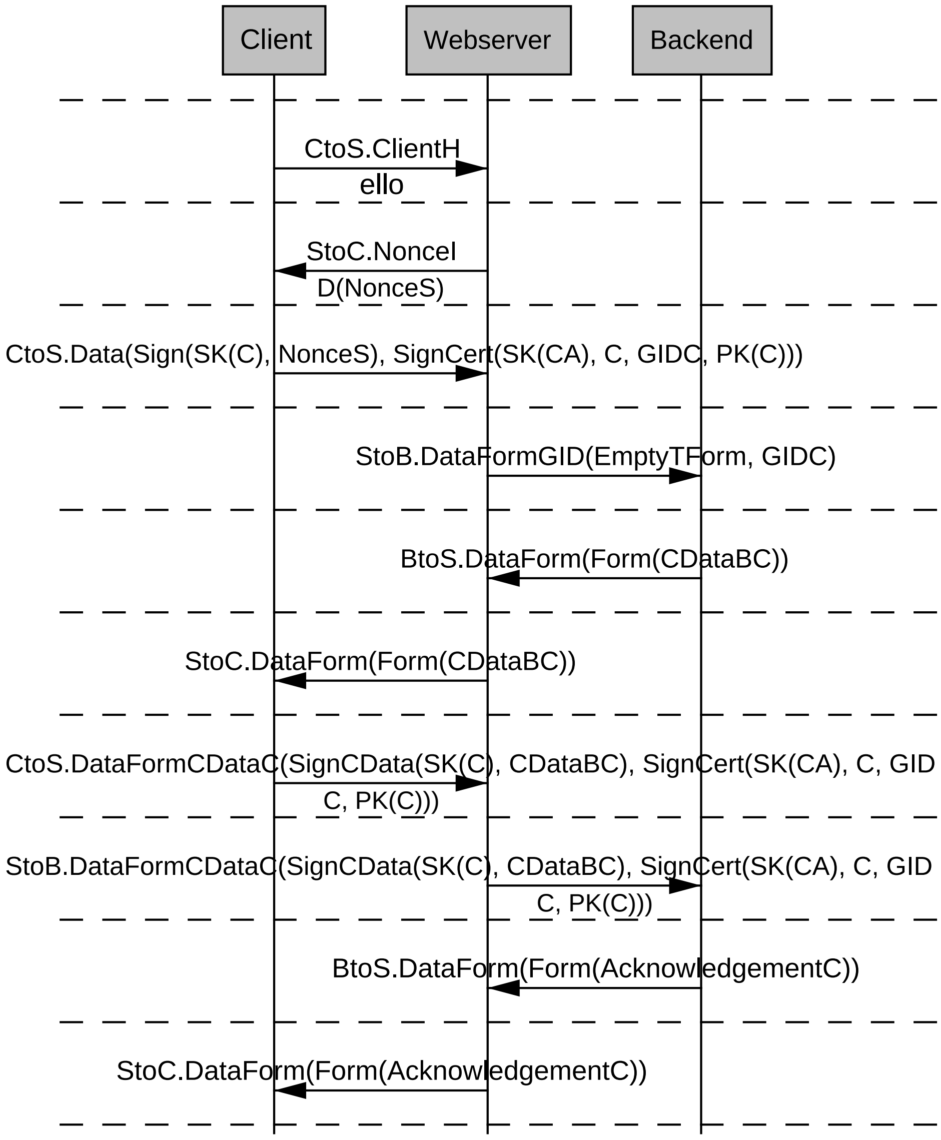
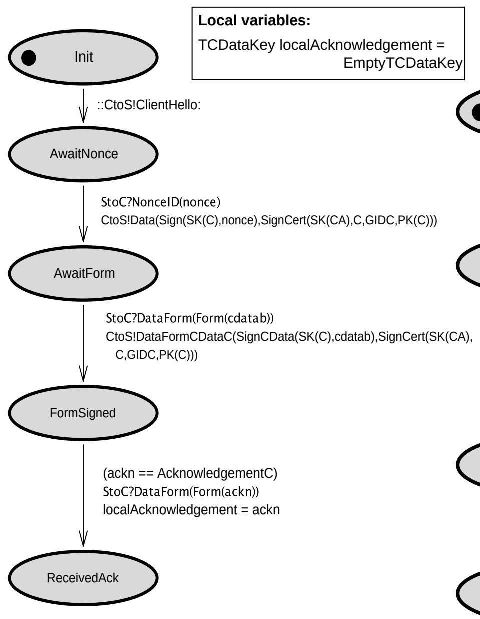

Client
Webserver
Backend
CtoS.ClientH
ello
StoC.NonceI
D(NonceS)
CtoS.Data(Sign(SK(C), NonceS), SignCert(SK(CA), C, GIDC, PK(C)))
StoB.DataFormGID(EmptyTForm, GIDC)
BtoS.DataForm(Form(CDataBC))
StoC.DataForm(Form(CDataBC))
CtoS.DataFormCDataC(SignCData(SK(C), CDataBC), SignCert(SK(CA), C, GID
C, PK(C)))
StoB.DataFormCDataC(SignCData(SK(C), CDataBC), SignCert(SK(CA), C, GID
C, PK(C)))
BtoS.DataForm(Form(AcknowledgementC))
StoC.DataForm(Form(AcknowledgementC))

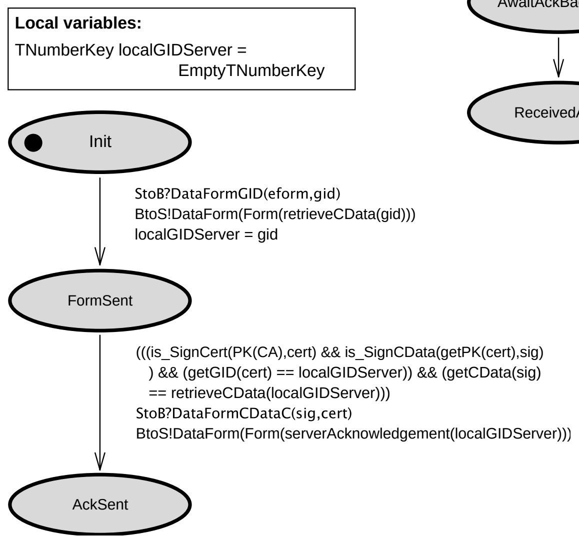
Figure 4.21.: Bank Application: Extended Event Trace

```
/* type definition for asymmetric key:
   Client, Server, CA, Adversary */

data TAgent = C | S | A | CA;
data TAKey = PK(TAgent) | SK(TAgent) | EmptyTAKey;
fun invTAKey(PK(x)) = SK(x)
    | invTAKey(SK(x)) = PK(x)
    | invTAKey(x) = x;

/* message type */
data TMessage = EmptyTMessage
                | ClientHello
                | NonceID(TNumberKey)
                | Data(TSign, TSignCert)
                | DataForm(TForm)
                | DataFormGID(TForm, TNumberKey)
                | DataFormCDataC(TSignCData, TSignCert);

/* signed nonce / certificate */
data TSign = Sign(TAKey, getNonce: TNumberKey);
data TSignCert = SignCert(TAKey, getName: TAgent,
                          getGID: TNumberKey,
                          getPK: TAKey);

/* form with client data */
data TForm = EmptyTForm | Form(TCDataKey);

/* signed client data / client data */
data TSignCData = SignCData(TAKey, getCData: TCDataKey);
data TCDataKey = CDataBC | CDataBA | AcknowledgementC |
                 AcknowledgementA | EmptyTCDataKey;

/* secret data (nonces / global IDs) */
data TNumberKey = EmptyTNumberKey | GIDC | GIDA | NonceS | NonceA;

/* backend: retrieve client data from GID /
   determine acknowledgement */

fun retrieveCData(GID) = CDataBC
    | retrieveCData(GIDA) = CDataBA
    | retrieveCData(n) = EmptyTCDataKey;
fun serverAcknowledgement(GIDC) = AcknowledgementC
    | serverAcknowledgement(GIDA) = AcknowledgementA
    | serverAcknowledgement(n:TNumber) = EmptyTCDataKey;
```

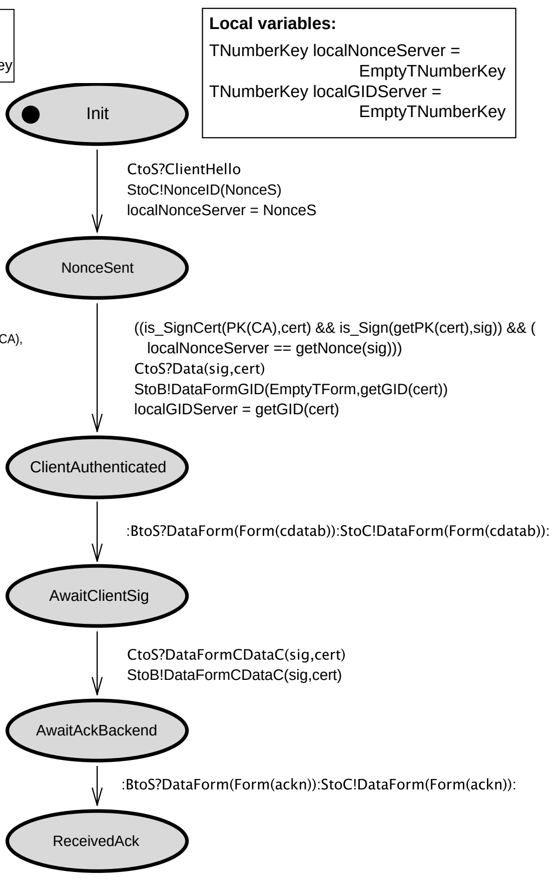
Figure 4.22.: Bank Application: Data Type Definition


Local variables:
TCDataKey localAcknowledgement = EmptyTCDataKey
Init
::CtoS!ClientHello:
AwaitNonce
StoC?NonceID(nonce)
CtoS!Data(Sign(SK(C),nonce),SignCert(SK(CA),C,GIDC,PK(C)))
AwaitForm
StoC?DataForm(Form(cdatab))
CtoS!DataFormCDataC(SignCData(SK(C),cdatab),SignCert(SK(CA), C,GIDC,PK(C)))
FormSigned
(ackn == AcknowledgementC)
StoC?DataForm(Form(ackn))
localAcknowledgement = ackn
ReceivedAck

(a) STD Client


Local variables:
TNumberKey localGIDServer = EmptyTNumberKey
Init
StoB?DataFormGID(eform,gid)
BtoS!DataForm(Form(retrieveCData(gid)))
localGIDServer = gid
FormSent
(((is_SignCert(PK(CA),cert) && is_SignCData(getPK(cert),sig)) && (getGID(cert) == localGIDServer)) && (getCData(sig) == retrieveCData(localGIDServer)))
StoB?DataFormCDataC(sig,cert)
BtoS!DataForm(Form(serverAcknowledgement(localGIDServer)))
AckSent

(b) STD Backend


Local variables:
TNumberKey localNonceServer = EmptyTNumberKey
TNumberKey localGIDServer = EmptyTNumberKey
Init
CtoS?ClientHello
StoC!NonceID(NonceS)
localNonceServer = NonceS
NonceSent
((is_SignCert(PK(CA),cert) && is_Sign(getPK(cert),sig)) && (localNonceServer == getNonce(sig)))
CtoS?Data(sig,cert)
StoB!DataFormGID(EmptyTForm,getGID(cert))
localGIDServer = getGID(cert)
ClientAuthenticated
:BtoS?DataForm(Form(cdatab)):StoC!DataForm(Form(cdatab)):
AwaitClientSig
CtoS?DataFormCDataC(sig,cert)
StoB!DataFormCDataC(sig,cert)
AwaitAckBackend
:BtoS?DataForm(Form(ackn)):StoC!DataForm(Form(ackn)):
ReceivedAck

(c) STD Webserver

Figure 4.23.: Bank Application: State Transition Diagrams

Security Requirements The two main security requirements taken from an informal security analysis of a system are:

- **Client data confidentiality** The personal data in the forms must be kept confidential:

$$\mathsf{SR.CONF\_CLDATA} \equiv \mathsf{never(learnedIntruder(CDataBC))}$$

  This property states that the client's data CDataBC must be kept confidential. There are similar security requirements for other pieces of data involved in the transaction, such as the global ID GIDC.

- **Authenticated orders** Orders cannot be submitted in the name of others:

$$\begin{aligned}&\mathsf{SR.AUTH\_ORDERS} \equiv\\&\quad \mathsf{precedes(State_{Client} == AwaitForm,}\\&\qquad \mathsf{(State_{Webserver} == ClientAuthenticated)\&\&}\\&\qquad \mathsf{(Webserver.localGID == GIDC))}\end{aligned}$$

  This property states that the Webserver can only reach the ClientAuthenticated state at which the global ID GIDC will be sent to the backend (and therefore allow an order in the name of C to be submitted) if the honest component Client has issued a request.

Threats and Assumptions The transaction protocol relies on the connection between Client and Webserver being confidential, integrity protected and authenticated on the server side. The connection between Webserver and Backend is assumed to be secure. We also assume that the internals of the components cannot be accessed or manipulated by the intruder and therefore annotate them with node. Thus, the remaining threat is the replacement of the Client by an adversary, specified by the annotation replace. Figure 4.24 shows the corresponding security-enriched model. To analyse instead the security of the system assuming an unprotected connection between Client and Webserver, one would have to annotate the two respective channels with public and omit the replace annotation. To additionally consider an unprotected connection between Webserver and Backend, public annotations would have to be placed at the channels between these components as well.

In practice, confidentiality, integrity protection, and authentication of the server are achieved by sending the messages between Client and Webserver over a previously established SSL connection. We will deal with the relationship between our abstract threat specification and the use of a lower level protocol to achieve the implied assumptions in Section 7.4.

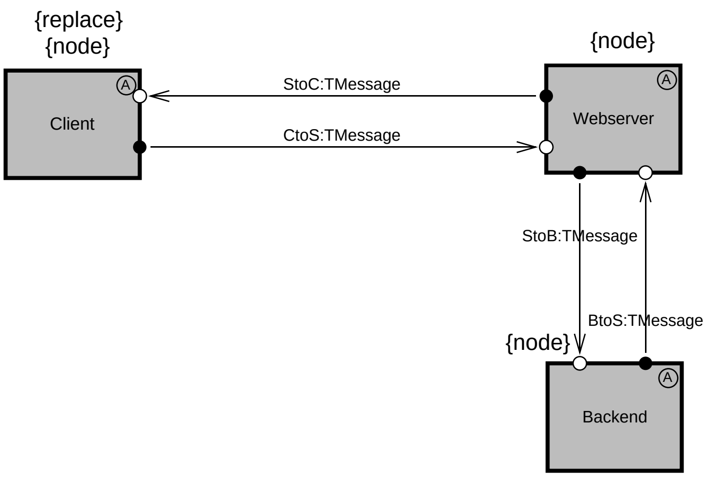


Figure 4.24.: Bank Application: Security-Enriched Model (SSD)

Functionality that is not relevant for the security of the transaction is not included in the model. Therefore, all states and transitions of the STDs Client, Webserver and Backend are assumed to be annotated with critical.

Finally, we have specified that customer data, nonces, global IDs, and asymmetric keys are secret data not guessable by the intruder, by using names for the corresponding data types which contain "Key", i.e. TAKey, TCDataKey and TNumberKey. The intruder is assumed to initially know all public keys and his own secret key, his own certificate, his own global ID and a nonce NonceA that is different from the nonce of the honest Client component. This is specified by appropriate definitions of the functions knowsIntruder^k, as follows:

```
fun knowsIntruderTAKey(PK(x)) = True
  | knowsIntruderTAKey(SK(A)) = True
  | knowsIntruderTAKey(x) = False;

fun knowsIntruderTNumberKey(GIDA) = True
  | knowsIntruderTNumberKey(NonceA) = True
  | knowsIntruderTNumberKey(x) = False;

fun knowsIntruderTSignCert(
          SignCert(SK(CA),A,GIDA,PK(A))) = True
  | knowsIntruderTSignCert(x) = False;
```

Here, as with inv^k, we use the name of type^k prefixed with "knowsIntruder" as the name of a function knowsIntruder^k.

Simulation, Verification and Test Case Generation The security-enriched model of the bank application can be simulated and its functional correctness can be checked using the built-in verification features of AutoFocus/Quest. In this stage, the security annotations are treated like comments. The EET in Figure 4.21 originated as the result of a simulation run.

For security verification and test case generation, the model must be transformed into a threat scenario. We will explain this in further detail in Section 5.4.1.

4.4. Case Study II: The Common Electronic Purse Specifications

The Common Electronic Purse Specifications (CEPS) [CEP01] define requirements for a globally interoperable electronic purse scheme providing accountability and auditability. CEPS is supported by organisations (including Visa International) representing 90 percent of the world's electronic purse cards and is likely to become an accepted standard [AJSW00] .

The CEPS electronic purse scheme is based on smart cards, on which account balances are stored in different currencies, corresponding to electronic value the cardholder can use for payments. The smart cards take part in transactions that are protected by cryptographic means, most importantly to ensure that the account balances can only be updated in a consistent way.

In the following, we consider the two main parts of CEPS: the load transaction and the purchase transaction. In addition, CEPS defines an unload transaction (to remove electronic value from a CEP card), a currency exchange transaction, and a cancel last purchase transaction.

4.4.1. CEPS Load Transaction

The (unlinked, cash-based) load transaction allows the cardholder to load electronic value onto a card in exchange for cash at a load device belonging to a load acquirer. The participants involved in the transaction protocol are the customer's card, the load device and the card issuer. The load device contains a Load Security Application Module (LSAM) that is used to store and process transaction data. During the transaction, the account balance in the card is incremented and the amount is sent to the issuer for later financial settlement. In addition, each participant writes information about the transaction to secure log files. We concentrate on vulnerabilities arising from possible failures during one protocol execution. Therefore, to increase readability and ease verification and test case generation, protocol details that aim to prevent replay attacks (such as transaction date and time, transaction numbers and identifiers) were abstracted away. Besides, we omitted mechanisms for

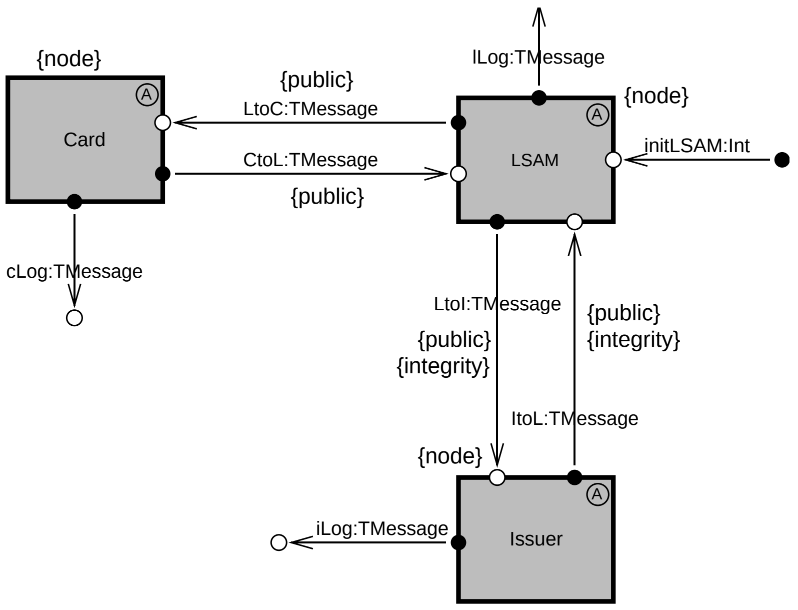


Figure 4.25.: CEPS Load Transaction: Security-Enriched Model (SSD)

integrity protection of messages between the issuer and the load device and instead included integrity as an assumption on the connection.

Figure 4.25 shows the SSD of a security-enriched AutoFocus/Quest model of the CEPS load transaction, consisting of three components Card, LSAM and Issuer. The public tags indicate the channels that can be attacked (all channels between the components, but not the channels used to write log information). The channels between LSAM and Issuer are additionally assumed to be integrity protected. The components themselves are assumed to be protected from manipulation, denoted by the security tag node: in the implementation, the card and the security module must ensure tamper resistance, and the issuer system must be adequately protected from external attacks. The messages in CEPS are protected by message authentication codes based on secret keys and by random nonces. We assume the attacker initially knows his own secret keys and random nonces, but not those of Card, Issuer and LSAM.

The CEPS load protocol works roughly as follows: first, the user inserts her card into the load device and requests to load a particular amount m onto the card by inserting cash. The LSAM is initialised with this amount: in the model, m is sent to the LSAM on channel initLSAM. The LSAM then retrieves the current balance from the card (message InquireCardInfo) and sends to it an initialisation message Init(m). The card's response RespC to the LSAM contains a message authentication code MacS1, which is forwarded to the issuer together with data securing the

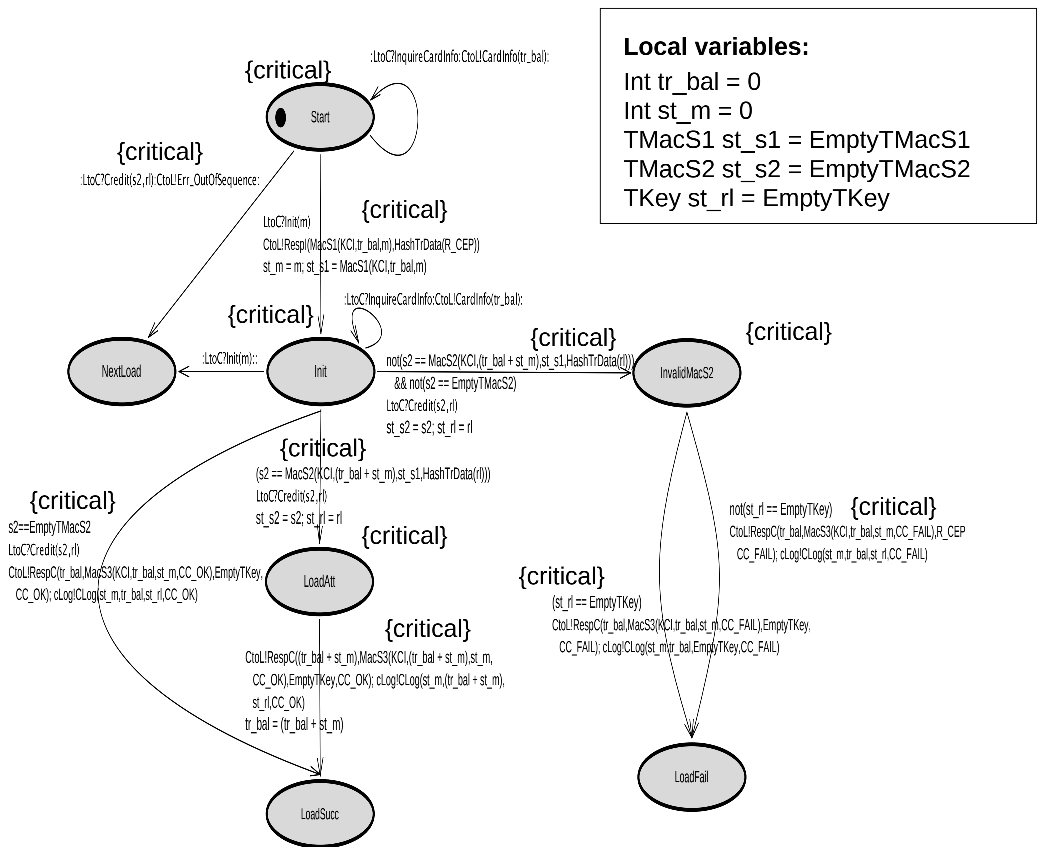

Local variables:
Int tr_bal = 0
Int st_m = 0
TMacS1 st_s1 = EmptyTMacS1
TMacS2 st_s2 = EmptyTMacS2
TKey st_rl = EmptyTKey
{critical}
Start
:LtoC?InquireCardInfo:CtoL!CardInfo(tr_bal):
{critical}
:LtoC?Credit(s2,rl):CtoL!Err_OutOfSequence:
{critical}
LtoC?Init(m)
CtoL!RespI(MacS1(KCI,tr_bal,m),HashTrData(R_CEP))
st_m = m; st_s1 = MacS1(KCI,tr_bal,m)
{critical}
:LtoC?InquireCardInfo:CtoL!CardInfo(tr_bal):
NextLoad
:LtoC?Init(m)::
Init
{critical}
not(s2 == MacS2(KCI,(tr_bal + st_m),st_s1,HashTrData(rl)))
&& not(s2 == EmptyTMacS2)
LtoC?Credit(s2,rl)
st_s2 = s2; st_rl = rl
InvalidMacS2
{critical}
{critical}
(s2 == MacS2(KCI,(tr_bal + st_m),st_s1,HashTrData(rl)))
LtoC?Credit(s2,rl)
st_s2 = s2; st_rl = rl
{critical}
s2==EmptyTMacS2
LtoC?Credit(s2,rl)
CtoL!RespC(tr_bal,MacS3(KCI,tr_bal,st_m,CC_OK),EmptyTKey,
CC_OK); cLog!CLog(st_m,tr_bal,st_rl,CC_OK)
{critical}
LoadAtt
not(st_rl == EmptyTKey)
{critical}
CtoL!RespC(tr_bal,MacS3(KCI,tr_bal,st_m,CC_FAIL),R_CEP
CC_FAIL); cLog!CLog(st_m,tr_bal,st_rl,CC_FAIL)
{critical}
(st_rl == EmptyTKey)
CtoL!RespC(tr_bal,MacS3(KCI,tr_bal,st_m,CC_FAIL),EmptyTKey,
CC_FAIL); cLog!CLog(st_m,tr_bal,EmptyTKey,CC_FAIL)
{critical}
CtoL!RespC((tr_bal + st_m),MacS3(KCI,(tr_bal + st_m),st_m,
CC_OK),EmptyTKey,CC_OK); cLog!CLog(st_m,(tr_bal + st_m),
st_rl,CC_OK)
tr_bal = (tr_bal + st_m)
LoadFail
LoadSucc

Figure 4.26.: CEPS Load Transaction: Security-Enriched Model (STD Card)

issuer's decisions. On successful verification of this data, the issuer sends a MAC MacS2 to the LSAM, indicating his decision to allow the load transaction. The LSAM forwards MacS2 to the card (Credit message), which verifies it, adjusts the balance and replies with a MAC MacS3. MacS3 is forwarded to the issuer informing him about the result of the transaction. At the end of a transaction, all parties write entries to their log files and stop in a state LoadSucc if the transaction was successfully completed or in a state LoadFail if an error was detected.

Figure 4.26 shows the behavioural specification of the Card component together with security annotations indicating critical states and transitions. In the Start state, the Card component is ready to answer InquireCardInfo and Init messages. If a Credit message is received, it reports an error (Err_OutOfSequence) and stops its execution in state NextLoad. As a reply RespI to an Init message, the card sends the MAC MacS1 together with a hash HashTrData(R_CEP) of transaction data including a random number R_CEP shared between the card and the issuer.

The MAC MacS1 and the transaction amount are stored in local variables and the card changes its state to Init. In the Init state, the reception of a second Init message causes the card to stop its execution in state NextLoad (as the model describes the processing of just a single transaction). If a correct MAC MacS2 is received as part of the Credit message also containing a random value rl from the LSAM, the card adjusts its balance, replies with a corresponding MAC MacS3 (message RespC) and writes a log entry with condition code CC_OK (transitions Init–LoadAtt–LoadSucc). If an empty MAC EmptyTMacS2 is received, no updates are performed and again success is reported (this is used to end a transaction if the load acquirer cannot obtain a MacS2 from the issuer; transition Init–LoadSucc). If the S2 MAC does not verify correctly, the card reports an error (condition code CC_FAIL; transitions Init–InvalidMacS2–LoadFail). Besides, it outputs R_CEP if it received the random value rl from the LSAM in the Credit message . The purpose of outputting R_CEP is to provide the LSAM with proof of a valid error from the card if the LSAM authorised the load by releasing rl.

The focus of the model is on the security-related behaviour. Thus, most of the states and transitions correspond to receiving, verifying or creating cryptographic messages or writing secure logs. On the other hand, the part of the STD where the InquireCardInfo message is processed is not regarded as security-critical. It is just provided for information purposes and not protected from manipulation (a possible manipulation during a load transaction would be dealt with in the following steps). The transition from state Init to state NextLoad corresponding to the reception of a second Init message is not considered security-critical as well, as it belongs to a second load transaction which is out of the scope of the model. If the model were more concrete, the proportion of security-critical states and transitions would be smaller. However, in general it is a good idea to structure a system in a way that security-critical functionality is concentrated in few components, so a high proportion of security-critical transactions and states in an STD is an indication of an adequate model. The other components should then contain fewer security-critical states and transitions and therefore require less verification and testing effort.

The main security objective of CEPS is resistance to fraud between the participating parties. We give three examples for security requirements and formalise them with respect to our model.

Firstly, we require that the random value R_CEP received by the LSAM from the card is indeed a proof of a failed transaction. I.e., if the LSAM received a load approval from the issuer ($\mathsf{State}_{\mathsf{LSAM}} == \mathsf{LoadApproved}$) and receives in the RespC message from the card a value for R_CEP (retrieved using the selector getRespCR_CEP) whose hash is the same as the hash the LSAM received previously (stored in a local variable st_hc), then the card must have reported an error to its log (condition code getCLCC(cLog) not equal to CC_OK).

This corresponds to the following security requirement as an LTL formula, using the property pattern precedes (**non-repudiation of failed transaction at card**):

$$\begin{array}{l}\mathsf{SR.NONREP\_FAILED\_TRANS} \equiv \\ \quad \mathsf{precedes(is\_Msg(cLog)} \,\&\&\, \mathsf{not(getCLCC(cLog)} == \mathsf{CC\_OK)}, \\ \qquad \mathsf{State_{LSAM}} == \mathsf{LoadApproved} \,\&\& \\ \qquad \mathsf{is\_Msg(LSAM.CtoL)} \,\&\&\, \mathsf{is\_RespC(LSAM.CtoL)} \,\&\& \\ \qquad \mathsf{not(getRespCR\_CEP(LSAM.CtoL)} == \mathsf{EmptyTKey)} \&\& \\ \qquad \mathsf{HashTrData(getRespCR\_CEP(LSAM.CtoL))} == \mathsf{LSAM.st\_hc)}\end{array}$$

Secondly, we require that if the card experiences an error, indeed a correct value for R_CEP which matches the hash received by the LSAM is sent to the load acquirer. However, this is only necessary if the LSAM received a load approval ($\mathsf{State_{LSAM}} == \mathsf{LoadApproved}$) and a secret rl was included in the Credit message that when released by the LSAM indicates the commitment to perform the load. In this case, rl has been stored in the local variable Card.st_rl of the card. Thus, the corresponding security requirement is: (**fair exchange from the load acquirer's point of view**)

$$\begin{array}{l}\mathsf{SR.FAIREXC\_LACQ} \equiv \\ \quad \mathsf{leadsto(State_{LSAM}} == \mathsf{LoadApproved} \,\&\& \\ \qquad \mathsf{not(Card.st\_rl} == \mathsf{EmptyTKey)} \,\&\& \\ \qquad \mathsf{is\_Msg(cLog)} \,\&\&\, \mathsf{not(getCLCC(cLog)} == \mathsf{CC\_OK)}, \\ \qquad \mathsf{is\_Msg(Card.CtoL)} \,\&\&\, \mathsf{is\_RespC(Card.CtoL)} \,\&\& \\ \qquad \mathsf{not(getRespCR\_CEP(Card.CtoL)} == \mathsf{EmptyTKey)} \,\&\& \\ \qquad \mathsf{HashTrData(getRespCR\_CEP(Card.CtoL))} == \mathsf{LSAM.st\_hc)}\end{array}$$

Thirdly, we require that when a transaction was successfully completed from the point of view of the card with an amount $m > 0$ (reflected by a corresponding log entry and a non-empty MacS2 stored in st_s2), the issuer must have authorised the loading of the card with m. Thus, the cardholder can use the log entry together with the stored MacS2 to prove the authorisation.

I.e., for all $m > 0$, the following LTL formula must hold (**non-repudiation of load authorisation**):

$$\begin{array}{l}\mathsf{SR.NONREP\_LOAD\_AUTH} \equiv \\ \quad \mathsf{precedes(State_{Issuer}} == \mathsf{Load} \,\&\&\, \mathsf{Issuer.tr\_m} == m, \\ \qquad \mathsf{is\_Msg(Card.cLog)} \,\&\&\, \mathsf{getCLM(Card.cLog)} == m \,\&\& \\ \qquad \mathsf{getCLCC(Card.cLog)} == \mathsf{CC\_OK} \,\&\& \\ \qquad \mathsf{not(Card.st\_s2} == \mathsf{EmptyTMacS2))}\end{array}$$

Here, Load is the control state of the issuer that is reached when the transaction was authorised, and the local variable Issuer.tr_m stores the corresponding amount.

The results of the verification of the above security requirements with respect to the threat scenario resulting from the security-enriched model are described Section 5.4.2. Besides, the CEPS Load transaction served as the main case study for model-based test sequence generation and concretisation. For more detail, see Section 6.5.

4.4.2. CEPS Purchase Transaction

The purchase transaction allows the cardholder to use the electronic value on a card to pay for goods. The participants involved in the transaction protocol are the customer's card and the merchant's point-of-sale (POS) device. The purchase transaction is an off-line protocol, as the issuer is not involved. The POS device contains a Purchase Security Application Module (PSAM) that is used to store and process transaction data and that is assumed to be tamper resistant. During the transaction, the account balance in the card is decremented and the balance in the PSAM is incremented by the corresponding amount. The card issuer later receives transaction logs.

Figure 4.27 shows the SSD of a security-enriched model of the CEPS purchase transaction. Here, tamper resistance of the card and the PSAM is indicated by the node tags. This scenario will serve us as an example for the consideration of complex threats that can be analysed by manually editing the generated threat scenario – see Section 5.4.2 for more detail. In the security-enriched model, it is only specified that the card and the PSAM can be replaced by an intruder.

In a similar way as in the load transaction, during the purchase transaction protected transaction messages are exchanged between the PSAM and the card to authenticate each other and authorise withdrawal and credit of the amount to be transferred. However, in the purchase transaction, public key cryptography is used, as there are no shared secrets between the card and the PSAM. As a security requirement, we considered correct authentication between the PSAM and the card in presence of an intruder.

More detail on modelling the CEPS Purchase transaction in AutoFocus/Quest can be found in [JW01b]. The generation and customisation of the threat scenario and the results of the security analysis are described in Section 5.4.2 (based on a model adapted from [JW01b] to make use of the modelling concepts described in this thesis).

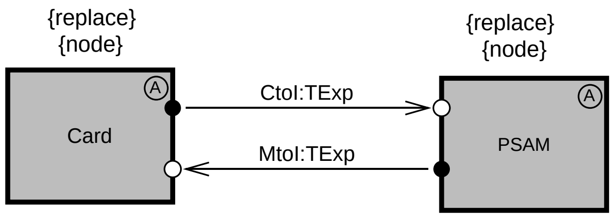


Figure 4.27.: CEPS Purchase Transaction: Security-Enriched Model (SSD)

4.5. Related Work

AutoFocus and Model Transformations

Descriptions of the AutoFocus/Quest tool and its meta model in the form of UML diagrams can also be found in [Sch01, Löt03, Bra03]. [Löt03] describes the semantics of a simplified version of AutoFocus/Quest in the form of composed state machines. A formal semantics for the QuestF language based on its abstract syntax is not given in these works and the semantics of LTL properties with respect to AutoFocus/Quest models is not considered. The semantics of AutoFocus/Quest models described in Section 4.1.5 is based on [WLPS00], which was derived from [HSE97, PS99]. [Bra03] describes an object-oriented transformation language called BOTL, which is also applied to the transformation of AutoFocus/Quest models. [SPHP02] contains simple examples for model transformations such as "pulling up" a subcomponent out of the component it is contained in. [BHS99] gives a general overview over AutoFocus/Quest and the concepts behind the tool.

Extended Description Techniques for Security Aspects

An early stage of the work on security extensions for the AutoFocus/Quest description techniques presented in Section 4.2 is reported in [WW01]. [Jür04], developed in parallel to this work, describes the UMLsec profile, an extension to the UML allowing to integrate threats and assumptions into UML models. The UMLsec approach differs from the approach described in Section 4.2 in that threats on two layers are considered (the deployment level is represented by deployment diagrams with stereotypes such as «`internet`» or «`wire`» and the logical level is represented by static structure diagrams with stereotypes such as «`secrecy`» or «`integrity`»), connected by consistency conditions. In AutoFocus/Quest, the same description technique (SSDs) is used for both purposes. Besides, security requirements are modelled as tags belonging to fixed stereotypes (e.g. a tag $\{\text{secrecy} = X\}$ belonging to a stereotype «`data security`») rather than general LTL properties with patterns. The use of general LTL properties with patterns allows more flexibility in specifying security requirements, while at the same time aiding the designer in their correct formalisation. The replace threat is not included in UMLsec. In this work, we chose AutoFocus/Quest as a basis because of its conceptual simplicity, its clear semantics, its adequacy for the considered application domain and its extensive tool connections. The examination of translations between the two modelling approaches (in particular, from UMLsec to AutoFocus for threat scenario generation, verification, mechanism application and testing) is subject of future work.

[LBD02, BDL03] describe an extension of the UML to specify (role based) access control requirements and how they can be transformed to access control mechanisms on the deployment platform. [PJWB03, Pop05] deal with a methodology for use case based development of security-critical systems and shows how to extend use cases with the specification of security aspects. [FH97] presents general extensions of use cases for the elaboration of non-functional requirements, including security.

An example for the inclusion of cryptographic mechanisms into a formal specification language is the Spi calculus [AG99], an extension of the Pi calculus. In [GL97], encryption and signatures are represented in a specification language for abstract data types (ACT ONE), without extending the language itself. The authors remark that care must be taken to avoid the specification of unrealistic behaviour (such as extracting data from an encrypted message without knowledge of the key) by not obeying the implicit rules for the use of the abstract data types for encryptions or signatures. Our extension concept prevents this problem. The soundness of symbolic models of cryptography with respect to complexity-theoretic or probabilistic models is examined in [AJ01, PSW00].

4.6. Summary and Discussion

We introduced AutoFocus/Quest, a **generic formal specification language** based on communicating state machines with tool support for simulation, verification, test case generation and code generation. We described the syntax of AutoFocus/Quest and its underlying functional language QuestF in the form of an **integrated meta model** consisting of entity sets and functions specified by UML class diagrams. Based on this meta model, we gave an **operational semantics** of AutoFocus/Quest and described a formalism for **model transformations**. The choice of AutoFocus/Quest was motivated by its appropriateness for the considered application domain (systems where transactions are processed, in particular in electronic business) and by its conceptual simplicity and clear semantics.

To be able to include additional information into a model, the used model-based specification language must be extensible. We presented a simple, generic **extension mechanism** for the AutoFocus/Quest specification language that provides the possibility to add annotations to model elements. The more complex extension mechanism provided by the UML proved not to be necessary for our purposes, as we did not need to define new model elements. An important feature of our extension is that data type definitions in the underlying functional language can be extended as well, because the definition of the abstract syntax of the functional language is part of the meta model.

We showed how to explicitly **integrate security aspects** relevant at different stages of the development into a model-based specification language, such that security-related activities can be defined that take into account the additional information in a **security-enriched model**. In particular, we gave security extensions for AutoFocus/Quest that allow

- to specify **global security requirements** in the form of property patterns in temporal logic,
- to specify **threats and assumptions** on components and channels,
- to specify **security mechanisms**, both on an abstract level by specifying for instance that particular channels are protected by given lower-level security protocols, and on a more concrete level by the integration of cryptographic operations and secret data elements into the functional language QuestF.

The extensions cover all information necessary for the security-related activities described in this thesis. Our approach is distinguished in particular by the use of property patterns for common security requirements, the provision of security annotations on transitions and states of state machines to be able to guide security testing, the features for the model-based specification of layered security protocols, and the sound treatment of cryptography within a general-purpose functional language by the specification of additional attributes for user-defined data types. The usual cryptographic interpretation of data types in models using a generic specification language is error-prone, because it is possible to specify unrealistic behaviour, such as the extraction of data from an encrypted message without knowledge of the key. We formally justified that our extension avoids this problem. Besides, our extension also allows specifications using pattern matching on cryptographic terms.

The resulting security-enriched models are rather intuitive, and it proved to be fairly straightforward to add security-related information to existing AutoFocus/Quest models. Often, we already detected errors simply because of the need to precisely specify the system and the security-related aspects, before any verification was performed at all. Besides, our approach makes it possible to take full advantage of existing (non-security specific) tool support. This strengthens our assumption that the development of security-critical systems can be aided by the described model-based concepts with low additional effort.

The presented security extensions are aimed at basic security aspects relevant for transaction-processing communicating systems, providing guidelines to a developer performing a security analysis. We do not claim to give a complete set of extensions for every possible security aspect (which would be infeasible). If additional or more complex classes of security requirements, threats, assumptions or mechanisms occur within the development of a system, new annotations can be defined in an analogous way to the ones given. For certain security aspects, it also suggests itself to extend the meta model, for instance by roles and permissions to specify access control properties. In our approach, more complex threats and assumptions can also be dealt with by modifying the generated threat scenario (see Chapter 5).

We demonstrated the application of the security extensions at the example of two **case studies**, a bank application and the Common Electronic Purse Specifications. The security extensions described in this chapter proved to be appropriate for the treatment of the considered systems. Further case studies are described in Section 5.4.

5. Threat Scenario Generation and Verification

As pointed out in Section 2.1, security must be provided in presence of a malicious environment from which attempts can be performed to access or manipulate a system in an unauthorised way. Therefore, models of security-critical systems must be verified and their implementations must be tested relative to the malicious environment. We refer to a model of a system in a malicious environment as a *threat scenario*. In this chapter, we show how to generate threat scenarios from security-enriched models and how to make use of threat scenarios as a basis for the verification of security requirements.

This chapter is structured as follows. In Section 5.1, we describe in more detail the concepts of threat scenarios and threat scenario generation and their role in our model-based design process for security-critical systems. In Section 5.2, we show how a generic threat scenario can be derived from a security-enriched AutoFocus/Quest model, by introducing an intruder. To keep it general, we specify the behaviour of the generic intruder on the semantical level, as a discrete system. In Section 5.3, we give an example for a concrete behavioural specification of a threat scenario as a system of AutoFocus/Quest state machines (STDs), tailored for model checking. The resulting threat scenario generated in this way is again an AutoFocus/Quest model, thus it can be verified for security violations using the connection to the symbolic model checker SMV available in AutoFocus/Quest. To enable model checking, the capabilities of the intruder in this specialised intruder model have to be restricted. We show that the specialised intruder model is sound (i.e., does not allow more attacks than the generic intruder model), and give a class of systems and properties for which it is complete (i.e., all violations of the properties with respect to the generic intruder model are found using the specialised intruder model). We show optimisations of the intruder model that enable more efficient model checking. The generated threat scenario can be manually edited to analyse more complex threats than can be specified in the security-enriched model.

In Section 5.4, we demonstrate threat scenario generation and verification at the example of a number of case studies, including the bank application case study presented in Section 4.3. We give references to related work in Section 5.5 and summarise the presented results in Section 5.6.

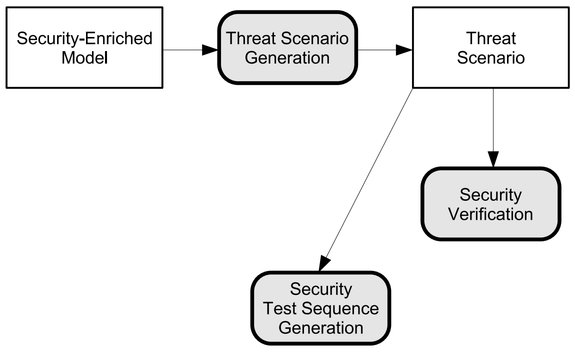


Figure 5.1.: Threat Scenario Generation

5.1. Threat Scenarios

The result of a security analysis is the identification of security requirements, critical entities in a system, threats which these entities are exposed to, and assumptions that can be taken for granted. Security verification and testing must be carried out under consideration of this security-specific information: the aim of security verification and testing is to show that the security requirements hold for a system *under attack*, i.e. when the identified threats are realised (under the given assumptions). In our model-based approach, the security-specific information is available as part of the security-enriched model.

A *threat scenario* makes the threats explicit. In the context of a threat-scenario based development approach using the formal language Focus, Lotz defines the notion of a threat scenario as follows [Lot97]:

> A threat scenario is a modification of the system specification that describes a situation in which the system is attacked by an adversary, according to the result of threat identification and risk analysis.

In our model-based approach, threat scenarios are generated from security-enriched models based on the additional information available: in other words, *threat scenario generation* is a model transformation which, applied on a security-enriched model, results in a threat scenario (see Figure 5.1). The threat scenario is an additional model that is used specifically for security verification and testing. For this purpose, one can then take advantage of available verification and test generation support of the used specification language.

If tool supported, threat scenario generation can proceed automatically. The generated threat scenario can be manually edited to analyse custom threats not directly supported by the annotations in the security-enriched model.

5.2. Generic Threat Scenario

In the following, we explain the derivation of a threat scenario from a security-enriched model, at the example of AutoFocus/Quest. Both structural and behavioural aspects have to be considered: the structural view of the threat scenario describes *where* an intruder can interfere with the system, and the behavioural view describes *how* he can do so. We first specify the behavioural view on the semantical level, as a *generic intruder model*.

5.2.1. Structural View

The structure of the threat scenario is determined by the annotations that specify general threats, listed under the "threat" role in Table 4.1 on p. 53: channels annotated with public can be accessed and manipulated by the intruder, and components annotated with public or replace can be replaced by the intruder.

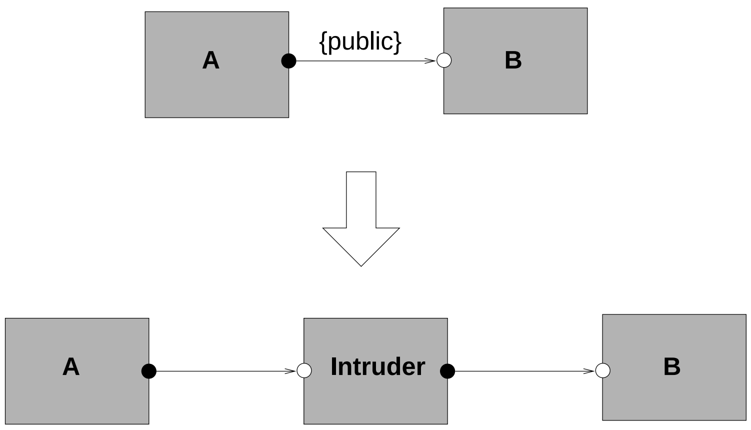
In the following, for simplicity we assume that the security-enriched model has a non-hierarchical structure, i.e. it consists of one top-level component with a number of subcomponents that are not further decomposed. The threat scenario generation that was implemented into AutoFocus/Quest can be applied to hierarchical models as well and preserves their hierarchical structure.

We model the interference of the intruder by adding a new component Intruder to the model and changing the connection structure depending on the specified threats.

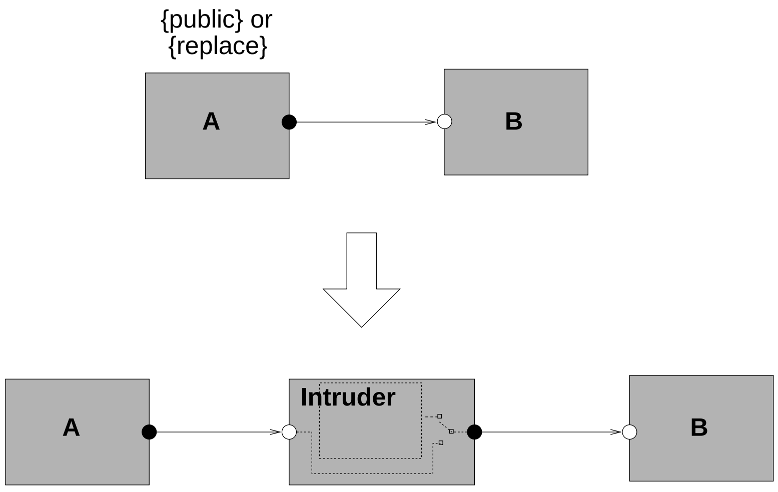
Threats on Channels Channels annotated with public are split into two parts leading to and originating from the intruder (see Figure 5.2(a)).

Threats on Components If a component c is annotated with public or replace, it can be replaced by an intruder. This means that for the threat scenario, we have to consider two cases: either c communicates with the components connected to it, or the intruder communicates with them in place of c. To be able to analyse both cases at the same time, we insert the Intruder component into all channels leading to c or originating from c (see Figure 5.2(b)). The behaviour of the Intruder component must then be restricted in such a way that it either forwards all communication from/to c without eavesdropping or manipulation, or cuts off c completely. This is hinted by the dashed lines in the figure (which have no semantics but are just there to clarify the idea).

If there is more than one channel or component that can be attacked, we connect them to the *same* Intruder component. This reflects the most general attack: the system is accessed and manipulated at several different points at the same time in



(a) Threat Scenario for public Channel



(b) Threat Scenario for public or replace Component

Figure 5.2.: Threat Scenarios: Structural View

a coordinated way. The amount of coordination can later be weakened by tailoring the behavioural specification of the $\mathsf{Intruder}$ component.

In the following, we give a formal description of the model transformation resulting in the structural view of the generic threat scenario. Let $\mathcal{M}$ be the security-enriched model with top-level component Main and $\mathcal{M}'$ the corresponding threat scenario. We define the set $\mathsf{PublicChannel}(\mathcal{M})$ of channels of $\mathcal{M}$ annotated with public and the set $\mathsf{ReplaceChannel}(\mathcal{M})$ of channels of $\mathcal{M}$ that are connected to components annotated with public or $\mathsf{replace}$ as follows:

$$\begin{array}{lcl}
\mathsf{PublicChannel}(\mathcal{M}) & = & \{ch \in \mathsf{channels}^{\mathcal{M}}(\mathsf{Main}) : \mathsf{public} \in \mathsf{tags}^{\mathcal{M}}(ch)\} \\
\mathsf{ReplaceChannel}(\mathcal{M}) & = & \{ch \in \mathsf{channels}^{\mathcal{M}}(\mathsf{Main}) : \exists c \in \mathsf{Component}^{\mathcal{M}} : \\
 & & \quad (\mathsf{destP}^{\mathcal{M}}(ch) \in \mathsf{inPorts}^{\mathcal{M}}(c) \vee \\
 & & \qquad \mathsf{sourceP}^{\mathcal{M}}(ch) \in \mathsf{outPorts}^{\mathcal{M}}(c)) \wedge \\
 & & \quad (\mathsf{public} \in \mathsf{tags}^{\mathcal{M}}(c) \vee \mathsf{replace} \in \mathsf{tags}^{\mathcal{M}}(c))\}
\end{array}$$

Let $\mathsf{Intruder}$ be a new component ($\mathsf{Intruder} \notin \mathsf{Component}^{\mathcal{M}}$), which is added to the model of the system:

$$\mathsf{Component}^{\mathcal{M}'} \;=\; \mathsf{Component}^{\mathcal{M}} \cup \{\mathsf{Intruder}\}$$

For each channel ch affected by the transformation, we define an additional channel ch^{Intr} and two intruder ports $p^{\mathsf{Intr}}_{ch;\mathsf{in}}$ and $p^{\mathsf{Intr}}_{ch;\mathsf{out}}$:

$$\begin{array}{lcl}
\mathsf{IntrChannel}(\mathcal{M}) & = & \{ch^{\mathsf{Intr}} : \; ch \in \mathsf{PublicChannel}(\mathcal{M}) \cup \mathsf{ReplaceChannel}(\mathcal{M})\} \\
\mathsf{IntrInPort}(\mathcal{M}) & = & \{p^{\mathsf{Intr}}_{ch;\mathsf{in}} : \; ch \in \mathsf{PublicChannel}(\mathcal{M}) \cup \mathsf{ReplaceChannel}(\mathcal{M})\} \\
\mathsf{IntrOutPort}(\mathcal{M}) & = & \{p^{\mathsf{Intr}}_{ch;\mathsf{out}} : \; ch \in \mathsf{PublicChannel}(\mathcal{M}) \cup \mathsf{ReplaceChannel}(\mathcal{M})\}
\end{array}$$

The additional channels and ports must not appear in $\mathcal{M}$, i.e. $\mathsf{IntrChannel}(\mathcal{M}) \cap \mathsf{Channel}^{\mathcal{M}} = \emptyset$ and $(\mathsf{IntrInPort}(\mathcal{M}) \cup \mathsf{IntrOutPort}(\mathcal{M})) \cap \mathsf{Port}^{\mathcal{M}} = \emptyset$. They are added to $\mathcal{M}$ as follows:

$$\begin{array}{lcl}
\mathsf{Channel}^{\mathcal{M}'} & = & \mathsf{Channel}^{\mathcal{M}} \cup \mathsf{IntrChannel}(\mathcal{M}) \\
\mathsf{channels}^{\mathcal{M}'}(\mathsf{Main}) & = & \mathsf{channels}^{\mathcal{M}}(\mathsf{Main}) \cup \mathsf{IntrChannel}(\mathcal{M}) \\
\mathsf{Port}^{\mathcal{M}'} & = & \mathsf{Port}^{\mathcal{M}} \cup \mathsf{IntrInPort}(\mathcal{M}) \cup \mathsf{IntrOutPort}(\mathcal{M}) \\
\mathsf{inPorts}^{\mathcal{M}'}(\mathsf{Intruder}) & = & \mathsf{IntrInPort}(\mathcal{M}) \\
\mathsf{outPorts}^{\mathcal{M}'}(\mathsf{Intruder}) & = & \mathsf{IntrOutPort}(\mathcal{M})
\end{array}$$

Finally, the channels in $\mathcal{M}$ that are annotated with public or connected to components annotated with public or $\mathsf{replace}$ are re-routed through the $\mathsf{Intruder}$ component and given the correct types. For each channel $ch \in \mathsf{PublicChannel}(\mathcal{M}) \cup$

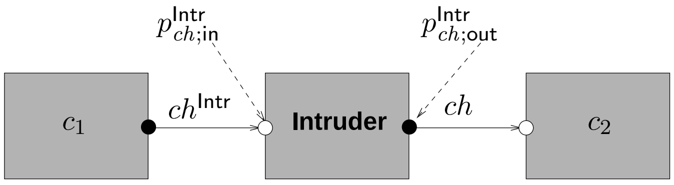

$p^{\mathsf{Intr}}_{ch;\mathsf{in}}$
$p^{\mathsf{Intr}}_{ch;\mathsf{out}}$
c_1
ch^{Intr}
Intruder
ch
c_2

Figure 5.3.: Threat Scenarios: Transformation of Channel

$\mathsf{ReplaceChannel}(\mathcal{M})$, we have:

$$\begin{array}{lcl} \mathsf{sourceP}^{\mathcal{M}'}(ch^{\mathsf{Intr}}) & = & \mathsf{sourceP}^{\mathcal{M}}(ch) \\ \mathsf{destP}^{\mathcal{M}'}(ch^{\mathsf{Intr}}) & = & p^{\mathsf{Intr}}_{ch;\mathsf{in}} \\ \mathsf{sourceP}^{\mathcal{M}'}(ch) & = & p^{\mathsf{Intr}}_{ch;\mathsf{out}} \end{array}$$

$$\mathsf{typeof}^{\mathcal{M}'}(ch^{\mathsf{Intr}}) = \mathsf{typeof}^{\mathcal{M}'}(p^{\mathsf{Intr}}_{ch;\mathsf{in}}) = \mathsf{typeof}^{\mathcal{M}'}(p^{\mathsf{Intr}}_{ch;\mathsf{out}}) = \mathsf{typeof}^{\mathcal{M}}(ch)$$

Figure 5.3 depicts the result of this transformation for two components c_1 and c_2 that were previously connected via a channel ch.

5.2.2. Semantics of the Behavioural View

The behavioural view describes how the intruder can interfere with the system. The intruder's capabilities are dependent on

- the threat annotations that lead to the insertion of the Intruder component (i.e., public channels or public/replace components, as described above);
- the annotated assumptions such as secret, which weaken the intruder's capabilities to access and manipulate the communication;
- the use of cryptography and secrets and the assumptions about the properties of the cryptographic mechanisms; and
- the initial knowledge of the intruder.

We specify the behaviour of the intruder on the semantical level, in the form of a discrete system. This makes it more general and independent of idiosyncrasies of AutoFocus/Quest state transition diagrams (most importantly, the fact that transitions have to be deterministic and take one time tick to execute, and missing support for sets). Besides, the generic intruder model is more suitable for proof purposes. A behavioural specification of a specialised intruder as a network of

AutoFocus/Quest STDs, tailored for model checking using the tool connection of AutoFocus/Quest to SMV, is described in Section 5.3.2.

Let $\mathsf{IK}(\mathcal{M})$ be the intruder's *initial knowledge*, given as follows:

$$\begin{aligned}\mathsf{IK}(\mathcal{M}) \quad = \quad & \{x \in \textstyle\bigcup_{\mathsf{type}^k \in \mathsf{T}_{\mathsf{Key}}} \mathsf{Value}(\mathsf{type}^k) : \mathsf{I}(\mathsf{knowsIntruder}^k)(x) = \mathsf{True}\} \cup \\ & \textstyle\bigcup_{c \in \mathsf{subComponents}^{\mathcal{M}}(\mathsf{Main}) : \mathsf{public} \in \mathsf{tags}^{\mathcal{M}}(c)} \mathsf{keys}(\mathcal{M}, c)\end{aligned}$$

Thus, without further analysis of the messages transmitted, part of the keys as specified by $\mathsf{knowsIntruder}^k$ plus the keys $\mathsf{keys}(\mathcal{M}, c)$ that appear in the specification of the components annotated with public are available to the intruder. Note that for types not in $\mathsf{T}_{\mathsf{Key}}$, arbitrary values can be generated via $\mathsf{derivable}$.

The set $\mathsf{keys}(\mathcal{M}, c)$ of keys that appear in the specification of the component c is defined by:

$$\mathsf{keys}(\mathcal{M}, c) = \{x : x \in \textstyle\bigcup_{\mathsf{type}^k \in \mathsf{T}_{\mathsf{Key}}} \mathsf{Value}(\mathsf{type}^k) \wedge \exists t \in \mathsf{terms}(\mathcal{M}, c) : x \trianglelefteq t\}$$

Here, $\mathsf{terms}(\mathcal{M}, c)$ is the set of terms appearing in the specification of the automaton $\mathsf{automaton}^{\mathcal{M}}(c)$ as precondition, input or output expression or action of a transition.

Using these definitions, the behaviour of the generic $\mathsf{Intruder}$ component is given by the discrete system $[\![\mathsf{Intruder}]\!]^{\mathsf{G}}_{\mathcal{M}'} = (V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}, I^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}, T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}})$ whose specification is shown in Figure 5.4. The superscript G stands for "generic".

The state of the $\mathsf{Intruder}$ component consists of the state of its input and output ports, of the intruder knowledge $\mathsf{K}_{\mathsf{Intr}}$ with $\beta(\mathsf{K}_{\mathsf{Intr}}) \in \mathcal{P}(\mathsf{Value})$ and of message stores $\mathsf{M}_p \in \mathcal{P}(\mathsf{Value})$ for messages received via the input ports $p \in \mathsf{IntrInPort}(\mathcal{M})$. Initially, the M_p are empty and the intruder knowledge contains at most the values derivable from $\mathsf{IK}(\mathcal{M})$.

At each execution step, the messages received via the ports $p \in \mathsf{IntrInPort}(\mathcal{M})$ can be added to the corresponding message histories (line (1) in the definition of $T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}(\beta, \beta')$). The new intruder knowledge is given by the values that can be derived from the current intruder knowledge, the messages received and the initial knowledge (2). To keep the specification general, we also allow that received messages are not remembered or that previously received messages or messages in the intruder knowledge are forgotten (hence we use the "$\subseteq$" relation rather than equality). The initial intruder knowledge is treated as knowledge available to the intruder at any time during the execution (rather than only at the beginning, as in the usual attacker formalisations), which is also more general and facilitates the consideration of specialised intruder specifications with limited storage capacity later on. The output of the intruder is either empty, a value in his knowledge, or a replay chosen from one of the message stores (3). Note that the messages output

$$
\begin{aligned}
V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}} &= \mathsf{IntrInPort}(\mathcal{M}) \cup \mathsf{IntrOutPort}(\mathcal{M}) \cup \\
&\quad \{\mathsf{K}_{\mathsf{Intr}}\} \cup \{\mathsf{M}_p : p \in \mathsf{IntrInPort}(\mathcal{M})\} \cup V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{replace}} \\[1em]
I^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}(\beta) &= \beta(\mathsf{K}_{\mathsf{Intr}}) \subseteq \mathsf{derivable}(\mathsf{IK}(\mathcal{M})) \wedge \\
&\quad \forall p \in \mathsf{IntrInPort}(\mathcal{M}) : \beta(\mathsf{M}_p) = \emptyset \\[1em]
T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}(\beta, \beta') &= \forall p \in \mathsf{IntrInPort}(\mathcal{M}) : \beta'(\mathsf{M}_p) \subseteq \beta(\mathsf{M}_p) \cup \{\beta(p)\} \wedge && (1) \\
&\quad \beta'(\mathsf{K}_{\mathsf{Intr}}) \subseteq \mathsf{derivable}\Big(\beta(\mathsf{K}_{\mathsf{Intr}}) \cup && (2) \\
&\qquad\qquad \textstyle\bigcup_{p \in \mathsf{IntrInPort}(\mathcal{M})} \{\beta(p)\} \cup \mathsf{IK}(\mathcal{M})\Big) \wedge \\
&\quad \forall p \in \mathsf{IntrOutPort}(\mathcal{M}) : \beta(p) \in \mathsf{Value}(\mathsf{typeof}^{\mathcal{M}}(p)) \wedge && (3) \\
&\qquad \beta(p) \in \{\bot\} \cup \beta'(\mathsf{K}_{\mathsf{Intr}}) \cup \textstyle\bigcup_{p \in \mathsf{IntrInPort}(\mathcal{M})} \beta'(\mathsf{M}_p) \wedge \\
&\quad T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{replace}}(\beta, \beta') \wedge T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{secret}}(\beta, \beta') \wedge \\
&\quad T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{auth}}(\beta, \beta') \wedge T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{integrity}}(\beta, \beta') \wedge \\
&\quad T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{noreplay}}(\beta, \beta') \wedge T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{avail}}(\beta, \beta')
\end{aligned}
$$

Figure 5.4.: Behavioural Specification of Generic Intruder Component

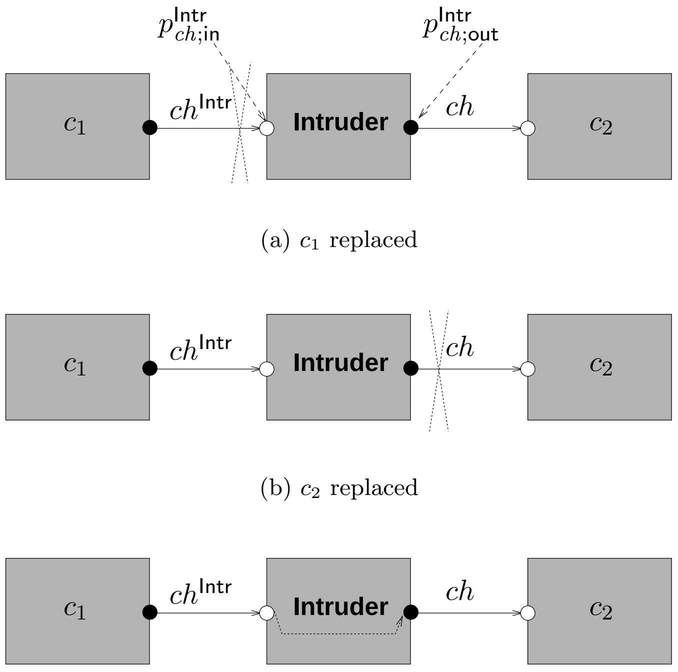
must be of the correct type. Moreover, the intruder outputs his values *immediately* ($\beta(p)$ is used rather than $\beta'(p)$). This makes the intruder transparent to the other components and allows him to react faster than them by waiting until near the end of the clock cycle and only then sending his messages. Such a behaviour is called "rushing attackers" in [Pfi98].

The additional variables and conjuncts appearing in the specification, such as $T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{replace}}$, specify the restrictions of the intruder's behaviour due to the consideration of the annotated threats and assumptions. They are described in the following.

Threat: Replaceable/Public Components

As explained in Section 5.2.1, for a component c annotated with public or replace there are two cases to be considered in the threat scenario: either c was replaced and the messages originally sent to and received by c are instead sent to and received by the intruder; or c was not replaced so the communication from/to c must remain unchanged.

We specify this by an additional set of boolean variables $\mathsf{replace}_c$ indicating if c has been replaced by the intruder in the current execution scenario. We do not fix an initial value, i.e. the value of $\mathsf{replace}_c$ is nondeterministically decided at the beginning of the execution.



$p^{\mathsf{Intr}}_{ch;\mathsf{in}}$
$p^{\mathsf{Intr}}_{ch;\mathsf{out}}$
c_1
ch^{Intr}
Intruder
ch
c_2

(a) c_1 replaced

(b) c_2 replaced

(c) neither c_1 nor c_2 replaced

Figure 5.5.: Replaceable/Public Components

$$
\begin{aligned}
V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{replace}} \quad = \quad & \{\mathsf{replace}_c : c \in \mathsf{subComponents}^{\mathcal{M}}(\mathsf{Main}) \wedge \\
& (\mathsf{replace} \in \mathsf{tags}^{\mathcal{M}}(c) \vee \mathsf{public} \in \mathsf{tags}^{\mathcal{M}}(c))\}
\end{aligned}
$$

To see how the intruder behaviour is affected, consider Figure 5.5. If a component is replaced, it is cut off from the system and the intruder communicates with the connected components in its place. I.e., if c_1 is replaced, no messages can be read from port $p^{\mathsf{Intr}}_{ch;\mathsf{in}}$ but the intruder can send messages to the connected component c_2 via port $p^{\mathsf{Intr}}_{ch;\mathsf{out}}$ (Figure 5.5(a)). Conversely, if c_2 is replaced, the intruder cannot send messages to c_2 via $p^{\mathsf{Intr}}_{ch;\mathsf{out}}$, but receive messages from c_1 via $p^{\mathsf{Intr}}_{ch;\mathsf{in}}$ (Figure 5.5(b)). If both c_1 and c_2 are not replaced, the messages must be forwarded unchanged, except if the channel ch was annotated with public (Figure 5.5(c)). For now, we assume that the set of components that have been replaced does not change during an execution. However, $T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{replace}}$ can be easily adjusted to include this possibility. Such a scenario is examined in Section 5.4.2 (CEPS Purchase Transaction).

For a formal definition, for each channel $ch \in \mathsf{ReplaceChannel}(\mathcal{M})$ we define two predicates $\mathsf{replaceSourceComp}(\mathcal{M}, ch)$ and $\mathsf{replaceDestComp}(\mathcal{M}, ch)$ indicating if ch originates at or leads to a component that is currently replaced:

$$\begin{aligned}
&\mathsf{replaceSourceComp}(\mathcal{M}, ch) \ = \\
&\quad \exists c \in \mathsf{Component} : \mathsf{sourceP}^{\mathcal{M}}(ch) \in \mathsf{outPorts}^{\mathcal{M}}(c) \wedge \\
&\quad (\mathsf{replace} \in \mathsf{tags}(c) \vee \mathsf{public} \in \mathsf{tags}(c)) \wedge \mathsf{replace}_c
\end{aligned}$$

$$\begin{aligned}
&\mathsf{replaceDestComp}(\mathcal{M}, ch) \ = \\
&\quad \exists c \in \mathsf{Component} : \mathsf{destP}^{\mathcal{M}}(ch) \in \mathsf{inPorts}^{\mathcal{M}}(c) \wedge \\
&\quad (\mathsf{replace} \in \mathsf{tags}(c) \vee \mathsf{public} \in \mathsf{tags}(c)) \wedge \mathsf{replace}_c
\end{aligned}$$

Using these predicates, the transition relation is restricted by the additional conjunct $T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{replace}}(\beta, \beta')$ as follows:

$$\begin{aligned}
&T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{replace}}(\beta, \beta') = \\
&\quad \forall ch \in \mathsf{ReplaceChannel}(\mathcal{M}) : \\
&\qquad \mathsf{replaceSourceComp}(\mathcal{M}, ch) \Rightarrow \\
&\qquad\quad \beta'(\mathsf{M}_{p^{\mathsf{Intr}}_{ch;\mathsf{in}}}) = \beta(\mathsf{M}_{p^{\mathsf{Intr}}_{ch;\mathsf{in}}}) \wedge \\
&\qquad\quad \beta'(\mathsf{K}_{\mathsf{Intr}}) \subseteq \mathsf{derivable}\big(\beta(\mathsf{K}_{\mathsf{Intr}}) \cup \\
&\qquad\qquad\qquad\qquad \textstyle\bigcup_{p \in \mathsf{IntrInPort}(\mathcal{M}) \setminus p^{\mathsf{Intr}}_{ch;\mathsf{in}}} \{\beta(p)\} \cup \mathsf{IK}(\mathcal{M})\big) \wedge \\
&\qquad \mathsf{replaceDestComp}(\mathcal{M}, ch) \Rightarrow \\
&\qquad\quad \beta(p^{\mathsf{Intr}}_{ch;\mathsf{out}}) = \perp \ \wedge \\
&\qquad (\neg \mathsf{replaceSourceComp}(\mathcal{M}, ch) \wedge \neg \mathsf{replaceDestComp}(\mathcal{M}, ch) \wedge \\
&\qquad\qquad \mathsf{public} \notin \mathsf{tags}(ch)) \Rightarrow \\
&\qquad\quad \beta'(\mathsf{M}_{p^{\mathsf{Intr}}_{ch;\mathsf{in}}}) = \beta(\mathsf{M}_{p^{\mathsf{Intr}}_{ch;\mathsf{in}}}) \wedge \\
&\qquad\quad \beta'(\mathsf{K}_{\mathsf{Intr}}) \subseteq \mathsf{derivable}\big(\beta(\mathsf{K}_{\mathsf{Intr}}) \cup \\
&\qquad\qquad\qquad\qquad \textstyle\bigcup_{p \in \mathsf{IntrInPort}(\mathcal{M}) \setminus p^{\mathsf{Intr}}_{ch;\mathsf{in}}} \{\beta(p)\} \cup \mathsf{IK}(\mathcal{M})\big) \wedge \\
&\qquad\quad \beta(p^{\mathsf{Intr}}_{ch;\mathsf{out}}) = \beta(p^{\mathsf{Intr}}_{ch;\mathsf{in}}) \wedge \\
&\quad \forall c \in \mathsf{subComponents}^{\mathcal{M}}(\mathsf{Main}) : (\mathsf{replace} \in \mathsf{tags}^{\mathcal{M}}(c) \vee \mathsf{public} \in \mathsf{tags}^{\mathcal{M}}(c)) \Rightarrow \\
&\qquad \beta'(\mathsf{replace}_c) = \beta(\mathsf{replace}_c)
\end{aligned}$$

Assumption: Secret Channels

Channels annotated with secret cannot be eavesdropped by the intruder:

$$\begin{aligned}
&T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{secret}}(\beta, \beta') = \\
&\quad \forall p^{\mathsf{Intr}}_{ch;\mathsf{in}} \in \mathsf{IntrInPort}(\mathcal{M}) : \mathsf{secret} \in \mathsf{tags}^{\mathcal{M}}(\mathsf{sourceP}^{\mathcal{M}}(ch)) \Rightarrow \\
&\qquad \beta'(\mathsf{K}_{\mathsf{Intr}}) \subseteq \mathsf{derivable}\left(\beta(\mathsf{K}_{\mathsf{Intr}}) \cup \textstyle\bigcup_{p \in \mathsf{IntrInPort}(\mathcal{M}) \setminus p^{\mathsf{Intr}}_{ch;\mathsf{in}}} \{\beta(p)\} \cup \mathsf{IK}(\mathcal{M})\right)
\end{aligned}$$

Assumption: Authentic Channels

Only replays of the values from the same channel or the empty value can be inserted into channels annotated with auth:

$$\begin{aligned}
&T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{auth}}(\beta,\beta') = \\
&\quad \forall p^{\mathsf{Intr}}_{ch;\mathsf{out}} \in \mathsf{IntrOutPort}(\mathcal{M}) : \mathsf{auth} \in \mathsf{tags}^{\mathcal{M}}(\mathsf{sourceP}^{\mathcal{M}}(ch)) \Rightarrow \\
&\qquad \beta(p^{\mathsf{Intr}}_{ch;\mathsf{out}}) \in \{\bot\} \cup \beta'(\mathsf{M}_{p^{\mathsf{Intr}}_{ch;\mathsf{in}}})
\end{aligned}$$

Assumption: Integrity-Preserving Channels

Only replays or the empty value can be inserted into channels annotated with integrity:

$$\begin{aligned}
&T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{integrity}}(\beta,\beta') = \\
&\quad \forall p^{\mathsf{Intr}}_{ch;\mathsf{out}} \in \mathsf{IntrOutPort}(\mathcal{M}) : \mathsf{auth} \in \mathsf{tags}^{\mathcal{M}}(\mathsf{sourceP}^{\mathcal{M}}(ch)) \Rightarrow \\
&\qquad \beta(p^{\mathsf{Intr}}_{ch;\mathsf{out}}) \in \{\bot\} \cup \bigcup_{p \in \mathsf{IntrInPort}(\mathcal{M})} \mathsf{M}_p
\end{aligned}$$

Assumption: Replay-Protected Channels

Replays of earlier messages are not possible for channels annotated with noreplay:

$$\begin{aligned}
&T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{noreplay}}(\beta,\beta') = \\
&\quad \forall p^{\mathsf{Intr}}_{ch;\mathsf{out}} \in \mathsf{IntrOutPort} : \mathsf{noreplay} \in \mathsf{tags}^{\mathcal{M}}(\mathsf{sourceP}^{\mathcal{M}}(ch)) \Rightarrow \\
&\qquad \beta(p^{\mathsf{Intr}}_{ch;\mathsf{out}}) \in \{\beta(p^{\mathsf{Intr}}_{ch;\mathsf{in}})\} \cup \{\bot\} \cup \beta'(\mathsf{K}_{\mathsf{Intr}})
\end{aligned}$$

Assumption: Channels Ensuring Availability

If a channel is annotated with avail, the messages are forwarded unchanged and without delay. [1]

$$\begin{aligned}
&T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder};\mathsf{avail}}(\beta,\beta') = \\
&\quad \forall p^{\mathsf{Intr}}_{ch;\mathsf{out}} \in \mathsf{IntrOutPort}(\mathcal{M}) : \mathsf{auth} \in \mathsf{tags}^{\mathcal{M}}(\mathsf{sourceP}^{\mathcal{M}}(ch)) \Rightarrow \\
&\qquad \beta(p^{\mathsf{Intr}}_{ch;\mathsf{out}}) = \beta(p^{\mathsf{Intr}}_{ch;\mathsf{in}})
\end{aligned}$$

[1] This is a very strong notion of availability. Weaker notions (e.g. allowing a delay) would more easily be formulated as trace properties than as part of the transition relation and require the specification of a “fair” intruder who does not block messages completely. As most cryptographic protocols do not ensure availability, we do not consider availability in more detail in this thesis.

Interpretation of Properties Referring to Intruder Knowledge

Finally, we show how properties referring to the intruder knowledge using the special functions $\mathsf{learnedIntruder}^k$ are interpreted with respect to the generic threat scenario: The evaluation of $\mathsf{learnedIntruder}^k$ is defined by

$$\mathsf{eval}^{\mathsf{G}}_{\beta}(\mathsf{learnedIntruder}^k(t)) \;=\; \mathsf{eval}^{\mathsf{G}}_{\beta}(t) \in \beta(\mathsf{K}_{\mathsf{Intr}})$$

In the other cases, $\mathsf{eval}^{\mathsf{G}}_{\beta}$ is defined in the same way as eval_{β} (see Section 4.1.3). By $[\![pr]\!]^{\mathsf{G}}$, we denote the semantics of a property pr where $\mathsf{eval}^{\mathsf{G}}_{\beta}$ is used for the evaluation of terms that are not LTL properties.

5.3. Security Verification Using Model Checking

The threat scenario derived from the security-enriched model forms the basis of security verification (see Figure 5.6):

Definition 5.3.1. A security-enriched model is *verified secure* if the stated global security requirements are fulfilled in the corresponding threat scenario $\mathcal{M}'$ (here, Main is the top-level component of $\mathcal{M}'$):

$$\begin{aligned} \forall pr \in \mathsf{properties}^{\mathcal{M}'}(\mathsf{Main}) : \mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}'}(pr) \Rightarrow& \\ \forall \sigma : \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'}}(\sigma) \Rightarrow [\![pr]\!](\sigma)& \end{aligned}$$

If the model is verified secure, one can continue with implementation and testing activities (for testing, see Chapter 6). Besides, if there are assumptions in the model that are not fulfilled by the environment or by measures out of scope of the model, appropriate security mechanisms must be applied at later stages of the development (see Chapter 7).

On the other hand, if a security requirement is violated, the security-enriched model must be adjusted. This can be achieved by the following measures:

(1) revision of the violated security requirement (because it was stronger than intended or incorrectly formalised),

(2) weakening threats or strengthening assumptions (for instance by removing the public annotations or introducing secret or auth annotations), or

(3) adjusting or introducing security mechanisms.

Note that there is often a choice between (2) or (3): for example, data can be kept confidential by either ensuring that it is only transmitted via channels annotated with secret, or by encrypting it with an appropriate key. In the former case, it

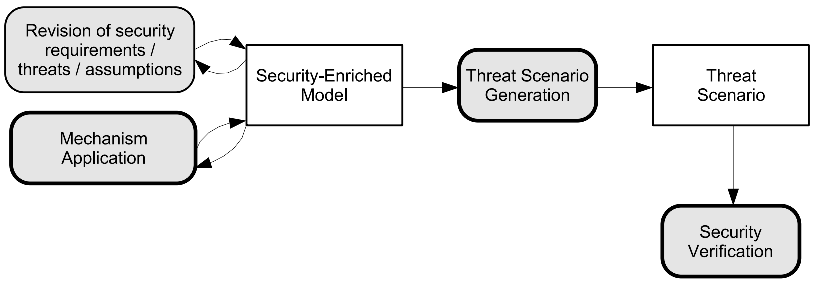

Revision of security requirements / threats / assumptions
Mechanism Application
Security-Enriched Model
Threat Scenario Generation
Threat Scenario
Security Verification

Figure 5.6.: Security Verification Process

must again be ensured that the weakening of the threats or the strengthening of the assumptions is justified — if necessary, by introducing security mechanisms in a second step. Adjusting threats and assumptions first to fulfil the security requirements leads to a more abstract model that can be more easily verified. In Chapter 7, this approach is described in more detail at the example of layered protocols.

In the remaining part of this section, we will address security verification using model checking.

5.3.1. Model Checking in AutoFocus/Quest

AutoFocus/Quest offers an interface to the model checker SMV [McM93] (including the NuSMV [CCG+02] and Cadence SMV[McM99] versions), based on the semantics described in Section 4.1.5. The translation of AutoFocus/Quest models to the input language of SMV is described in more detail in [Wim00]. The basic idea of the translation is to utilise the feature of the SMV input language to directly specify a system by its state variables ("`VAR`" declaration) and by predicates that characterise the initial states and the transition relation ("`INIT`" and "`TRANS`" declarations). The `VAR`, `INIT` and `TRANS` declarations correspond to the components V, I, T of the discrete system model $\mathcal{D} = (V, I, T)$ used as a semantics in Section 4.1.5. However, the SMV input language does not support hierarchic data types, which are therefore mapped appropriately to integer range types.

Security verification using the model checking interface of AutoFocus/Quest is carried out as follows (see Figure 5.7): The threat scenario is translated to a model in the SMV input language and the security requirements are translated to SMV specifications ("`SPEC`" declarations). The SMV model checker is then run on this input. If the model checker can finish the verification within a given time bound, for each requirement it is reported if it is true or false in the model. For a requirement that is false, a counterexample is given, which is a trace that demonstrates how the requirement can be violated (in case of security verification, this corresponds to

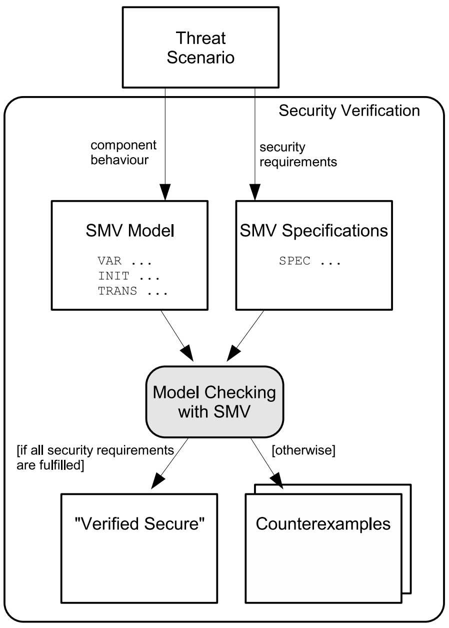


Figure 5.7.: Security Verification Using SMV

a possible attack). The counterexamples are re-translated to AutoFocus/Quest EETs.

If the model checker does not finish given the available time and memory resources, a partial verification can be carried out using bounded model checking [BCC+03]. Here, the verification is restricted to a search for counterexamples of a chosen maximum length l. As the execution time of bounded model checking depends exponentially on l, one starts with small values for l and increases it until again the given time or memory limits are exceeded. Support for bounded model checking has been built into the model checkers Cadence SMV and NuSMV.

Note that SMV is a classical finite-state symbolic model checker. This implies that an AutoFocus/Quest model can only be translated to SMV if it does not contain recursive data types or functions. Models with infinite entity sets (for example an infinite number of components, local variables or transitions) are not supported by the AutoFocus/Quest tool set.

5.3.2. Specialised Intruder Model for Model Checking

In the following, we describe the generation of a specification of the Intruder component in the AutoFocus/Quest specification language, as a network of STDs. The generated intruder model is specialised for security verification using model check-

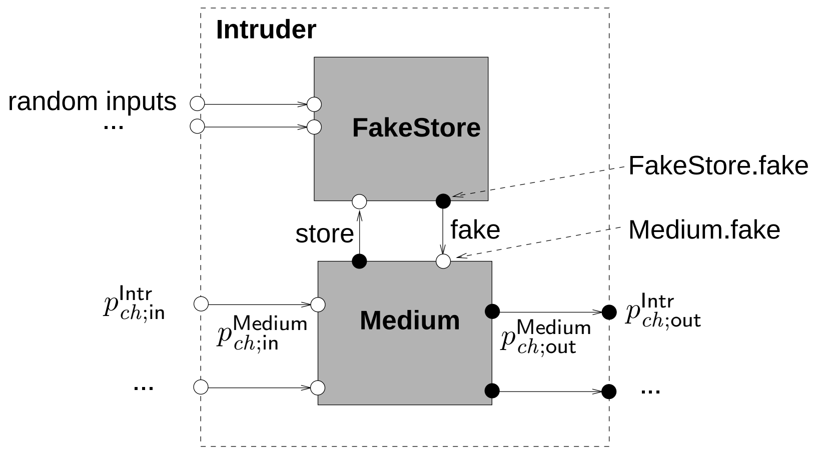

Intruder
random inputs
...
FakeStore
FakeStore.fake
store
fake
Medium.fake
$p^{\mathsf{Intr}}_{ch;\mathsf{in}}$
$p^{\mathsf{Medium}}_{ch;\mathsf{in}}$
Medium
$p^{\mathsf{Medium}}_{ch;\mathsf{out}}$
$p^{\mathsf{Intr}}_{ch;\mathsf{out}}$
...
...

Figure 5.8.: Decomposition of Intruder Component

ing via the SMV connection of AutoFocus/Quest. The semantics of the Intruder component is an adequate restriction of the generic intruder model presented in Section 5.2.2, as will be justified in Section 5.3.3. We refer to this intruder model as "specialised intruder model" and to the threat scenario using this intruder model as "specialised threat scenario".

We apply a pattern similar to the one described in [Lot97]: we split the Intruder component into two subcomponents Medium and FakeStore. The Medium component models the intruder's influence on the communication, whereas the FakeStore component models his knowledge and his abilities to manipulate cryptographic messages.

Figure 5.8 illustrates this substructure: the intruder's input and output ports $p^{\mathsf{Intr}}_{ch;\mathsf{in}}$ and $p^{\mathsf{Intr}}_{ch;\mathsf{out}}$ are connected with the Medium component via the ports $p^{\mathsf{Medium}}_{ch;\mathsf{in}}$ and $p^{\mathsf{Medium}}_{ch;\mathsf{out}}$ for reading and writing on the attacked channels. The Medium component is connected to the FakeStore component via two channels store and fake connecting the ports Medium.store and FakeStore.store, respectively Medium.fake and FakeStore.fake. Via the store channel, the FakeStore component receives eavesdropped messages, and via the fake channel, it passes generated messages to the Medium component that can be inserted into the communication. We require that all attacked channels (and therefore all ports $p^{\mathsf{Intr}}_{ch;\mathsf{in}}$ and $p^{\mathsf{Intr}}_{ch;\mathsf{out}}$) have the same type TMessage. [2]

The corresponding model transformation (decomposing the Intruder component into the two subcomponents, adding the corresponding ports and channels and

[2] This can easily be overcome by defining a more complex intruder model, e.g. with more than one store and fake channel.

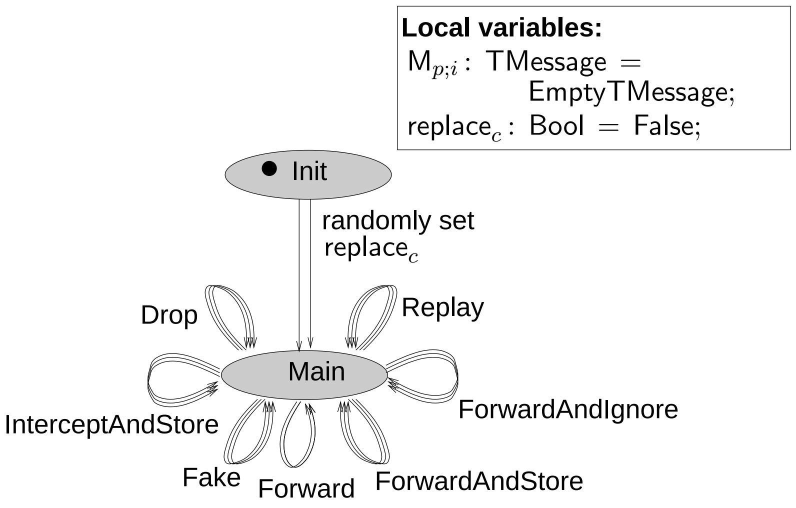


Figure 5.9.: STD Schema for Medium Component

generating state transition diagrams for Medium and FakeStore) can be formally described in a similar way to the structural transformation given in Section 5.2.1, which we omit here for space reasons. The transformation has been implemented into the AutoFocus/Quest tool in the form of a Java program.

In the following, we give a short explanation of the state transition diagrams generated for the Medium and FakeStore components.

The Medium Component in AutoFocus/Quest

Figure 5.9 shows the schema for the AutoFocus/Quest STD for the Medium component. The data state of the Medium component must be finite, so the message stores M_p in the generic intruder model are represented by a number of local variables $\mathsf{M}_{p;i}$. Their initial value is EmptyTMessage, which must be one of the constructors of TMessage and stands for an empty/illegal message. Besides, the Medium component contains the replace$_c$ variables indicating if a component c is currently replaced by the intruder (see Section 5.2.2). At the beginning of a system run, one of the transitions from state Init to state Main is chosen to initialise the replace$_c$ variables with random values. Then, the Medium component repeatedly executes transitions corresponding to actions the intruder can perform on an attacked channel (i.e., a channel $ch \in \mathsf{PublicChannel}(\mathcal{M}) \cup \mathsf{ReplaceChannel}(\mathcal{M})$). There are the following kinds of transitions for each attacked channel ch:

- InterceptAndStore: read a message from $p^{\mathsf{Medium}}_{ch;\mathsf{in}}$, forward it to FakeStore and store it in one of the $\mathsf{M}_{p^{\mathsf{Medium}}_{ch;\mathsf{in}};i}$ variables.

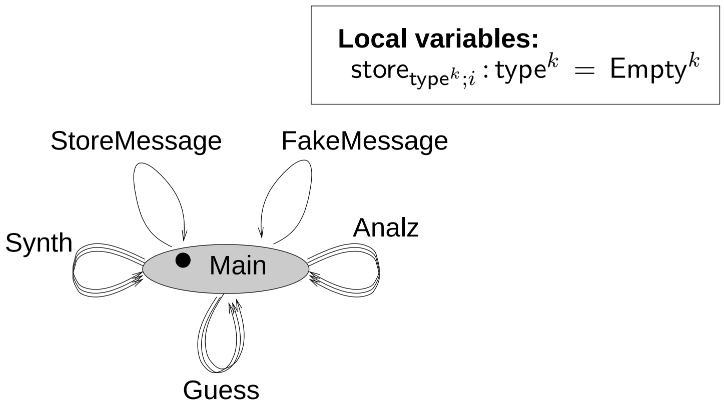


Figure 5.10.: STD Schema for FakeStore Component

- Fake: write a message generated by FakeStore to $p^{\mathsf{Medium}}_{ch;\mathsf{out}}$.
- Forward: read a message from $p^{\mathsf{Medium}}_{ch;\mathsf{in}}$, forward it unchanged and store it in one of the $\mathsf{M}_{p^{\mathsf{Medium}}_{ch;\mathsf{in}};i}$ variables.
- ForwardAndStore: combination of Forward and InterceptAndStore.
- ForwardAndIgnore: same as Forward, but do not store the message in one of the $\mathsf{M}_{p^{\mathsf{Medium}}_{ch;\mathsf{in}};i}$ variables.
- Replay: replay a message received from an attacked channel (and stored in one of the $\mathsf{M}_{p;i}$ variables) to a $p^{\mathsf{Medium}}_{ch;\mathsf{out}}$ port.
- Drop: read a message from $p^{\mathsf{Medium}}_{ch;\mathsf{in}}$, store it in one of the $\mathsf{M}_{p;i}$ variables, and drop it.

Depending on the threat scenario specification, some of these transitions are omitted or augmented with preconditions that control if they can be executed. For example, for a public channel, all above actions are possible, whereas for a channel annotated with secret, the InterceptAndStore and ForwardAndStore transitions must be omitted. Preconditions containing the $\mathsf{replace}_c$ variables are added for channels $ch \in \mathsf{ReplaceChannel}(\mathcal{M})$, because the possible intruder actions depend on which component is currently replaced.

The FakeStore Component in AutoFocus/Quest

Figure 5.10 shows the schema for the AutoFocus/Quest STD for the FakeStore component. The FakeStore component corresponds to the function derivable (see

Section 4.2.6): it accepts messages on the FakeStore.store port and outputs messages on the fake port that can be derived from the received messages using operations such as encryption, decryption, concatenation or splitting.

For each type type^k of terms that can be subterms of a term of type TMessage (including TMessage itself), FakeStore has a number of variables $\mathsf{store}_{\mathsf{type}^k;i}$. The $\mathsf{store}_{\mathsf{type}^k;i}$ variables are initially assigned to Empty^k, standing for an empty/illegal value. The STD for FakeStore has only one control state Main and the following kinds of transitions:

- StoreMessage: read a message from the store port and write it to the local store.
- FakeMessage: write a message from the local store to the fake port (to be forwarded to the attacked system by the Medium component).
- Analz transitions: extract a part of a stored message and write the part to the local store. To extract parts of encrypted messages, check if the corresponding key is in the local store as well.
- Synth transitions: combine messages from the local store and write the result to the local store.
- Guess transitions: nondeterministically determine a value x in the initial knowledge of the intruder (i.e., for which $\mathsf{eval}_\beta(\mathsf{knowsIntruder}(x)) = \mathsf{True}$) and write it to the local store.

At each execution step, the action to be performed is nondeterministically chosen. The only way to specify nondeterministic behaviour in AutoFocus/Quest is via multiple transitions that are executable at the same time — the result of the execution of a transition must be deterministic. Therefore, to simplify the model, the STD for the FakeStore component also depends on external random inputs (see Figure 5.8). For example, in a guess transition, the value x is read from the environment, which is not further constrained (i.e., the security requirements must hold for any possible sequence of random inputs).

Interpretation of Properties Referring to Intruder Knowledge

Finally, we show how properties referring to the intruder knowledge using the special functions $\mathsf{learnedIntruder}^k$ are interpreted with respect to the specialised threat scenario: within the definition of the semantics of properties in Section 4.1.5, the evaluation of $\mathsf{learnedIntruder}^k$ is defined by

$$\begin{aligned}\mathsf{eval}_\beta(\mathsf{learnedIntruder}^k(t)) \quad = \quad & \exists i : \mathsf{eval}_\beta(\mathsf{store}_{\mathsf{type}^k;i} == t) = \mathsf{True} \vee \\ & \mathsf{eval}_\beta(\mathsf{store} == t) = \mathsf{True}\end{aligned}$$

This definition states that t is in the intruder knowledge if the value of t is stored in one of the variables $\mathsf{store}_{\mathsf{type}^k;i}$ or currently sent to the FakeStore component via the FakeStore.store port.

5.3.3. Justification of the Specialised Intruder Model

Specialised intruder models are designed for a specific purpose. To serve this purpose, they can be limited with respect to the generic threat scenario. For example, the model checking specific threat scenario described in Section 5.3.2 must be finite state and of a complexity that allows model checking with reasonable execution times and space requirements. In addition, idiosyncrasies of the specification language have to be taken into account, such as the fact that transitions in AutoFocus/Quest STDs are always deterministic and that their execution takes up one time tick.

A specialised intruder model must be justified with respect to the generic threat scenario, such as the one described in Section 5.2 for AutoFocus/Quest. Desirable properties are soundness and completeness: we say that a specialised intruder model is sound if it does not allow more attacks than the generic intruder model, and that it is complete, if whenever the generic intruder can perform a certain attack, the specialised intruder can do so as well. If the latter is not the case, there will be security violations (with respect to the generic intruder model) that can not be found using the specialised intruder model. If a specialised threat scenario is not complete for all possible security-enriched models, one should state under which conditions it is complete. Note that even the generic threat scenario must necessarily be incomplete in some way (with respect to the actual attacks that can be performed on the system), as it is an abstraction and only corresponds to part of the whole system, given certain assumptions (e.g. only a certain number of parallel executions of a specified transaction protocol). However, even incomplete specialised intruder models are still beneficial to find attacks. One just has to keep in mind that a successful security verification does not indicate their absence.

In this section, we formally justify the model-checking specific threat scenario with respect to the generic threat scenario. We show that it is sound and give conditions under which it is complete.

Soundness

Soundness of the specialised intruder model ensures that if an attack is found using the specialised threat scenario (for example by performing model checking), there is also an attack with respect to the generic threat scenario.

For a formal definition, let us denote by $[\![\mathsf{Main}]\!]^{\mathsf{G}}_{\mathcal{M}'} = (V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Main}}, I^{\mathsf{G}}_{\mathcal{M}';\mathsf{Main}}, T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Main}})$ the semantics of the generic threat scenario, where $[\![\mathsf{Intruder}]\!]^{\mathsf{G}}_{\mathcal{M}'} = (V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}},$

$I^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}, T^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}})$ is used as the semantics of the Intruder component. Let us denote by $[\![\mathsf{Main}]\!]_{\mathcal{M}'} = (V_{\mathcal{M}';\mathsf{Main}}, I_{\mathcal{M}';\mathsf{Main}}, T_{\mathcal{M}';\mathsf{Main}})$ the semantics of the specialised threat scenario, where $[\![\mathsf{Intruder}]\!]_{\mathcal{M}'} = (V_{\mathcal{M}';\mathsf{Intruder}}, I_{\mathcal{M}';\mathsf{Intruder}}, T_{\mathcal{M}';\mathsf{Intruder}})$ is used as the semantics of the Intruder component. Besides, let $\Psi^{\mathsf{G}} := \Psi_{[\![\mathsf{Main}]\!]^{\mathsf{G}}_{\mathcal{M}'}}$ characterise the set of computations in the generic threat scenario and $\Psi^{\mathsf{S}} := \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'}}$ the set of computations in the specialised threat scenario. A specialised intruder model is sound if for all security properties pr, if there is a trace σ in Ψ^{S} that violates pr, there is also a trace σ' in Ψ^{G} that violates pr.

Definition 5.3.2. A specialised intruder model is *sound* with respect to a threat scenario $\mathcal{M}'$, if

$$\begin{aligned}&\forall pr \in \mathsf{properties}^{\mathcal{M}'}(\mathsf{Main}) : \mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}'}(pr) \Rightarrow\\ &\quad \exists\sigma : \Psi^{\mathsf{S}}(\sigma) \wedge \neg[\![pr]\!](\sigma) \Rightarrow\\ &\qquad \exists\sigma' : \Psi^{\mathsf{G}}(\sigma') \wedge \neg[\![pr]\!]^{\mathsf{G}}(\sigma')\end{aligned}$$

Theorem 5.3.3. *The specialised intruder model described in Section 5.3.2 is sound.*

Proof Sketch To prove Theorem 5.3.3, it is sufficient to show that the specialised intruder is a refinement of the generic intruder in the following sense:

Theorem 5.3.4. *Let $\sigma^{\mathsf{S}} = [\beta^{\mathsf{S}}_0, \beta^{\mathsf{S}}_1, \ldots]$ be a computation of the specialised intruder, i.e.*
$\Psi_{[\![\mathsf{Intruder}]\!]_{\mathcal{M}'}}(\sigma^{\mathsf{S}})$ is true. Then there is a computation $\sigma^{\mathsf{G}} = [\beta^{\mathsf{G}}_0, \beta^{\mathsf{G}}_1, \ldots]$ of the generic intruder (i.e., $\Psi_{[\![\mathsf{Intruder}]\!]^{\mathsf{G}}_{\mathcal{M}'}}(\sigma^{\mathsf{G}})$ is true) where for all $j \geq 0$:

(1) $\forall p^{\mathsf{Intr}}_{ch;\mathsf{in}} \in \mathsf{IntrInPort}(\mathcal{M}) : \beta^{\mathsf{G}}_j(p^{\mathsf{Intr}}_{ch;\mathsf{in}}) = \beta^{\mathsf{S}}_j(p^{\mathsf{Intr}}_{ch;\mathsf{in}})$

(2) $\forall p^{\mathsf{Intr}}_{ch;\mathsf{out}} \in \mathsf{IntrOutPort}(\mathcal{M}) : \beta^{\mathsf{G}}_j(p^{\mathsf{Intr}}_{ch;\mathsf{out}}) = \beta^{\mathsf{S}}_j(p^{\mathsf{Intr}}_{ch;\mathsf{out}})$

(3) $\beta^{\mathsf{G}}_j(\mathsf{K}_{\mathsf{Intr}}) = \bigcup_{i,\mathsf{type}^k} \beta^{\mathsf{S}}_j(\mathsf{store}_{\mathsf{type}^k;i}) \cup \beta^{\mathsf{S}}_j(\mathsf{store})$

This means that the corresponding computation σ^{G} has the same input/output behaviour and intruder knowledge as σ^{S} at each execution step.

We proved Theorem 5.3.4 by induction over j and case distinctions over the different kinds of transitions in the automata assigned to FakeStore and Medium components of the specialised intruder. We omit the proof, because it depends on specification details of the generated automata and is not necessary for further understanding of the thesis.

With the help of Theorem 5.3.4, we can now prove Theorem 5.3.3. Assume there is a trace σ with $\Psi^{\mathsf{S}}(\sigma) \wedge \neg[\![pr]\!](\sigma)$. Let $\sigma^{\mathsf{S}} := \sigma|_{V_{\mathcal{M}';\mathsf{Intruder}}}$ be the projection of σ with respect to the state variables of the specialised intruder. Then, $\Psi_{[\![\mathsf{Intruder}]\!]_{\mathcal{M}'}}(\sigma^{\mathsf{S}})$ is true. By Theorem 5.3.4, there is a trace σ^{G} for which $\Psi_{[\![\mathsf{Intruder}]\!]^{\mathsf{G}}_{\mathcal{M}'}}(\sigma^{\mathsf{G}})$ is true, and σ^{G} and σ^{S} have equal input/output behaviour. The generic and the specialised threat scenario only differ with respect to the intruder behaviour. Thus, if we replace in σ the valuations of the variables $V_{\mathcal{M}';\mathsf{Intruder}}$ of the specialised intruder in σ^{S} by the valuations of the variables $V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}$ of the generic intruder in σ^{G}, we obtain a trace σ' with $\sigma'|_{V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Main}} \setminus V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}} = \sigma|_{V_{\mathcal{M}';\mathsf{Main}} \setminus V_{\mathcal{M}';\mathsf{Intruder}}}$ for which $\Psi^{\mathsf{G}}(\sigma')$ is true. This trace also fulfils $\neg[\![pr]\!]^{\mathsf{G}}(\sigma')$: For state variables not belonging to the intruder, σ and σ' have the same values at each execution step. The only way to refer to state variables of the intruder is via the $\mathsf{learnedIntruder}^k$ functions. However, $[\![\mathsf{learnedIntruder}^k(t)]\!](\sigma^j)$ states that the value of t is the same as the value of one of the variables $\mathsf{store}_{\mathsf{type}^k;i}$ or of the store port, and $[\![\mathsf{learnedIntruder}^k(t)]\!]^{\mathsf{G}}((\sigma')^j)$ states that the value of t is contained in $\mathsf{K}_{\mathsf{Intr}}$. Because of (3) in Theorem 5.3.4, $[\![\mathsf{learnedIntruder}^k(t)]\!](\sigma^j)$ and $[\![\mathsf{learnedIntruder}^k(t)]\!]^{\mathsf{G}}((\sigma')^j)$ are equivalent. □

Completeness

Completeness is the counterpart of soundness: a specialised intruder model is complete if each security requirement that can be violated in the generic threat scenario can also be violated in the threat scenario generated using the specialised intruder model. We use the same notation as defined above in the soundness case.

Definition 5.3.5. A specialised intruder model is *complete* with respect to a threat scenario $\mathcal{M}'$, if

$$\begin{aligned}&\forall pr \in \mathsf{properties}^{\mathcal{M}'}(\mathsf{Main}) : \mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}'}(pr) \Rightarrow\\ &\quad \exists \sigma : \Psi^{\mathsf{G}}(\sigma) \wedge \neg[\![pr]\!]^{\mathsf{G}}(\sigma) \Rightarrow\\ &\qquad \exists \sigma' : \Psi^{\mathsf{S}}(\sigma') \wedge \neg[\![pr]\!](\sigma')\end{aligned}$$

The specialised intruder model described in Section 5.3.2 is not complete for all possible threat scenarios and security requirements. It is tailored for an efficient security analysis of systems based on a request/reply architecture, using the model checking feature of AutoFocus/Quest. In particular, the intruder can only intercept or forge one message at a time (by the corresponding transitions of the Medium component) and requires a number of execution steps to process messages, thus introducing delay into the communication. For example, a message is intercepted by the Medium component and sent to the FakeStore component (InterceptAndStore transition), there it is stored (StoreMessage transition), analysed (Analz/Synth transitions), a fake message derived from the intruder knowledge is sent to the Medium component (FakeMessage transition) and then sent on to another component (Fake transition). The generic intruder can perform all these actions immediately, within the same time tick. Besides, as model checking is used for the analysis, the intruder can only store finitely many messages with a finite number of possible values. The specialised intruder model for model checking is complete under the following conditions:

(1) No recursive data types and functions are used.

(2) The Medium component has enough $\mathsf{M}_{p;i}$ variables to store all possible different messages that can be received at port p, and the $\mathsf{FakeStore}$ component has enough $\mathsf{store}_{\mathsf{type}^k;i}$ variables to store all values that can occur in the intruder's knowledge.

(3) The threat scenario has a request/reply architecture from the point of view of the intruder.

(4) The security properties are instances of the "Patterns for Common Global Security Requirements" described in Section 4.2.2.

Request/reply architectures are characterised as follows.

Definition 5.3.6. Let $\mathcal{U} \subseteq \mathsf{subComponents}^{\mathcal{M}'}(\mathsf{Main})$ be a set of components of the threat scenario such that $\mathsf{Intruder} \notin \mathcal{U}$. Let $[\![\mathcal{U}]\!] = (V_{\mathcal{U}}, I_{\mathcal{U}}, T_{\mathcal{U}})$ be the semantics of $\mathcal{U}$. $[\![\mathcal{U}]\!]$ is derived from the semantics of the components in $\mathcal{U}$ in the same way as the semantics of a composed component is derived from the semantics of its subcomponents (see Section 4.1.5), but only the channels connected to the components in $\mathcal{U}$ are considered.

An *idle step* of $\mathcal{U}$ is a pair (β, β') of valuations of $V_{\mathcal{U}}$ with $T_{\mathcal{U}}(\beta, \beta')$ such that $\forall v \in V_{\mathcal{U}} : v \in \mathsf{LocVar}^{\mathcal{M}'} \Rightarrow (\beta'(v) = \beta(v))$ (the local variables and control states keep their values) and $\forall v \in V_{\mathcal{U}} : v \in \mathsf{outPorts}^{\mathcal{M}'}(\mathsf{Main}) \Rightarrow \beta'(v) = \perp$ (the output ports are cleared).

A *waiting state* of $\mathcal{U}$ is a valuation β of $V_{\mathcal{U}}$ such that if there are no inputs from the intruder, for all valuations β' of $V_{\mathcal{U}}$ with $T_{\mathcal{U}}(\beta, \beta')$, (β, β') is an idle step.

Assume $\mathcal{U}$ is in a waiting state and receives a message from the intruder via one of its input ports. We call $\mathcal{U}$ a *request/reply subsystem*, if in this case

- $\mathcal{U}$ always returns to a waiting state in a finite number of execution steps,
- before returning to a waiting state, $\mathcal{U}$ sends at most one message back to the intruder, and
- the behaviour of $\mathcal{U}$ is independent of further messages received from the intruder during these execution steps.

We call such an interaction a *request/reply interaction.* A threat scenario has a *request/reply architecture* from the point of view of the intruder if it can be partitioned into the Intruder component and a number of request/reply subsystems. In addition, we require that initially, all but one of the subsystems are in a waiting state. One of the subsystems may send a message to the intruder independently of messages received from it, before entering a waiting state as well.

Request/reply architectures are typical for models of transaction protocols. The threat scenarios of all systems we considered have a request/reply architecture, including the bank application described in Section 4.3. Besides, if the transferred messages have a fixed maximum length and a bounded number of possible values per data element (as in our examples), using non-recursive data types and functions is sufficient. In this case, one can also determine an upper bound for the number of variables needed by the intruder, given by the number of different values of the respective type. The number of variables in the store of the intruder can be reduced to speed up verification by model checking, at the cost of having to justify that the reduced number of variables is still sufficient. As security verification is only one of the activities dealt with in our methodology, the closer examination of such justifications is out of scope of this thesis, as well as the examination of the possible use of infinite-state model checking techniques to enlarge the class of systems for which completeness of the security verification can be guaranteed.

Theorem 5.3.7. *The specialised intruder model described in Section 5.3.2 is complete under the conditions (1)–(4) given above.*

Proof Sketch To prove Theorem 5.3.7, we first show that the generic intruder is a refinement of the specialised intruder, but only for particular computations we call attacks with delay N.

Definition 5.3.8. A computation σ of the intruder component is an *attack with delay* N, if

- at state $\sigma(0)$, no messages are sent or received by the intruder (i.e., the input and output ports are empty) and σ^1 is an attack with delay N, or
- there are $n, n' \geq N$ such that at state $\sigma(0)$, no messages are sent and exactly one message is received by the intruder, at states $\sigma(i)$ with $1 \leq i < n$ no messages are sent or received, at states $\sigma(i)$ with $n \leq i < n'$, no message is received and at most one message is sent, and $\sigma^{n'}$ is again an attack with delay N.

This means that after receiving a message, the intruder does not receive more messages or send out messages for at least N clock ticks, which gives him time to store and analyse the received message, and that the intruder only either sends or receives at most one message at a time.

Theorem 5.3.9. *Assume the threat scenario fulfils the conditions (1) and (2) given above for Theorem 5.3.7. Let* $\sigma^{\mathsf{G}} = [\beta_0^{\mathsf{G}}, \beta_1^{\mathsf{G}}, \ldots]$ *be a computation of the generic intruder. If* σ^{G} *is an attack with delay* N *(for a certain value of* N *dependent on the specialised intruder model), then there is a computation* $\sigma^{\mathsf{S}} = [\beta_0^{\mathsf{S}}, \beta_1^{\mathsf{S}}, \ldots]$ *of the specialised intruder, where for all* $j \geq 0$ *:*

(1) $\forall p_{ch;\mathsf{in}}^{\mathsf{Intr}} \in \mathsf{IntrInPort}(\mathcal{M}) : \beta_j^{\mathsf{S}}(p_{ch;\mathsf{in}}^{\mathsf{Intr}}) = \beta_j^{\mathsf{G}}(p_{ch;\mathsf{in}}^{\mathsf{Intr}})$

(2) $\forall p_{ch;\mathsf{out}}^{\mathsf{Intr}} \in \mathsf{IntrOutPort}(\mathcal{M}) : \beta_j^{\mathsf{S}}(p_{ch;\mathsf{out}}^{\mathsf{Intr}}) = \beta_j^{\mathsf{G}}(p_{ch;\mathsf{out}}^{\mathsf{Intr}})$

(3) $\mathsf{derivable}(\bigcup_{i,\mathsf{type}^k} \beta_j^{\mathsf{S}}(\mathsf{store}_{\mathsf{type}^k;i}) \cup \beta_j^{\mathsf{S}}(\mathsf{store}) \cup \mathsf{IK}(\mathcal{M})) \supseteq \beta_j^{\mathsf{G}}(\mathsf{K}_{\mathsf{Intr}})$

This means that the corresponding computation σ^{S} *has the same input/output behaviour as* σ^{G} *at each execution step and the knowledge of the specialised intruder is at least large enough to derive the values in the knowledge of the generic intruder, possibly with the help of values in the initial intruder knowledge.*

In addition, given $x \in \mathsf{Value}$ *with* $x \in \beta_j^{\mathsf{G}}(\mathsf{K}_{\mathsf{Intr}})$ *for some* j*,* σ^{S} *can be chosen such that there is a* j' *such that* $x \in \bigcup_{i,\mathsf{type}^k} \beta_{j'}^{\mathsf{S}}(\mathsf{store}_{\mathsf{type}^k;i}) \cup \beta_{j'}^{\mathsf{S}}(\mathsf{store})$*. Thus, if the generic intruder knows* x *at some step in* σ^{G}*, so does the specialised intruder at some (possibly later) step in* σ^{S}*.*

We proved Theorem 5.3.9 by induction over j under consideration of the fact that σ^{G} is an attack with delay N, by giving for each step of σ^{G} corresponding transitions of the Medium and FakeStore components of the specialised intruder. In particular, during the delay between receiving messages and sending messages, enough Analz and Synth transitions can be executed such that the FakeStore component can generate any messages that can be derived from the previous knowledge and the messages received. We omit the full proof, because it depends on specification details of the generated automata.

To prove Theorem 5.3.7, we derive from a computation of the generic threat scenario a computation where the intruder behaviour is an attack with delay N.

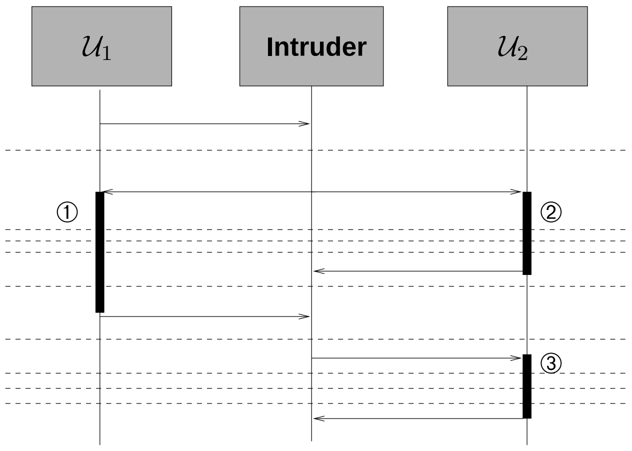
Theorem 5.3.10. *Let* σ *with* $\Psi^{\mathsf{G}}(\sigma)$ *be a computation of the generic threat scenario, which has a request/reply architecture. Let* $\mathcal{U}_1, \ldots, \mathcal{U}_{n_\mathcal{U}}$ *be the request/reply subsystems of the generic threat scenario with semantics* $[\![\mathcal{U}_i]\!] = (V_{\mathcal{U}_i}, I_{\mathcal{U}_i}, T_{\mathcal{U}_i})$*. Then there is a computation* $\hat{\sigma}$ *with* $\Psi^{\mathsf{G}}(\hat{\sigma})$ *of the generic threat scenario such that* $\hat{\sigma}|_{V_{\mathsf{Intr}}^{\mathsf{G}}}$ *is an attack with delay* N*, and for all* $\mathcal{U}_i$*,* $\sigma|_{V_{\mathcal{U}_i}}$ *can be derived from* $\hat{\sigma}|_{V_{\mathcal{U}_i}}$ *only by removing idle steps.*

In addition, if pr *is an instance of one of the patterns for common security requirements described in Section 4.2.2 and* $\neg[\![pr]\!]^{\mathsf{G}}(\sigma)$ *is true (i.e.,* σ *violates* pr*),* *then there is a computation $\hat{\sigma}$ fulfilling the above properties such that $\neg[\![pr]\!]^{\mathsf{G}}(\hat{\sigma})$ is true as well.*

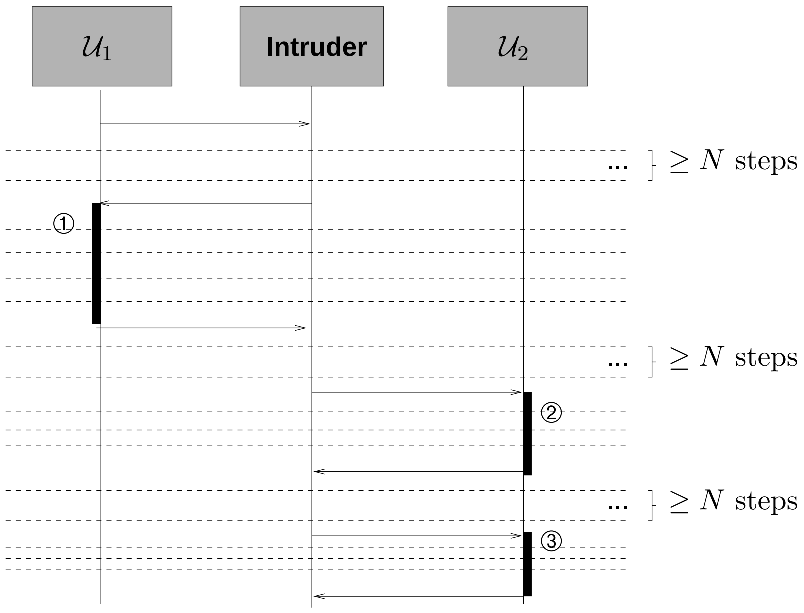
Theorem 5.3.10 can be proved by showing how to derive $\hat{\sigma}$ from σ. The basic idea of the proof is illustrated in Figure 5.11. Figure 5.11(a) shows an example for an original computation σ. $\sigma|_{V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}}$ is not an attack with delay N (for any N): for example, at the beginning the intruder sends out two messages at the same time, only one step after having received a message from $\mathcal{U}_1$. The request/reply interactions of $\mathcal{U}_1$ and $\mathcal{U}_2$ are marked with ①, ②, ③: they consist of a message reception, some computation and the sending of a message. Other messages received from the intruder during such an interaction are ignored. We can observe that interactions ① and ② are only dependent on the intruder having received the initial message from $\mathcal{U}_1$. Interaction ③ is additionally dependent on the intruder having received the reply messages from interactions ① and ②. Thus, by moving the request/reply interactions under consideration of their dependencies by introducing delays at waiting states, a computation $\hat{\sigma}$ can be derived which is depicted in Figure 5.11(b), where only a single interaction is processed at a time and in between, the request/reply subsystems perform idle steps. If enough idle steps are performed between the interactions, $\sigma|_{V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}}$ is an attack with delay N for any fixed N.

In addition, the violation of security requirements conforming to patterns of the form described in Section 4.2.2 can be preserved. For properties of the form $\mathsf{never}(\mathsf{knowsIntruder}(x))$ this is the case because if the intruder knows x at some step in σ, he also knows x at some (possibly later) step in $\hat{\sigma}$, because just delays are introduced. Violation of properties of the form $\mathsf{precedes}(pr_1, pr_2)$ and $\mathsf{leadsto}(pr_1, pr_2)$ can be preserved because the order of the request/reply interactions can be chosen accordingly (for instance, choosing the order ①–②–③ in the above example, if a $\mathsf{precedes}$ property is violated because some state in ① occurs without some other state having occurred in ②).

Now, let us consider again Definition 5.3.5: by applying Theorem 5.3.10 to a computation σ with $\Psi^{\mathsf{G}}(\sigma) \wedge \neg[\![pr]\!]^{\mathsf{G}}(\sigma)$, we obtain a computation $\hat{\sigma}$ with $\Psi^{\mathsf{G}}(\hat{\sigma}) \wedge \neg[\![pr]\!](\hat{\sigma})$, such that $\sigma^{\mathsf{G}} := \hat{\sigma}|_{V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}}$ is an attack with delay N. According to Theorem 5.3.9, for σ^{G} there is a computation σ^{S} for which $\Psi_{[\![\mathsf{Intruder}]\!]_{\mathcal{M}'}}(\sigma^{\mathsf{S}})$ is true, and σ^{G} and σ^{S} have equal input/output behaviour. Thus, as in the proof for Theorem 5.3.3, we can replace in $\hat{\sigma}$ the valuations of the variables $V^{\mathsf{G}}_{\mathcal{M}';\mathsf{Intruder}}$ of the generic intruder by the valuations of the variables $V_{\mathcal{M}';\mathsf{Intruder}}$ of the specialised intruder in σ^{S} and obtain a trace σ' for which $\Psi^{\mathsf{S}}(\sigma')$ is true. pr is also violated in σ': for the $\mathsf{precedes}$ and $\mathsf{leadsto}$ patterns, this is true because they do not refer to the state of the intruder. A security requirement of the form $\mathsf{never}(\mathsf{learnsIntruder}(x))$ is violated in $\hat{\sigma}$ if x is in the intruder knowledge at some step in the computation. In this case, by Theorem 5.3.9 (last paragraph), we can conclude that it is also



(a) Original Computation σ



(b) Delayed Computation $\hat{\sigma}$

Figure 5.11.: Deriving an Attack with Delay N

violated in σ'. □

5.3.4. Optimisations

The implementation of the threat scenario generation for AutoFocus/Quest models includes a number of optimisations of the specialised intruder model described in Section 5.3.2 to improve model checking performance, described as follows.

Immediate Ports. In the Intruder component, *immediate ports* are used to send messages from Medium to FakeStore or to the connected components in the threat scenario. Immediate ports are a recent feature of AutoFocus/Quest making it possible to send messages without the usual delay of one clock tick. This reduces the overall delay introduced by the intruder into the communication and thus the length of runs of transactions in presence of the intruder — which considerably speeds up the model checking process. Note that even with immediate ports it is not possible to reduce the delay introduced by the intruder to zero in the described specialised intruder model, as for the analysis and synthesis of messages, in general a sequence of several transitions must be fired in the FakeStore components. Besides, no circular dependencies may be introduced by immediate ports in AutoFocus/Quest, such as two components sending messages to each other via immediate ports.

Combination of Synth/Analz Transitions. The specialised intruder model requires a number of variables $\mathsf{store}_{\mathsf{type}^k;i}$ for every type type^k of subterms of the type TMessage. In the implementation, several Analz/Synth transitions can be combined into one, which makes it possible to omit variables for intermediate results. As an example, consider the message $m = \mathsf{Data}(\mathsf{Sign}(\mathsf{SK}(\mathsf{C}), \mathsf{NonceS}), \mathsf{SignCert}(\ldots))$ sent from Client to Webserver during a transaction of the bank application (see the EET in Figure 4.21 on page 74). Instead of extracting the nonce NonceS by executing two Analz transitions (first, extract Sign(SK(C), NonceS) from m and store it into a variable of type TSign, and then extract NonceS and store it into a variable of type TNumberKey), the intermediate variable of type TSign is omitted and NonceS is extracted via a combined Analz/Analz transition in one step. A Synth transition generating a value of type TMessage from values of type TSign and TSignCert directly extracts the value of type TSign from m, stored in a variable of type TMessage (combined Synth/Analz transition).

Combining Synth/Analz transitions reduces the number of variables of the intruder at the cost of increasing the number (as all combinations have to be taken into account) and complexity of the transitions. Because of this tradeoff, the combination feature must be applied with care and requires experimentation by the user to find an appropriate balance with respect to model checking performance.

Omitting Unnecessary Variables and Transitions. Depending on the threat scenario, some of the transitions and variables in the schema for the specialised intruder model are omitted, which reduces the complexity of the model. For example, if the intruder has no replay capabilities, the $\mathsf{M}_{p;i}$ variables of the Medium component are left out, or if the intruder can always intercept a message from a particular port (InterceptAndStore transition), the corresponding Drop transition is not generated.

Rotating the Stores. For the Medium component, instead of generating a transition InterceptAndStore, Forward, ForwardAndStore, Replay, and Drop for each $\mathsf{M}_{p;i}$ variable (for $i \in \{1, \ldots, n_{\mathsf{M}_p}\}$, where n_{M_p} is the number of variables available to store messages from p), just one transition is generated storing the message in $\mathsf{M}_{p;1}$, and between these transitions, the store can be rotated by the execution of additional "RotateStore" transitions (i.e., $\mathsf{M}_{p;i}$ is copied to $\mathsf{M}_{p;i+1}$ for $i \in \{1, \ldots, n_{\mathsf{M}_p} - 1\}$ and $\mathsf{M}_{p;n_{\mathsf{M}_p}}$ is copied to $\mathsf{M}_{p;1}$). An analogous optimisation is used for the $\mathsf{store}_{\mathsf{type}^k;i}$ variables of the FakeStore component.

5.4. Case Studies

In the following, we describe some applications of threat scenario generation and verification using AutoFocus/Quest.

5.4.1. The Bank Application

Figure 5.12 shows the SSD of the threat scenario generated for the security-enriched model of the bank application described in Section 4.3. Here, we used immediate ports as an optimisation (see Section 5.3.4), indicated by the diamond shapes. The FakeStore component of the intruder has 1 variable for values of type TMessage, 1 variable for values of type TSignCert, 2 variables for values of type TAKey, 1 variable for values of type TNumberKey and 1 variable for values of type TCDataKey. The generated threat scenario consists of 5 atomic components with altogether 22 channels, 11 local variables and 50 transitions.

Verification of Security Requirements

The input file generated for the model checker SMV from this threat scenario defines a state space of 295 bits. The result of checking the security requirements stated in Section 4.3, SR.CONF_CLDATA (client data confidentiality) and SR.AUTH_ORDERS (authenticated orders), is that both properties are true in the

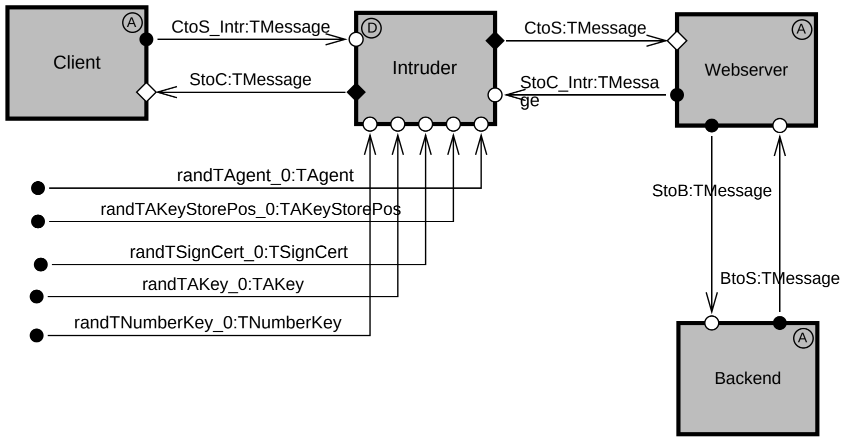


Figure 5.12.: Bank Application: Threat Scenario (SSD)

threat scenario. The required computation time is less than one hour on the used hardware (see Table 5.1 for detailed figures; in the table, the names of the security requirements are abbreviated to CONF and AUTH).

Variation of Threat Scenario: Public Channels

The specification of threats in a security-enriched model from which a threat scenario is automatically generated makes it possible to easily analyse variations of the threat scenario. As an example, consider again the SSD of the security-enriched model of the bank application, depicted in Figure 4.24 on page 78. To analyse the security of the system assuming an unprotected connection between Client and Webserver, we omit the replace annotation and annotate the channels between Client and Webserver with public. Then we again generate a threat scenario and verify the security requirements.

In this case, the input file generated for the model checker SMV defines a state space of 292 bits. The result is that the requirement SR.AUTH_ORDERS is still fulfilled (intuitively, because the intruder cannot forge the client's signature used for authentication), but the requirement SR.CONF_CLDATA is violated (because the customer data sent from the Webserver component to the Client component is not encrypted). For the latter requirement, a counterexample trace is generated showing an interaction in which the intruder can obtain the customer data. The required computation times are approximately 1h (SR.CONF_CLDATA) and 2.5h (SR.AUTH_ORDERS), respectively.

Completeness

The intruder models in the above described threat scenarios for the bank application both fulfil the completeness requirements stated in Section 5.3.3. In particular, the threat scenarios have a request/reply architecture from the point of view of the intruder: except for the initial message to the intruder (directed to the Web server, transition :: CtoS!ClientHello :), the Client component has only transitions of the form ... : StoC?... : CtoS!... : ..., i.e. waiting for a message from the intruder (expected to come from the Web server) and then immediately sending a reply. The Webserver component has a similar behaviour, but can also perform some interaction with the Backend component before sending a reply to the intruder (directed to the Client component). In addition, by a (manual) examination of the transferred messages, we justified that the number of variables in the FakeStore component of the intruder is sufficient to store all values needed by the intruder at the same time. Besides, both security requirements SR.CONF_CLDATA and SR.AUTH_ORDERS are instances of the patterns for common security requirements described in Section 4.2.2.

Therefore, by Theorem 5.3.7, these requirements, which have been verified by the model checker to be true in the specialised threat scenario, are also fulfilled with respect to the generic threat scenario if the threat is considered that the Client component can be replaced by the intruder. The requirement SR. AUTH_ORDERS is also fulfilled with respect to the generic threat scenario in the variation of the threat scenario considering public channels between Client and Webserver.

5.4.2. Further Case Studies

Beside the bank application, threat scenario generation and verification were applied in a number of other case studies, including the Common Electronic Purse Specifications described in Section 4.4. Table 5.1 shows information about the case studies and the resources used by model checking the threat scenario. For each examined threat scenario variation, Table 5.1 shows the number of atomic components (#C), the total number of transitions in the state transition diagrams (#Tr), the size of the global state space in bits (#bits), and the number of reachable states (#states).

In addition, for each considered security requirement (req.) the number of BDD nodes (#knodes, in multiples of 1000) allocated by the model checker (only for non-bounded model checking), the used computation time (t[s], in seconds) and the result of the verification are listed. The performance figures were measured on a computer with a 1.7 GHz Intel Pentium M processor and 512 MB RAM, under the operating system Linux. We used the Cadence version of SMV, which performed considerably better than NuSMV and the original CMU SMV on our examples.

For non-bounded model checking, Cadence SMV was run with BDD variable order sifting turned on (option '-sift') to prevent it from running out of memory too quickly when calculating the BDD for the transition relation. For bounded model checking, the SAT solver Chaff [MMZ+01] was used.

CEPS Load Transaction (CEPSLoad)

The CEPS load transaction described in Section 4.4.1 demonstrates the use of bounded model checking for security verification. Here, the state space was too large (495 bits, corresponding to approximately 10^{149} potentially reachable states) to allow verification using symbolic model checking with SMV within a time limit of 3h.

Using bounded model checking within the same time limit, it could be shown that no counterexample exists for the two non-repudiation requirements SR.NONREP_FAILED_TRANS and SR.NONREP_LOAD_AUTH stated in Section 4.4.1 with lengths up to $l = 25$ and $l = 23$, respectively. For the fair exchange requirement, the reached bound was $l = 51$. Actual execution times are shown in Table 5.1.

While the integrity of the load request sent by the LSAM to the issuer is indeed provided in the CEP specifications (by a message authentication code), this is not the case for the load authorisation sent back to the LSAM. However, to achieve non-repudiation of a failed transaction at the card, the LSAM must rely on the verification of the hash of the random number R_CEP (as the LSAM and the card do not own shared keys). If the integrity assumption for the connection from the issuer to the LSAM is omitted, non-repudiation of a failed transaction at the card cannot be ensured, which is detected by bounded model checking. This indicates a weakness of CEPS pointed out in [Jür04], where it is suggested to include a signature of the hash of R_CEP in the load authorisation message.

Besides, the fair exchange requirement is fulfilled because it refers to the output of the message containing the random number R_CEP by the card (rather than to its reception by the LSAM) and because internal errors within the card were not considered. In the specified threat scenario, it cannot be guaranteed that the LSAM indeed receives this message. In this case, according to the CEP specifications the transition is marked "suspicious" and later settlement between the load acquirer and the issuer is necessary. If the property is changed to require the actual reception of the message containing R_CEP, it is correctly determined by bounded model checking that the property is false and an appropriate counterexample is given.

It is possible to show that when a certain bound l is reached (which depends on the system), verification using bounded model checking is complete, i.e. no counterexamples exist with length greater than l (see [BCC+03] for more detail). We omit such an analysis, as the main aim of the CEPSLoad case study was its use for security test sequence generation (see Section 6.5).

Table 5.1.: Security Verification: Experimental Results

Model	**Variation of Threat Scenario**	**Threat Scenario Complexity**			**Verification**			
		#C	#Tr	#bits #states	req.	#knodes	t[s]	res.
BankApp	replace comp.	5	50	295	CONF	1892	2918	T
				1.26e+13	AUTH	1740	2831	T
BankApp	public ch.	5	48	292	CONF	2387	3825	F
				4.07e+13	AUTH	6124	9408	T
CEPSLoad	public ch.	5	70	495	NONREP_			
				$\leq$1.02e+149	FAILED_TRANS	–	7394	T
					LOAD_AUTH	–	10540	T
					FAIREXC_			
					LACQ	–	5966	T
CEPSPurch	$\text{replace}_{\text{PSAM}} \rightarrow \text{replace}_{\text{Card}}$	4	29	114 7.85e+5	AUTH	189	14	T
CEPSPurch	$\text{replace}_{\text{PSAM}} \rightarrow \text{replace}_{\text{Card}}$, key leakage	4	29	114 6.27e+6	AUTH	279	28	F
CEPSPurch	public ch.	4	24	109 5.82e+5	AUTH	427	18	F
NSPK (original)	public ch.	4	69	173 2.22e+10	AUTH	153	65	F
NSPK (corrected)	public ch.	4	69	187 2.74e+10	AUTH	245	137	T
TLS variant (original)	public ch.	4	33	163 4.83e+9	CONF	2488	502	F
TLS variant (corrected)	public ch.	4	35	195 9.91e+10	CONF	13393	1983	T
PalME	secret/integr.	3	15	101	FAIREXC_			
				1.47e+7	NO_GAIN	109	38	T
					NO_LOSS	283	16	F
PalME	secret/integr., msg. no. 5 not attacked	3	15	101	FAIREXC_			
				1.09e+7	NO_GAIN	127	11	T
					NO_LOSS	386	36	T
SSLServer-Auth	public ch.	4	47	180	CONF	1228	450	T
				1.97e+12	AUTH_SRV	2491	1349	T
					AUTH_CLI	3392	1297	F

CEPS Purchase Transaction (CEPSPurch)

The CEPS purchase case study described in Section 4.4.2 provides a good example for the consideration of complex threats by manually editing the generated threat scenario. We first considered a threat scenario where the intruder makes a POS device publicly available which only communicates with the card (to receive transaction information) and returns the card with an error message without actually having completed a transaction. Then the intruder uses the obtained information to attack a merchant's POS device by buying goods with transaction messages signed by the earlier attacked card. If the attack succeeds, the attacker terminal or card do not show up in the audit trail, so the attacker cannot be made responsible. For security verification, in the security-enriched model we annotated both the Card and PSAM components with replace, and automatically generated a threat scenario. By deleting in the Medium component of the intruder all transitions from state Init to state Main but the transition "$:::\mathsf{replace}_{\mathsf{PSAM}} = \mathsf{True}; \mathsf{replace}_{\mathsf{Card}} = \mathsf{False}$", it is ensured that initially, the PSAM is replaced by the intruder. By adding a transition "$:::\mathsf{replace}_{\mathsf{Card}} = \mathsf{True}; \mathsf{replace}_{\mathsf{PSAM}} = \mathsf{False}$" from state Main to itself, we specified that at any time the intruder can move to a state where the Card is replaced by the intruder (i.e., the intruder uses the information obtained in interaction with a card to attack a POS device). In this threat scenario, the authentication requirement is fulfilled (see Table 5.1, variation "$\mathsf{replace}_{\mathsf{PSAM}} \rightarrow \mathsf{replace}_{\mathsf{Card}}$").

In addition, we examined the effect of key leakage: if the intruder gets to know the secret keys of the card and the PSAM (in the model reflected by a modification of the knowsIntruder function), he can authenticate himself to the PSAM. The model checker correctly indicates this as violation of the security requirement and outputs a corresponding counterexample (variant "$\mathsf{replace}_{\mathsf{PSAM}} \rightarrow \mathsf{replace}_{\mathsf{Card}}$, key leakage" in Table 5.1). In the same way, it can be determined that *both* secret keys have to leak to make an unauthorised authentication possible.

Finally, we considered a threat scenario in which the intruder can communicate both with the card and the PSAM at the same time (for example, if CEPS is used over the Internet, or if he in some way manages to set up a synchronised communication link between an attacked card and an attacked PSAM). For an analysis, the channels between the PSAM and the card are annotated with public (variant "public ch." in Table 5.1). In this case, the authentication requirement can be violated by a straightforward man-in-the-middle attack, in which the intruder acts as a relay between the card and the PSAM. Note that this vulnerability crucially depends on the assumed threat scenario. However, the CEPS consortium also intends to offer transactions over the Internet. The chairman of the CEPS security working group has been informed and acknowledged the weakness.

An earlier version of our analysis of the CEPS purchase transaction using AutoFocus/Quest, without the application of automatic threat scenario generation based on annotations, was published in [JW01b] (co-authored by J. Jürjens). [Jür04] contains a manual analysis of the CEPS purchase transaction using UMLsec.

Needham-Schroeder Public Key Protocol (NSPK)

We reproduced Lowe's well-known analysis of the Needham-Schroeder public key (NSPK) protocol [Low96] using AutoFocus/Quest threat scenario generation and verification. The SSD of the security-enriched model of NSPK consists of two components Initiator and Responder connected via public channels. On reception of an external $\mathsf{InitI}(x)$ message via a channel Init, the initiator runs the NSPK protocol with x as the assumed responder. $x \in \mathsf{Values(TAgent)} = \{\mathsf{A}, \mathsf{B}, \mathsf{I}, \mathsf{EmptyTAgent}\}$ stands for an agent name, where the agent names A and B are assigned to the initiator and the responder, and the agent name I is assigned to the intruder.

Initiator and Responder report the successful completion of the protocol by sending $\mathsf{Session}(x, y)$ messages to the environment via channels SessionI and SessionR, where x and y are the agent names between which a session is supposed to have been established.

The (client) authentication requirement considered by Lowe is that the responder B must only believe a session has been established with an initiator A if the initiator indeed requested such a session. In the model, this is formally stated as

$$\mathsf{SR.AUTH} \equiv \mathsf{precedes}(\mathsf{Init!InitI}(\mathsf{B}), \mathsf{SessionR!Session}(\mathsf{A}, \mathsf{B}))$$

As shown by Lowe, the specified authentication requirement is false in the original version of the Needham-Schroeder public key protocol. Threat scenario generation and verification using AutoFocus/Quest detects this violation and produces the correct counterexample (see Table 5.1, model "NSPK (original)").

Lowe also presents a correction of the protocol. If the correction is incorporated into the AutoFocus/Quest model, the authentication requirement is reported by the model checker to be true in the corresponding threat scenario (see Table 5.1, model "NSPK (corrected)").

Variant of TLS

TLS (Transport Layer Security) [DA99] is a protocol widely used in Internet applications to establish secure connections. In [Jür01], a variant of TLS proposed by [APS99] was analysed for confidentiality of the transmitted messages using formal methods (specifically, the formal method Focus). A flaw was discovered by manual analysis and a fix was proposed and proven correct.

We translated the formal model presented in [Jür01] to a security-enriched AutoFocus/Quest model and carried out the same analysis, supported by automatic threat scenario generation and verification. Data on the models and results are reported in Table 5.1.

In [Jür04], the TLS protocol is (manually) analysed using UMLsec.

PalME Purse-To-Purse Transaction

PalME (secure Palm-based Money Exchange) was a student project supervised by a team of research assistants (including the author) at Technische Universität München aimed at the application of a new development methodology based on the Common Criteria for security evaluation to the development of a secure electronic purse application on Palm pilot handhelds. The PalME application features direct transfer of electronic money between electronic purses (purse-to-purse transactions) via an infrared connection. For the security-critical part of the PalME purse-to-purse transaction protocol, a formal specification in AutoFocus/Quest was produced and used for the verification of security requirements, which would have been demanded for an evaluation at the highest assurance level of the CC, EAL 7.

The PalME protocol model assumes confidential and integrity-protected communication between the electronic purses (but no replay protection). In the original version of the protocol model, the behaviour of the intruder in the threat scenario was specified manually under consideration of these assumptions. In a version we adapted to the threat modelling concepts described in Section 4.2, the channels between the two purses are marked with secret and integrity. Shared-key encryption was applied in the implementation to fulfil the stated assumptions.

In addition to the intruder, the threat scenario consists of components for the payer and the payee. During the course of a transaction, the payer and the payee write their states to their logs, which are assumed to be secure and are modelled by channels LogPayer and LogPayee. The main security requirement was fair exchange, i.e. if electronic value is credited to the payee's purse, it must be debited from the payer's purse (no gain of electronic value), and vice versa (no loss of electronic value). Formally, these properties are stated as

$$\mathsf{SR.FAIREXC\_NO\_GAIN} \equiv \mathsf{precedes}(\mathsf{is\_Debit}(\mathsf{LogPayer}), \mathsf{is\_Credit}(\mathsf{LogPayee}))$$

and

$$\mathsf{SR.FAIREXC\_NO\_LOSS} \equiv \mathsf{leadsto}(\mathsf{is\_Debit}(\mathsf{LogPayer}), \mathsf{is\_Credit}(\mathsf{LogPayee}))$$

respectively.

The result of the analysis (see Table 5.1) was that gain of electronic money is indeed prevented (the payer's card is always debited before the payee's card is credited). Loss of electronic money could not be prevented in the general threat scenario, as the intruder can block messages at any time. However, loss of money is only possible in one particular scenario (the intruder intercepts a certain message of the transaction). If this capability of the intruder is ruled out by slightly modifying the threat scenario (see variant “msg. no. 5 not attacked” in Table 5.1), the second security requirement is fulfilled as well. The mentioned situation can be detected from the log entries and thus the money can be refunded by the card issuer.

For more information about the PalME case study, the formal analysis and the applied development process, see [Pal01, VWW02].

SSL Protocol with Server Authentication (SSLServerAuth)

As part of a case study for the modelling and verification of layered protocols, we built a simplified model of the SSL protocol [FKK96] with server authentication. The model and a security analysis of it are described in more detail in Section 7.4.2. Performance figures are included in Table 5.1.

5.5. Related Work

Threat Scenarios

[Lot97] describes a threat scenario based approach to the development of security-critical systems in terms of the formal language Focus, making an explicit distinction between a system design and derived threat scenarios formulated in the same language. Patterns for the construction of typical threat scenarios are given, whose (manual) application is left to the developer.

The general concept of using threat scenarios (there called trust models) for the analysis of security-critical systems is also outlined in [Pfi98], in the formal framework of communicating automata.

In the UMLsec approach [Jür04], the security extensions of UML define the capabilities of an intruder specified in the form of an abstract state machine communicating with other abstract state machines that form the semantical model of the UML specification of a system. A threat scenario, in the sense of a model of a system under attack in UML itself, is not available.

Security Protocol Verification

Research has produced a large number of approaches for the verification of security protocols based on formal models. The foundation of these works was laid by a paper of Dolev and Yao on a generic abstract intruder model for public-key protocols [DY83]. Overviews over approaches to security protocol verification can be found in [GSG99] or [RSG$^+$00]. [GSG99] classifies these approaches into the following three categories:

Inference-Construction Methods use modal logics to model knowledge and belief and their evolution during a protocol run. The most famous example is the BAN logic (named after their creators Burrows, Abadi and Needham). Possible logic statements in BAN include "P sees X" (i.e. P has received X), "P said X", "P believes X" (P has good reason to believe X is true), "X is fresh", and "K is a good key for communication between P and Q". Inferences allow the deduction of new statements — for example, if P receives X encrypted under the key K, and he also believes that K is a good key for communication with Q, then P believes Q said X. The necessary idealisation of messages as logic statements is problematic because of the unclear semantics and the peril of misinterpretation of the results. Furthermore,the BAN logic is restricted to the analysis of authentication protocols. There are various successors of the BAN logic, such as GNY, AT, SvO, and AUTLOG.

Attack-Construction Methods model the behaviour of the agents together with an attacker. The possible behaviours of the model are then searched for violations of security requirements using model checking. Examples are FDM and Ina Jo, NRL Analyzer, Murphi, CSP, CCS, LOTOS, Astral, and Interrogator. In general, in these approaches the intruder model is built manually in the specification language used for the analysis. Protocol specification languages such as CASPER [Low98] or CAPSL [Mil05] allow to abstractly model a protocol and translate the model to low-level specifications, for instance in CSP or a functional programming language. CASPER and CAPSL are restricted to the specification of authentication protocols with a fixed interaction schema. An example for a security verification tool based on a high-level protocol specification language (called HLPSL) similar to CAPSL is the AVISS tool [AVB$^+$02].

Proof-Construction Methods are based on a similar concept as attack-construction methods but use theorem proving to show if the security requirements are fulfilled. Examples are the inductive approach by Paulson, Strand Spaces, the Spi calculus, or a protocol security theory by Snekkenes using Higher Order Logic (HOL). Interacting state machines [OL02] is a proof construction method based on the theorem prover Isabelle that uses AutoFocus/Quest as a graphical representation (but a different semantical model). Proof-construction methods cannot be fully automated and thus their utilisation requires expert knowledge and is fairly time consuming.

Our application of AutoFocus/Quest to the verification of security protocols belongs to the category of attack-construction methods. The use of Synth/Analz transitions in the FakeStore component was inspired by [Pau98].

Completeness

Examples for work on the justification of the completeness of finite threat scenarios with respect to more general threats (related to our justification of the specialised intruder model given in Section 5.3.3) are [Low99, Sto02]. These approaches focus on the justification that the security analysis of a bounded number of parallel protocol executions (in [Low99], one honest agent per role performing one run) is sufficient to guarantee the security of a protocol in the general case. In contrast, we focus on the analysis of single protocol runs, but the results from these works could be adapted to our examples. However the conditions for the justifications are comparatively restrictive. The author is not aware of work justifying the completeness of intruder models in synchronous, globally clocked systems where the intruder takes a possibly observable number of time ticks for his actions. [GHJW03] gives an informal justification of the completeness of a manually constructed intruder model for the bank application analysed in Section 5.4.1.

5.6. Summary and Discussion

A **threat scenario** is a model of a system under attack. We explained the role of threat scenarios in model-based development of security-critical systems and showed how threat scenarios can be automatically generated by applying a model transformation to security-enriched models. We refer to this model transformation as **threat scenario generation**.

We presented a respective transformation for the security-enriched AutoFocus/Quest models introduced in Chapter 4. The generated threat scenario has a structural aspect (describing *where* the intruder can interfere with the system) and a behavioural one (describing *how* he can do so). An outstanding feature of our approach is that we both deal with a **generic intruder model** on the semantical level and **specialised intruder models** on the conceptual level. The generic intruder model allows a concise, generic specification of the intruder behaviour, which is independent of idiosyncrasies of the used specification language on the conceptual level (in our case, AutoFocus/Quest) and well suited for proof purposes. On the other hand, the result of the application of threat scenario generation includes a specialised intruder model given in the same specification language as the original security-enriched model. Therefore, all available features of the used tool can be applied, such as simulation, verification, or test case generation. Besides, the generated intruder model can be easily manually edited to analyse complex threats.

We gave a formal specification of a generic intruder model for security-enriched models in AutoFocus/Quest, in the form of a discrete system. All threats and assumptions defined in Chapter 4 were considered. Besides, we showed how to generate a specialised intruder model in AutoFocus/Quest consisting of communicating state machines that is tailored to the automated **verification of security requirements** using the translation of AutoFocus/Quest models to the input language of the **model checker SMV**.

The specification of the intruder model at both the semantical and the conceptual level requires soundness and completeness justifications. We proved the **soundness** of the specialised intruder model, in the sense that the specialised intruder model does not allow more attacks than the generic one. On the other hand, a specialised intruder model has the **completeness** property if whenever the generic intruder can perform a certain attack, the specialised intruder can do so as well. Because of the requirements involved by the use of the model checking feature of AutoFocus/Quest (and by the use of finite-state model checking in general), the specialised intruder model is not complete for all possible threat scenarios and security requirements. In particular, the specialised intruder requires a number of execution steps to process messages, thus introducing delay into the communication. The specialised intruder model is complete under a number of conditions — most importantly that the threat scenario is finite-state and has a request/reply architecture. We formally stated these conditions and sketched the proof. Note that there is a trade-off between the reached level of completeness and the complexity of the specialised intruder model.

The models we examined are restricted to one transaction between the involved parties (and the intruder). To verify the security requirements with respect to a several transactions running in parallel, the respective parts of the model would have to be copied. One could also adapt approaches such [Low99, Sto02] to justify that the consideration of a small number of transactions is sufficient to guarantee the security requirements with respect to arbitrarily many transitions running in parallel.

In general, specialised tools for the verification of security protocols achieve better performance than the described approach for security verification by model checking a generated threat scenario using the SMV connection of AutoFocus/Quest. This is due to the overhead introduced by the use of the general-purpose specification language AutoFocus/Quest for the specification of the considered system and the intruder. Increasing the efficiency and scope of verification approaches for security-critical systems is not the focus of this work, but rather the integration of security verification into model-based development, the consideration of a variety of possible threats and assumptions, and the use of threat scenarios for test sequence generation. The efficiency of security verification using our model-based approach can be improved both by optimising the generated threat scenario and by optimising the SMV connection or possibly by implementing connections to alternative model checkers. We described and implemented several optimisations of the generated threat scenario. The model checking connection is independent of security-specific issues. Currently, connections of AutoFocus/Quest to the model checkers Spin and SAL (which also supports infinite-state model checking) are developed.

The threat scenario generation transformation has been implemented together with the additional optimisations as a plugin for the AutoFocus/Quest tool. We demonstrated the successful application of security verification using threat scenario generation at the example of a number of **case studies** including the bank application presented in Section 4.3.

6. Model-Based Security Testing

In this chapter, we show how a threat scenario generated from a security-enriched model of a system can be used to gain confidence in the security of its implementation. For this purpose, we adapt methods from classical specification-based testing to the domain of security-critical systems. In particular, we give strategies for the selection of test sequences likely to detect possible vulnerabilities and present an approach for the translation of abstract test sequences derived from threat scenarios to concrete test sequences that can be applied to an existing implementation.

This chapter is structured as follows. First, we give an introduction to security testing in Section 6.1. In Section 6.2, we explain the basic concepts of model-based test sequence generation and its integration into the AutoFocus/Quest tool set. Section 6.3 is devoted to the description of criteria for the generation of test sequences for security-critical systems. An approach for the concretisation of abstract test sequences is presented in Section 6.4. We demonstrate our ideas in Section 6.5 at the example of the load transaction in the Common Electronic Purse Specifications. References to related work are given in Section 6.6, and Section 6.7 contains a summary and discussion of the presented results.

Part of the presented work has been published in [JW01a, WJ02].

6.1. Security Testing

Testing is the process of exercising an implementation to verify that it satisfies the specified requirements and to detect faults (after [IE92]). Testing is indispensable even if a formal specification is available, to gain confidence that the implementation conforms to its specification with respect to the aspects of the system reflected in the specification. A mathematical proof of this conformance statement can usually not be given, because it would require a formal semantics of the implementation language and of the environment the implementation runs in (including the operating system and even the hardware).[1] For this reason, also code generated from a specification should in general be distrusted and therefore still be tested, even if code generation is at all possible because the used modelling tool offers this feature and the generated code is suitable for the environment (with respect to program-

[1] A first attempt to achieve such conformance statements for an industrial-size application example is currently performed within the Verisoft project [Ver04].

ming language, size, performance, completeness, interfaces to other parts of the system, etc.). Finally, testing is necessary to check that no defects were introduced at lower levels of abstraction, which are not covered by the specification.

Security testing means testing an implementation with the aim to show that it fulfils the stated security requirements and that the security-related functionality is implemented correctly. Security testing methodologies must address the following particularities:

Consideration of attacks Security testing must be carried out under consideration of attacks, i.e. with respect to a malicious system environment.

Emphasis on completeness Completeness of the tests is particularly important, because untested parts of a system are likely to contain security-critical bugs or even malicious functionality [McG99]. It must be assumed that even obscure vulnerabilities triggered by input data unlikely to occur in normal operation will eventually be found and exploits will be published. However, exhaustive testing of all possible combinations of input data is not feasible, because of the extremely large number of tests required for this purpose.

Relevance of vulnerabilities on lower abstraction levels Faults on lower levels of abstraction can often be exploited to achieve violations of security requirements relevant at higher levels of abstraction. An example are buffer overflow related faults, which are highly dependent on the environment the implementation is executed in. As lower levels are usually abstracted from in formal specifications, it is especially necessary to consider them during security testing.

Processing of cryptographic data by test infrastructure If cryptography is used in the system to be implemented, the test infrastructure must support the generation and verification of data resulting from the application of cryptographic operations.

Support of testing of security requirements Security requirements are universal properties, i.e. should hold for all possible runs of the implementation. Thus per se they cannot be used to select particular runs to test. Besides, security requirements can refer to the intruder (e.g. confidentiality requirements). A security testing methodology must support testing with respect to such requirements.

Security testing techniques can be roughly classified into two categories, *security functional testing* and *vulnerability testing* (cf. [CB03]).

Security Functional Testing The aim of security functional testing is to test if the security functions (such as access control mechanisms or key generation algorithms) conform to their specifications.

The usual strategy to test conformance is to fix a set of tests and to argue that they are sufficient to demonstrate that the system operates according to the specification with the help of *coverage criteria*. Examples for coverage criteria are that all paths of the flowgraph of the program are traversed (path testing) or that at least one input value is chosen from each member of a fixed partition of the corresponding data type (domain testing). [Bei90] contains a comprehensive overview over such coverage criteria.

Vulnerability Testing The aim of vulnerability testing is to identify faults in the design or implementation of a system that can lead to the violation of security requirements. The common approach to vulnerability testing is to use knowledge of typical vulnerabilities of security-critical systems as a basis for selecting tests.

In *penetration testing* (or red teaming), a team of experts manually tries to break a system. Penetration testing is labour-intensive and therefore expensive. Besides, it is hard to assess the quality of a penetration test, which depends largely on the skill of the employed penetration testing team and on the time available.

A more systematic way of vulnerability testing is the use of *vulnerability scanners*, i.e. automated tools that expose a system to a large number of possible attacks. Vulnerability scanning is fast and cost-effective, as it is automatic and even freeware vulnerability scanners are available. However, vulnerability scanning must rely on the knowledge encoded into the tool, which mostly consists of surface vulnerabilities on low levels of abstraction and does not take into account application-specific security requirements. Moreover, vulnerability scanners tend to produce a considerable amount of false positives which have to be interpreted by a security expert.

Vulnerability testing can be aided by testing manuals, such as the Open Source Security Testing Manual (OSSTM) [Her03].

6.2. Model-Based Testing

The central idea of model-based testing (also called specification-based testing) is to determine tests based on an explicit model of the required behaviour of a system (rather than an implicit one in the human tester's head), which makes the testing process more systematic and introduces potential for automation. There has been extensive research into model-based testing, including [DF93, PS97, HNS97, OXL99, DBG01]; [Pet00] contains an annotated bibliography to a wide range of work.

In the following, we give a short introduction into the main concepts of model-based testing and their formalisation for AutoFocus/Quest models. For more detail, the reader is referred to [Wim00, WLPS00, PLP01, Pre03].

Test Sequences A *test sequence* is a sequence of inputs and expected outputs. In model-based testing, a test sequence consists of inputs and outputs of an execution trace of a model, stating that the implementation should exhibit a corresponding input-output behaviour. For the actual testing, the abstract values of the inputs and outputs given in terms of the data types of the model must be concretised by a test driver. Often, additional conditions are required before a test sequence can be applied, such as a *preamble* having been sent to put the implementation into a particular state. For simplicity, in our treatment we view these conditions as fixed and put the responsibility of ensuring them on the test driver.

In AutoFocus/Quest, a test sequence can be represented by an Extended Event Trace (EET) that only refers to channels connected to a particular component, the *component under test* (CUT). Here, the component under test represents the component in the model that corresponds to the part of the system we want to test — for instance the Card component in the models of the CEPS transactions (see Section 4.4) if the CEPS smart card should be tested. On the semantical level, a test sequence is represented by a finite sequence of valuations of the channels connected to the CUT.

Test Case Specifications A *test case specification* is a requirement that must be satisfied by a test sequence, for example that a certain message is output, a certain control state of a state machine in the model is reached, or a certain transition is fired.

Formally, we define a test case specification as a predicate $\Phi(\sigma)$ over sequences σ of valuations of the state variables of a model. On the syntactical level, in AutoFocus/Quest a test case specification can be stated as an LTL property pr with $\Phi = [\![pr]\!]$.

Testing Criteria and Coverage A *testing criterion* is a rule that leads to a set of test case specifications. We distinguish between *structural testing criteria* and *functional testing criteria.* Structural testing criteria are based on the code, or, in the context of model-based testing, on the structure of the model (for example, one test case specification for each transition in the component under test, stating that this transition should be executed). Functional testing criteria are based on the requirements.

The extent to which a testing criterion is satisfied by a set of test sequences is referred to as *coverage.* Coverage can be measured e.g. by the percentage of satisfiable test case specifications for which test sequences are computed and applied to the implementation.

Test Sequence Generation *Test sequence generation* is the determination of test sequences from a model and a number of test case specifications.

Formally, test sequence generation takes as input the semantics of a model (which can be expressed by the predicate $\Psi(\sigma)$, see Section 4.1.5) and test case specifications $\Phi_i(\sigma)$ and determines finite sequences σ of valuations fulfilling $\Psi(\sigma) \wedge \Phi_i(\sigma)$, i.e. traces that are computations of the model and fulfil the test case specification. Here, the definition of $\Psi(\sigma)$ is extended from infinite to finite sequences by unrolling the transition relation only up to the length of σ. Besides, we define the predicate $\Phi_i(\sigma)$ to be fulfilled for a finite sequence σ if $\Phi_i(\sigma')$ is fulfilled for all infinite sequences σ' that have σ as prefix. Thus, when the system continues to run after a behaviour corresponding to σ has been observed and σ fulfils a certain test case specification, it is guaranteed that the test case specification stays fulfilled. This reflects the fact that because test sequences are finite, one can only test a system for the violation of safety properties (as opposed to liveness properties, see [AS85]). The generated test sequences are the projections of the sequences σ to the channels connected to the component under test.

In AutoFocus/Quest, test sequences can be generated by means of SAT solving or Prolog-based constraint solving. The basic idea of the SAT solving method is to translate $\Psi(\sigma) \wedge \Phi_i(\sigma)$ for sequences σ of a length less than or equal to a fixed bound l to a formula in propositional logic and to use a SAT solver such as Chaff [MMZ$^+$01] to calculate the test sequences. This approach is described in more detail in [WLPS00]. It is closely related to bounded model checking [BCC$^+$03] (but allows some optimisations depending on the test case specifications). Support for bounded model checking has recently been added to Cadence SMV and NuSMV. Therefore, also the translation of AutoFocus/Quest models to the SMV input language described in Section 5.3.1 can be applied for test sequence generation, by using the negations of the $\Phi_i(\sigma)$ as specifications and having Cadence SMV or NuSMV compute counterexamples.

For test sequence generation via Prolog-based constraint solving, the transition relation need not be unrolled to the predicate Ψ. Instead, test sequences are determined based on the characterisations $I(\beta)$ and $T(\beta, \beta')$ of the initial states and the transition relation by using the depth-first search procedure built into Prolog. Advantages of the constraint-based test sequence determination are the support of recursive data types and floating point numbers and optimisations such as guided search for test sequences or the storage of previously visited states using constraints. However, it is less tightly integrated into AutoFocus/Quest and does not support test case specifications using general LTL formulas. The constraint-based test generation for AutoFocus/Quest is described in more detail in [PLP01, Pre03].

Test Drivers and Concretisation A *test driver* is a piece of software that tests an implementation based on a number of test sequences by sending test inputs, receiving test outputs and deciding if the behaviour of the implementation conforms to the test sequence. The test driver is also responsible for the mapping between abstract messages represented in terms of the model and concrete messages sent to and received from the implementation (*concretisation*) and for correctly initialising the implementation before executing a test. The result of the application of a test sequence is a *test verdict*, which can be Pass, Fail, or Inconclusive. Inconclusive is used when neither a Pass nor a Fail can be given.

AutoFocus/Quest supports the development of test drivers by offering a Java-based interface to access the models (including EETs representing test sequences) and a text-based file format to store test sequences.

Nondeterminism If the model contains nondeterministic behaviour, applying the test sequences separately is not sufficient, because for a sequence of test inputs there can be more than one sequence of expected outputs. An early treatment of this problem can be found in [LPvB94], in the context of conformance testing of communicating (non-extended) finite state machines. The basic idea is to apply a sequence of test inputs multiple times and to repeatedly execute the implementation with these test inputs until all possible output sequences are obtained. As the nondeterministic behaviour of the implementation cannot be controlled, the number of executions necessary for this purpose must be fixed by an assumption (*complete testing assumption*).

[Wim00, Pre03] are aimed at test sequence generation from deterministic AutoFocus/Quest models, but also contain first ideas towards the consideration of nondeterminism.

In this work, we focus on the generation and application of *deterministic test sequences*, i.e. test sequences whose expected outputs are the only possible reaction to their inputs [Wim00]. However, we allow nondeterminism on the implementation level. This kind of nondeterminism is an important part of security mechanisms. Examples are the random selection of keys or nonces in security protocols, which we represented as abstract entities in the security-enriched model. Besides, we allow nondeterminism with respect to message parts left out in the model (such as transaction numbers). Implementation-level nondeterminism is handled by the test driver.

6.3. Generation of Test Sequences for Security-Critical Systems

To complement conventional security testing techniques, we take advantage of a security-enriched model created during development. We assume that the security-enriched model is verified secure (i.e. the threat scenario fulfils the security requirements), which can be ensured by the techniques described in Chapter 5.

In principle, general model-based testing criteria can immediately be applied for the generation of test sequences for security tests from the security-enriched model. In the following, we explain how the particularities of security testing listed in Section 6.1 can be addressed by using the additional security-specific information.

Consideration of Attacks

To deal with the consideration of attacks, we explained in Chapter 5 how a threat scenario can be generated from a security-enriched model based on the included security-specific information. Thus, for security testing, it suggests itself to use the threat scenario as the model to compute test sequences from. In this case, the other components in the threat scenario and especially the intruder act as the environment for the component under test. As possible input data, exactly the messages are considered that can indeed be sent to the component under test due to the assumptions reflected in the security-enriched model. For example, test sequences with fake messages using keys the intruder is not assumed to possess are not generated.

Emphasis on Completeness

The required degree of completeness of the security tests must be realised by the choice of structural and/or functional testing criteria. The strongest testing criterion is exhaustive testing, i.e. generating test sequences for all possible input values to the component under test at each execution step. Exhaustive testing is the most desirable testing criterion for security testing, but is only feasible for very abstract models and short interactions (e.g. only one or two execution steps) because of the large number of test sequences to be generated and applied. Otherwise, one must resort to weaker testing criteria, with the aim to achieve the highest possible confidence in the security of the implementation given the available resources.

Various structural testing criteria are described e.g. in [Nta88, Bei90, OXL99]. [Löt03] gives structural testing criteria for models consisting of communicating extended finite state machines, in the context of AutoFocus/Quest. Structural testing criteria can be categorised into control flow oriented, data flow oriented and communication oriented testing criteria.

For state-based systems, control-flow oriented testing criteria refer to the control states and transitions of the state machines. An example is *transition coverage*, stating that for each transition in a particular state machine, there is at least one test sequence that forces this transition to be fired. Transition coverage can be extended to sequences of transitions (e.g., *transition pair coverage*), or additional requirements on the preconditions of the transitions can be added such that selected combinations of truth values of their subterms are exercised (different flavours of *condition coverage*). Data flow oriented testing criteria refer to the definitions and uses of variables (in the sense of assignments and references). An example is *all-defs*, requiring that for each definition of each variable v, at least one definition-free trace leading to a use of v is tested. Finally, communication oriented testing criteria refer to message sending and receiving activities. For example, for an appropriate partition of the set of messages that can be transferred on a channel, one can require that for each equivalence class in the partition, there is a test sequence such that a value from this equivalence class is sent on the channel (*channel coverage* in [Löt03]). *Communication coverage* requires that for each channel, all pairs of sending and receiving actions on this channel are tested.

For security testing, the focus of the testing activities is the security-critical behaviour. This makes it possible to restrict the general coverage criteria described above to the security-critical parts of the model. In the security-enriched AutoFocus/Quest models, the critical annotation was introduced for this purpose: for example, transition coverage can be restricted to those transitions annotated with critical. Due to the involved reduction in the number of test sequences, stronger testing criteria can be applied. To the same end, using the threat scenario for test sequence generation restricts the possible inputs that can appear at the interface of the component under test, as described above under "Consideration of Attacks".

Structural testing criteria do not take into account the security requirements. As generating test sequences to identify possible violations of the security requirements is the aim of vulnerability testing, for vulnerability testing corresponding security-specific functional testing criteria must be defined.

Note that due to the infeasibility of exhaustive testing, in general it is impossible to test an implementation for vulnerabilities introduced through intentional subversion (cf. [Irv00]). For example, suppose that a 128-bit number is received as part of a message to the implementation and a malicious programmer introduced hidden code that performs an action violating a security requirement if the number has a special value he chose. To detect this, basically all 2^{128} possibilities would have to be tested. As pointed out in [Irv00], more subtle subversions are even unlikely to be detected if the code is available to the testers. Intentional subversion must be prevented by a secure development process (e.g. using code reviews and protecting the integrity of the code), but is out of scope of the model-based security testing approach described in this chapter.

Relevance of Vulnerabilities on Lower Abstraction Levels

The approach for the concretisation of test sequences that we will describe in Section 6.4 allows testing implementations with respect to models in which part of the data was omitted or modelled symbolically to make verification feasible. Besides, it ensures that the parts of messages that were abstracted in the model have consistent values. Corresponding faults are automatically detected during the test.

To specifically test an implementation for security-critical faults arising on lower levels of abstraction (such as sending anomalous message parts, which is for example exploited in buffer overflow attacks), modifications of the model are necessary. A way to address this problem is to extend the data type definitions for messages such that anomalous message parts can be represented in an abstract way and mapping these to concrete anomalous message parts (such as overly long bit sequences or bit sequences containing illegal values) within the test driver. Then, test sequences with messages containing such anomalous message parts are generated using appropriate testing criteria and it is checked if they cause the implementation to crash (in which case a possible security hazard was detected). Such an approach (PROTOS) is described in [Kak01] and is supported by our framework. However, in the following we focus on security-critical faults on the level of the model or of the concretisation of its abstract messages.

The above described strategies should be complemented by classical penetration testing to account for application-independent vulnerabilities, e.g. in the operating system or in the configuration of the platform the implementation is installed on.

Processing of Cryptographic Data by Test Infrastructure

In our approach, the extensions to the syntax and semantics of data types and functions of a model (see Section 4.2.6) make it possible to compute test sequences with inputs and outputs that include the application of cryptographic operations. Also, the testing criteria can refer to applications of cryptographic operations and to their arguments, e.g. to compute a test sequence where a particular encrypted value is sent. The generation and verification of concrete cryptographic data corresponding to a particular symbolic representation in the model is handled by the test driver.

Support of Testing of Universal Security Requirements

A security requirement that is given as a universal property cannot be directly used as a test case specification: if a trace violating a universal property is found by the test sequence generator, the model violates this property and must be corrected. On the other hand, if the property is satisfied by all traces, it cannot be used to select relevant ones.

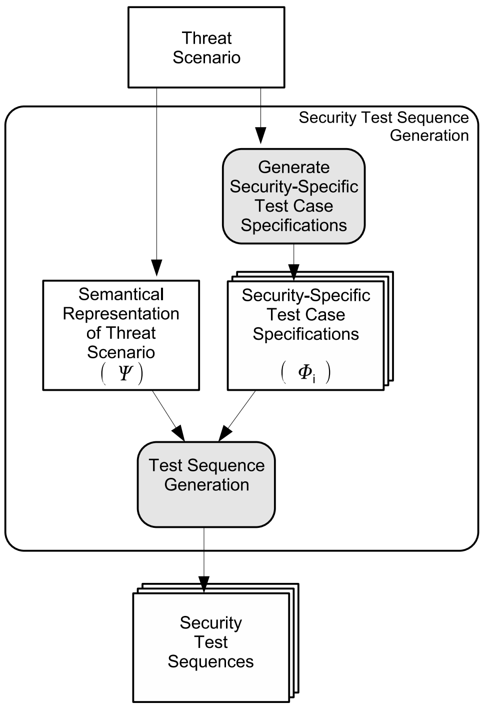

Threat Scenario
Security Test Sequence Generation
Generate Security-Specific Test Case Specifications
Semantical Representation of Threat Scenario (Ψ)
Security-Specific Test Case Specifications (Φi)
Test Sequence Generation
Security Test Sequences

Figure 6.1.: Security Test Sequence Generation Based on Threat Scenario

We present appropriate security-specific functional testing criteria that address this problem in Section 6.3.1 and Section 6.3.2. The fact that security requirements can refer to the intruder is accounted for by using the threat scenario as a basis for test sequence generation.

Process

Figure 6.1 summarises the general process for model-based generation of security test sequences. One input to the test sequence generation is the semantical representation of the (specialised) threat scenario. The other input are security-specific test case specifications, obtained from the threat scenario by the application of testing criteria. For test sequence generation using SAT solving, these are given by the predicates Ψ and Φ_i respectively.

Security-specific adaptations of structural testing criteria by taking into account the critical annotations in the security-enriched model and the threat scenario were discussed above, under "Emphasis on Completeness". In the following, we focus on vulnerability testing and describe two functional testing criteria for the determination of test sequences likely to detect the violation of security requirements.

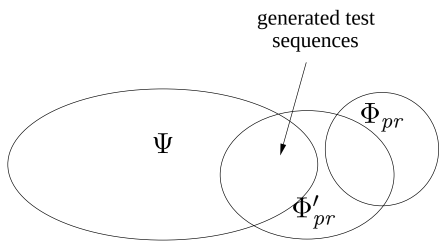


Figure 6.2.: Test Case Specifications for Security Requirements

6.3.1. Derivation of Test Case Specifications from Security Requirements

In this section, we show how a test case specification can be derived from a security requirement by modifying its negation in order to search for traces that represent valid computations of the model but also share relevant properties with traces that violate the security requirement. The underlying assumption is that a test sequence corresponding to a similar behaviour as in cases when a security requirement is violated is likely to detect a violation of the original security requirement in the implementation.

The basic idea of this approach is illustrated in Figure 6.2. Let $\mathcal{M}$ be the threat scenario (i.e., the security-enriched model after application of the threat scenario generation transformation described in Chapter 5) with top-level component Main, and $\Psi = \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}}}$ the characterisation of its set of computations. Let Φ_{pr} be the trace predicate corresponding to the negation of a security requirement pr:

$$\Phi_{pr} = \neg[\![pr]\!], \text{ with } pr \in \mathsf{properties}^{\mathcal{M}}(\mathsf{Main}) \ \wedge\ \mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}}(pr)$$

Because we assume the model is verified secure, Φ_{pr} has no traces in common with the model (characterised by Ψ). To be able to generate test sequences, we derive appropriate test case specifications Φ'_{pr} from Φ_{pr} such that there are both traces in Φ'_{pr} that are possible in the model and traces that violate the security requirement. A special case is weakening the violation of a security requirement (i.e., the set of traces fulfilling Φ_{pr} is included in the set of traces fulfilling Φ'_{pr}).

In the following, we explain how in this manner test case specifications can be derived for the common security requirements listed in Section 4.2.2.

Confidentiality Requirements We specified confidentiality by security requirements of the form $pr = \mathsf{never}(\mathsf{learnedIntruder}(x))$, where $x \in \mathsf{Value}$. By the semantics of properties, we have

$$\Phi_{pr}(\sigma) = \neg[\![pr]\!] = \exists i \geq 0 : [\![\mathsf{learnedIntruder}(x)]\!](\sigma^i)$$

I.e., if pr is violated for a trace σ, then x is contained in the intruder knowledge at some execution step i. Assume $\mathsf{typeof}(x) = \mathsf{type}^k$. We weaken $\Phi_{pr}(\sigma)$ by stating that the intruder knowledge should contain *an arbitrary* message of the same type type^k, which is not previously known by the intruder:

$$\begin{aligned}\Phi'_{pr}(\sigma) = \quad & \exists i \geq 0 : \exists x' \in \mathsf{Value}(\mathsf{type}^k) : \\ & \qquad [\![\mathsf{learnedIntruder}(x')]\!](\sigma^i) \wedge \neg[\![\mathsf{knowsIntruder}(x')]\!](\sigma^i)\end{aligned}$$

$\Phi'_{pr}(\sigma)$ specifies traces where a value similar to x (because it is of the same type) is learned by the intruder, and we use test sequences generated from this specification to check that the intruder does not obtain x instead. If x is a constructor application $x = \mathsf{C}_i^k(x_1, \ldots, x_{n_i^k})$ for $n_i^k > 0$, a stronger similarity can be specified by additionally requiring that x' must also be a constructor application with the same constructor C_i^k, by adding the conjunct $[\![\mathsf{is\_}\ \mathsf{C}_i^k(x')]\!](\sigma^i)$ to $\Phi'_{pr}(\sigma)$. One can proceed by stating that all but one of the arguments of C_i^k in x and x' must be equal, and recursively continue this approach if the arguments of C_i^k are themselves constructor applications.

Example 6.3.1. As an example, consider the security requirement SR.CONF_CLDATA (client data confidentiality) of the bank application case study described in Section 4.3, which is given by $pr = \mathsf{never}(\mathsf{learnedIntruder}(\mathsf{CDataBC}))$. Weakening its negation results in a test case specification where it is required that the intruder learns a not previously known message of type TCDataKey. A test sequence fulfilling this specification is for example a trace in which the intruder learns CDataBA, i.e. his own customer data. During this trace, the intruder authenticates himself to the system using his identity. It can now be verified that indeed the intruder's customer data is transferred, and not the original customer's data, caused by a possible fault in the implementation.

Authenticity / Integrity / Non-Repudiation / Fair Exchange Requirements Often, authenticity, integrity, non-repudiation and fair exchange requirements can be specified using property patterns of the form $pr = \mathsf{precedes}(pr_1, pr_2)$. The traces that violate pr are given by

$$\Phi_{pr}(\sigma) = \neg[\![pr]\!] = \exists i \geq 0 : [\![pr_2]\!](\sigma^i) \wedge \forall j : 0 \leq j \leq i \Rightarrow \neg[\![pr_1]\!](\sigma^j)$$

I.e., pr_2 is fulfilled at some step, but until this step, pr_1 is not fulfilled. The test case specification $\Phi_{pr}(\sigma)$ can be modified in two ways: (1) by deriving a test case specification $\Phi'_{pr}(\sigma)$ requiring that pr_2 is satisfied at some step (to check if pr_1 is fulfilled in an earlier step in the implementation), and (2) by deriving a test case specification $\Phi''^{(l)}_{pr}(\sigma)$ requiring that pr_1 is never fulfilled up to step l, where l is the length of the test sequence (to check if pr_2 is not fulfilled at some step in the implementation). Formally,

$$\Phi'_{pr}(\sigma) = \exists i \geq 0 : [\![pr_2]\!](\sigma^i)$$

and

$$\Phi''^{(l)}_{pr}(\sigma) = \forall j : 0 \leq j \leq l \Rightarrow \neg[\![pr_1]\!](\sigma^j)$$

$\Phi'(\sigma)$ is a weakening of $\Phi(\sigma)$, while $\Phi''^{(l)}(\sigma)$ is not.

Example 6.3.2. As an example, consider the security requirement SR.AUTH_ORDERS (authenticated orders) of the bank application case study, which is given by the property pr = precedes(State$_{\mathsf{Client}}$ == AwaitForm, (State$_{\mathsf{Webserver}}$ == ClientAuthenticated) && (Webserver.localGID == GIDC)). In this case, the derived test case specification $\Phi'_{pr}(\sigma)$ states that at some step in the execution, (State$_{\mathsf{Webserver}}$ == ClientAuthenticated) && (Webserver.localGID == GIDC) is fulfilled, which corresponds to a specification of all legal ways of the client to be authenticated by the Web server. Thus by applying an appropriate selection of test sequences fulfilling this test case specification to the implementation, we aim at detecting faults in the treatment of legal authentication attempts. The derived test case specification $\Phi''^{(l)}_{pr}(\sigma)$ states that at no step of the execution, State$_{\mathsf{Client}}$ == AwaitForm is fulfilled. This corresponds to cases where the client does not request authentication. With the corresponding test sequences, we aim at detecting faults in the treatment of illegal authentication attempts (in which the client is not involved).

Availability / Fair Exchange Requirements As pointed out in Section 4.2.2, availability and fair exchange requirements can often be specified with properties of the form pr = leadsto(pr_1, pr_2). The property pr is violated if

$$\Phi_{pr}(\sigma) = \neg[\![pr]\!] = \exists i \geq 0 : [\![pr_1]\!](\sigma^i) \wedge \forall j \geq i : \neg[\![pr_2]\!](\sigma^j)$$

holds, i.e. pr_1 is fulfilled in some step, and in the following steps pr_2 is not fulfilled. Note that as pr is a liveness property, its violation cannot be detected by a finite test sequence. However, we can derive a test case specification $\Phi^{(l)}_{pr}$ such that pr is violated up to execution step l, where l is the length of the test sequences:

$$\Phi^{(l)}_{pr}(\sigma) = \exists i \geq 0 : [\![pr_1]\!](\sigma^i) \wedge \forall j : i \leq j \leq l \Rightarrow \neg[\![pr_2]\!](\sigma^j)$$

I.e., pr_1 is fulfilled at some step i, but pr_2 is not fulfilled until step l. It can then be checked that after applying a test sequence fulfilling this test case specification, the implementation is in a state where pr_2 can still be fulfilled (by a postamble, i.e. appropriate additional test inputs). Besides, in a similar way as described for properties of the form $\mathsf{precedes}(pr_1, pr_2)$, Φ_{pr} can be modified by deriving test case specifications referring just to pr_1 and pr_2 respectively. Here, $\Phi'_{pr}(\sigma)$ states that pr_1 is fulfilled at some step (to test if pr_2 is fulfilled in one of the following steps or if the implementation is in a state where pr_2 can still be fulfilled) and $\Phi''^{(l)}_{pr}(\sigma)$ states that pr_2 is never fulfilled up to step l (to test that pr_1 was not fulfilled earlier or the implementation is in a state where pr_2 can still be fulfilled). Formally,

$$\Phi'_{pr}(\sigma) = \exists i \geq 0 : [\![pr_1]\!](\sigma^i)$$

and

$$\Phi''^{(l)}_{pr}(\sigma) = \forall j : 0 \leq j \leq l \Rightarrow \neg[\![pr_2]\!](\sigma^j)$$

$\Phi^{(l)}_{pr}(\sigma)$ and $\Phi'_{pr}(\sigma)$ are weakenings of $\Phi_{pr}(\sigma)$, while $\Phi''_{pr}(\sigma)$ is not.

Example 6.3.3. As an example, consider the security requirement SR.FAIREXC_NO_LOSS of the PalME system (see Section 5.4.2) stating that if electronic value is debited from the payer's purse, it must later be credited to the payee's purse. This requirement is given by the formula $pr = \mathsf{leadsto}(\mathsf{is\_Debit}(\mathsf{LogPayer}), \mathsf{is\_Credit}(\mathsf{LogPayee}))$. Here, $\Phi^{(l)}_{pr}(\sigma)$ specifies test sequences where electronic value is debited from the payer's purse, but not credited until step l. After application of corresponding test sequences, it can be checked if the payee's purse is in a state where it is still possible that the electronic money is indeed credited. $\Phi'_{pr}(\sigma)$ specifies that electronic money is debited from the payer's purse (to verify if the money is credited to the payee's purse in the later steps or the implementation is in a state where this is still possible), and $\Phi''^{(l)}_{pr}(\sigma)$ specifies test sequences where no electronic value is credited to the payee's purse (to check if because of faults in the implementation, it is possible that electronic value is debited from the payer's purse anyway and the implementation is in a state such that the electronic value will not be credited anymore).

Altogether, the testing criterion given by the derivation of test case specifications for a component c under test from the security requirements leads to a set $\mathsf{TS}^{\mathcal{M}}_{c;\mathsf{securityreq}}$ of test case specifications, which consists of the modified versions of the negations Φ_{pr} of the security requirements $pr \in \mathsf{properties}^{\mathcal{M}}(\mathsf{Main}) : \mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}}(pr)$.

Test sequences fulfilling the derived test case specifications can be selected at random or by combining the test case specifications with structural testing criteria such as channel coverage. Besides, the derived test case specifications can be refined manually by the tester using his knowledge about relevant behaviours of the application.

An example for a test sequence generated using a derived test case specification is given in Section 6.5.

6.3.2. Inserting Faults

In this section, we describe how to generate test sequences for vulnerability testing by introducing faults into the specification and determining if and how the introduced faults can lead to violations of the security requirements. Test sequence generation by inserting faults provides the test engineer both with possible faults that lead to violations of the security requirements and with test sequences to test the implementation for these faults.

We extend the process depicted in Figure 6.1 as follows for generating security test sequences (see Figure 6.3). From the threat scenario, we generate mutants by applying model transformations that insert faults into the behavioural specification of the component under test. Instead of the semantical representation $\Psi(\sigma)$ of the original system, semantical representations $\Psi'(\sigma)$ of the mutants of the threat scenario are used to generate test sequences. As test case specifications, we take the predicates $\Phi_{pr}(\sigma)$ corresponding to the negated security requirements. If a test sequence σ fulfilling $\Psi'(\sigma) \wedge \Phi_{pr}(\sigma)$ is found by the test sequence generation, this indicates that the inserted fault introduced a vulnerability with respect to pr, and the trace σ shows how it can be exploited. The input data to the component under test of all traces σ determined this way gives us tests for this component covering possible vulnerabilities with respect to the security requirements and the threat scenario. To determine the expected output when the system behaves correctly, we use the original specification of the component under test as an oracle.

Note that inserting faults can be seen as the counterpart of the test strategy described in Section 6.3.1. The difference to the test strategy described in Section 6.3.1 is that to obtain test sequences with respect to the negations of the security requirements even though $\Psi(\sigma) \wedge \Phi_{pr}(\sigma)$ has no solution, we modify Ψ rather than Φ_{pr}.

Generating Mutants

In mutation testing (see e.g. [MLS78, Off95, ABM98]), faults are introduced into a program, leading to a set of *mutants*. The mutants are used to evaluate the quality of a test suite, which can be measured by the proportion of mutants that are distinguished from the original program by the test suite. The distinguished mutants are said to be *killed* by the test suite. Conversely, we use mutants to determine test sequences rather than to evaluate test suites.

We generate mutants by selecting a transition $tr \in \mathsf{transitions}^{\mathcal{M}}(aut)$ annotated with $\mathsf{critical}$, where $aut = \mathsf{automaton}^{\mathcal{M}}(c)$ is the automaton associated to the com-

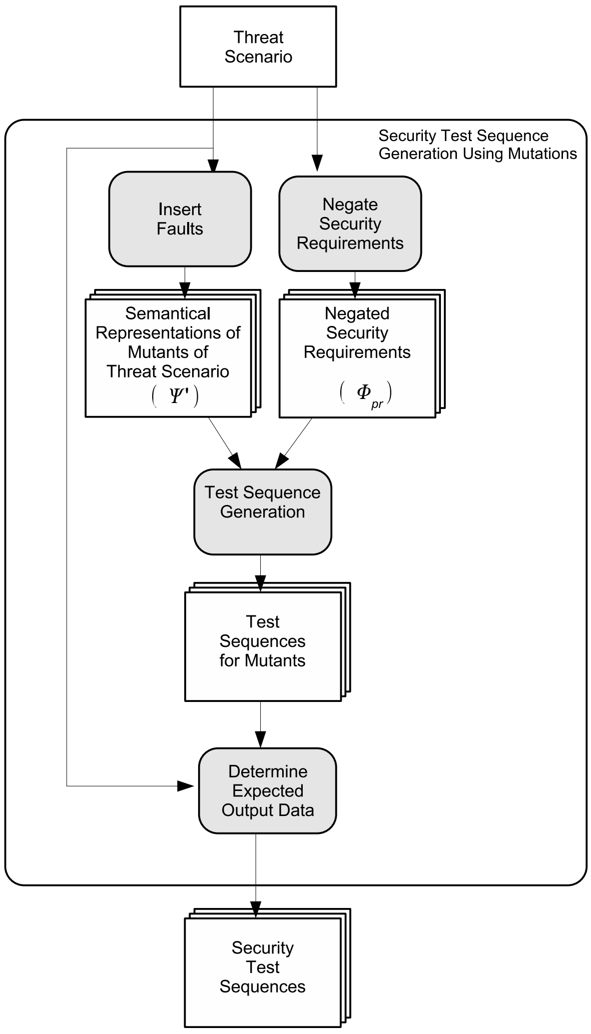

Threat Scenario
Security Test Sequence Generation Using Mutations
Insert Faults
Negate Security Requirements
Semantical Representations of Mutants of Threat Scenario (Ψ')
Negated Security Requirements (Φ_{pr})
Test Sequence Generation
Test Sequences for Mutants
Determine Expected Output Data
Security Test Sequences

Figure 6.3.: Security Test Sequence Generation by Inserting Faults

ponent c under test, and applying a mutation function $\epsilon : \mathsf{Term} \rightarrow \mathcal{P}(\mathsf{Term})$ either to the precondition $\mathsf{pre}^{\mathcal{M}}(tr)$ or to one of the input patterns, output expressions or postconditions. Mutations can be formulated as model transformations. For instance, a model transformation $\mathsf{Mutate}_{\mathsf{pre}}(tr, t')$ for the mutation of the precondition of tr by replacing it with a new term $t' \in \epsilon(\mathsf{pre}^{\mathcal{M}}(tr))$ is specified by

$$\mathsf{pre}^{\mathcal{M}'}(tr) = t'$$

As usual, all other entity sets and functions not mentioned stay unchanged in $\mathcal{M}'$.

The crucial question is which mutants to generate in order to obtain a suitable test suite (with respect to its potential to detect security violations). In [Woo93], the following general principles for the definition of mutations are listed:

- Mutation categories should model potential faults.
- Only simple, first order mutants (i.e., mutants where a single local change was made to the original program) should be generated.
- Only syntactically correct mutants should be generated.
- The user should have control over the selection of which mutation categories to apply at any one time.

Moreover, when generating mutants from abstract security models, the use of cryptographic operations and keys must be taken into account in order to on the one hand not generate mutants that are not assumed to be possible, e.g. mutants using secrets the respective party does not possess, and to on the other hand give the test engineer the possibility to select mutation categories with respect to the model that are relevant for vulnerability testing, such as wrongly implemented verification of cryptographic signatures.

A mutation function ϵ for terms in the security-extended version of the functional language QuestF is depicted in Table 6.1. The rows in the table correspond to particular kinds of mutations. We use the terminology from [BOY00], where mutations for specifications in the input language of the model checker SMV are described (but not defined formally as in Table 6.1). Here, ST stands for Stuck-At (replacement of a boolean expression with True and False, respectively), EN for Expression Negation (replacement of a boolean expression by its negation), OR for Operand Replacement (replacement of an operand by another syntactically legal operand), and LR for Logical Operator Replacement (replacement of a logical operator with another logical operator). In addition to the mutations described in [BOY00], Table 6.1 includes mutations to replace functions (such as constructor or selector functions) by other functions with the same functionality (denoted FR) and

Table 6.1.: Mutation Function for QuestF Terms

$t \in \mathsf{Term}$	$\epsilon(t)$	denot.	interpretation
$t_1 == t_2$	$\{\mathsf{True}, \mathsf{False}\} \cup$ $\{\mathsf{not}(t_1 == t_2)\} \cup$ $\{t_1 == t'_2 : t'_2 \in \epsilon(t_2)\} \cup$ $\{t'_1 == t_2 : t'_1 \in \epsilon(t_1)\}$	$\mathsf{ST}_{==}$ $\mathsf{EN}_{==}$ $\mathsf{OR}_{==}$	faulty equality check, e.g. for an identity of a party or correctness of a signature
$t_1 \&\& t_2$	$\{\mathsf{True}, \mathsf{False}\} \cup$ $\{\mathsf{not}(t_1 \&\& t_2)\} \cup$ $\{t_1 \Vert t_2\} \cup \{t_1 => t_2\} \cup$ $\{t_1 \&\& t'_2 : t'_2 \in \epsilon(t_2)\} \cup$ $\{t'_1 \&\& t_2 : t'_1 \in \epsilon(t_1)\}$	$\mathsf{ST}_{\&\&}$ $\mathsf{EN}_{\&\&}$ $\mathsf{LR}_{\&\&}$ $\mathsf{OR}_{\&\&}$	fault in condition term (conjunction)
	(similarly for other pre-defined boolean functions)		
$\mathsf{is\_C}^k_i(t_1)$ with $\mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Sign}}$	$\{\mathsf{True}, \mathsf{False}\} \cup$ $\{\mathsf{not}(\mathsf{is\_C}^k_i(t_1))\} \cup$ $\{\mathsf{is\_C}^k_i(t'_1) : t'_1 \in \epsilon(t_1)\} \cup$	$\mathsf{ST}_{\mathsf{SignD}}$ $\mathsf{EN}_{\mathsf{SignD}}$ $\mathsf{OR}_{\mathsf{SignD}}$	faulty/missing signature check
$\mathsf{is\_C}^k_i(t_1)$ with $\mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Encr}}$	$\{\mathsf{True}, \mathsf{False}\} \cup$ $\{\mathsf{not}(\mathsf{is\_C}^k_i(t_1))\} \cup$ $\{\mathsf{is\_C}^k_i(t'_1) : t'_1 \in \epsilon(t_1)\} \cup$	$\mathsf{ST}_{\mathsf{EncrD}}$ $\mathsf{EN}_{\mathsf{EncrD}}$ $\mathsf{OR}_{\mathsf{EncrD}}$	faulty/missing encryption check
$\mathsf{is\_C}^k_i(t_1)$ with $\mathsf{C}^k_i \notin \mathsf{C}_{\mathsf{Sign}} \cup \mathsf{C}_{\mathsf{Encr}}$	$\{\mathsf{True}, \mathsf{False}\} \cup$ $\{\mathsf{not}(\mathsf{is\_C}^k_i(t_1))\} \cup$ $\{\mathsf{is\_C}^k_i(t'_1) : t'_1 \in \epsilon(t_1)\} \cup$	$\mathsf{ST}_{\mathsf{MsgD}}$ $\mathsf{EN}_{\mathsf{MsgD}}$ $\mathsf{OR}_{\mathsf{MsgD}}$	faulty/missing type check
$\mathsf{C}^k_i(t_1, \ldots, t_{n^k_i})$ with $\mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Encr}}$	$\{\mathsf{C}^k_{i'}(t_1, \ldots, t_{n^k_i}) :$ $\mathsf{fct}(\mathsf{C}^k_{i'}) = \mathsf{fct}(\mathsf{C}^k_i) \wedge$ $\mathsf{C}^k_{i'} \in \mathsf{C}_{\mathsf{Encr}}\} \cup$ $\{\mathsf{C}^k_i(t_1, \ldots, t'_j, \ldots, t_{n^k_i}) :$ $1 \leq j \leq n^k_i \wedge t'_j \in \epsilon(t_j)\}$	$\mathsf{FR}_{\mathsf{EncrC}}$ $\mathsf{OR}_{\mathsf{EncrC}}$	faulty encryption
	(similarly with signature/ hash/MAC)		
$\mathsf{C}^k_i(t_1, \ldots, t_{n^k_i})$ with $\mathsf{C}^k_i \notin \mathsf{C}_{\mathsf{Encr}} \cup \mathsf{C}_{\mathsf{Sign}} \cup$ $\mathsf{C}_{\mathsf{Hash}} \cup \mathsf{C}_{\mathsf{Mac}} \wedge$ $\mathsf{typeof}(t) \notin \mathsf{T}_{\mathsf{Key}}$	$\{\mathsf{C}^k_{i'}(t_1, \ldots, t_{n^k_i}) :$ $\mathsf{fct}(\mathsf{C}^k_{i'}) = \mathsf{fct}(\mathsf{C}^k_i) \wedge$ $\mathsf{C}^k_{i'} \notin \mathsf{C}_{\mathsf{Encr}} \cup \mathsf{C}_{\mathsf{Sign}} \cup \mathsf{C}_{\mathsf{Hash}}$ $\mathsf{C}_{\mathsf{Mac}}\} \cup$ $\{\mathsf{C}^k_i(t_1, \ldots, t'_j, \ldots, t_{n^k_i}) :$ $1 \leq j \leq n^k_i \wedge t'_j \in \epsilon(t_j)\}$	$\mathsf{FR}_{\mathsf{MsgC}}$ $\mathsf{OR}_{\mathsf{MsgC}}$	corrupted message
$\mathsf{C}^k_i(t_1, \ldots, t_{n^k_i})$ with $\mathsf{typeof}(t) \in \mathsf{T}_{\mathsf{Key}} \wedge$ $t \in \mathsf{Value}$	$(\mathsf{keys}(c) \cap$ $\mathsf{Value}(\mathsf{typeof}(t))) \setminus t$	KR	key confusion
$\mathsf{sel}^k_{ij}(t_1, t_2)$ with $\mathsf{C}^k_i \in \mathsf{C}_{\mathsf{Encr}}$	$\{\mathsf{sel}^k_{ij'}(t_1, t_2) :$ $\mathsf{fct}(\mathsf{sel}^k_{ij'}) = \mathsf{fct}(\mathsf{sel}^k_{ij})\} \cup$ $\{\mathsf{sel}^k_{ij}(t'_1, t_2) : t'_1 \in \epsilon(t_1)\}$ $\{\mathsf{sel}^k_{ij}(t_1, t'_2) : t'_2 \in \epsilon(t_2)\} \cup$	$\mathsf{FR}_{\mathsf{EncrS}}$ $\mathsf{OR}_{\mathsf{EncrS}}$	faulty decryption
$\mathsf{sel}^k_{ij}(t_1)$ with $\mathsf{C}^k_i \notin \mathsf{C}_{\mathsf{Encr}}$	$\{\mathsf{sel}^k_{ij'}(t_1) :$ $\mathsf{fct}(\mathsf{sel}^k_{ij'}) = \mathsf{fct}(\mathsf{sel}^k_{ij})\} \cup$ $\{\mathsf{sel}^k_{ij}(t'_1) : t'_1 \in \epsilon(t_1)\}$	$\mathsf{FR}_{\mathsf{MsgS}}$ $\mathsf{OR}_{\mathsf{MsgS}}$	faulty message extraction

a mutation denoted KR to replace a secret (e.g. a key or a nonce) by another secret which is known by the component under test and which has the same type. We restrict ourselves to specifications where the main part of the behaviour is specified in the state transition diagrams, thus mutations for user-defined functions are not included (but could be easily added). According to the definition of the mutation function, only syntactically correct first-order mutants are generated.

Besides, mutations for different kinds of terms that are of interest for generating security tests because they correspond to faults that are likely to lead to vulnerabilities are distinguished in Table 6.1. I.e., instead of general mutations for function applications, we have a number of mutations for the '==' operator (corresponding to faulty equality checks, for instance for identities or for the correctness of signatures), denoted as $\mathsf{ST}_{==}$, $\mathsf{EN}_{==}$, and $\mathsf{OR}_{==}$, a number of mutations for signature checks $\mathsf{is\_C}_i^k(t_1)$ with $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Sign}}$, denoted as $\mathsf{ST}_{\mathsf{SignD}}$, $\mathsf{EN}_{\mathsf{SignD}}$, $\mathsf{OR}_{\mathsf{SignD}}$ [2]; plus mutations for encryptions, hash computations, etc. This makes it possible for the test engineer to determine an actual mutation function ϵ' to be applied by selecting a part of the mutations shown in the table depending on the testing requirements, the kind of system to be tested, and the available time and computing power — thus giving the user fine-grained control over the selection of which mutation categories to be used.

Let $\mathsf{Mutants}(\mathcal{M}, c, \epsilon')$ be the set of mutants of the model $\mathcal{M}$ resulting from the application of the mutation function ϵ' to the terms on the $\mathsf{critical}$ transitions of the automaton of the component c. If only preconditions are mutated, then

$$\begin{aligned}\mathsf{Mutants}(\mathcal{M}, c, \epsilon') \quad := \quad & \{\mathcal{M}' : \exists tr, t' : \mathsf{Mutate}_{\mathsf{pre}}(tr, t')(\mathcal{M}, \mathcal{M}') \wedge \\ & tr \in \mathsf{CriticalTrans}(\mathcal{M}, c) \wedge t' \in \epsilon'(\mathsf{pre}(tr))\}\end{aligned}$$

where

$$\mathsf{CriticalTrans}(\mathcal{M}, c) \quad = \quad \{tr \in \mathsf{transitions}^{\mathcal{M}}(\mathsf{automaton}^{\mathcal{M}}(c)) : \mathsf{critical} \in \mathsf{tags}^{\mathcal{M}}(tr)\}$$

In the same way, the set $\mathsf{Mutants}(\mathcal{M}, c, \epsilon')$ can be extended with mutants derived through mutations of input patterns, postconditions and output expressions, by defining corresponding model transformations $\mathsf{Mutate}_{\mathsf{in}}$, $\mathsf{Mutate}_{\mathsf{out}}$ and $\mathsf{Mutate}_{\mathsf{post}}$.

Test Sequence Generation

As described above, to generate test sequences using the fault insertion criterion with the mutation function ϵ', we use the behavioural specifications of the mutated threat scenarios:

[2] Here, the D in SignD stands for discriminator; likewise we use C in the subscript for constructor mutations and S for selector mutations.

$$\Psi' \in \{\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'}} : \mathcal{M}' \in \mathsf{Mutants}(\mathcal{M}, c, \epsilon')\}$$

The test case specifications are simply the negated security requirements:

$$\mathsf{TS}^{\mathcal{M}}_{c;\mathsf{ins\_fault}} = \{\Phi_{pr} : pr \in \mathsf{properties}^{\mathcal{M}}(\mathsf{Main}) : \mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}}(pr)\}$$

Now we determine solutions σ to $\Psi'(\sigma) \wedge \Phi_{pr}(\sigma)$ using SAT solving. As the sequences σ are calculated with respect to mutated specifications, it remains to determine the correct expected output data.

For this purpose, from the sequences σ we derive test sequences σ' with

$$\Psi_{[\![c]\!]_{\mathcal{M}}}(\sigma') \wedge (\sigma'|_{\mathsf{inPorts}(c)} = \sigma|_{\mathsf{inPorts}(c)})$$

I.e., σ' is a valid behaviour of c with the same sequence of messages appearing at the input ports. The sequences σ' can be generated via the simulation feature of AutoFocus/Quest, a model checker or constraint solver is not needed.

An example for a test sequence generated via fault insertion is given in Section 6.5.

6.4. Concretisation

To be able to actually test an implementation, the abstract test sequences derived from the threat scenario must be translated to concrete test data. We assume the implementation accepts and outputs bit sequences, which is general enough to represent e.g. TCP/IP network messages or messages sent to or received from smart cards.

In many cases, concretisation can be achieved using straightforward mappings between abstract and concrete test data (as described e.g. in [DBG01]), and executing the test using a test driver that passes the inputs to the component under test and verifies if the outputs are as expected. However, testing security-critical systems involves additional complications, mainly because of non-determinism, for example arising from randomly generated keys and nonces, and because of the use of cryptographic primitives:

- In formal specifications of security-critical systems, such as the AutoFocus/Quest specifications described in Chapter 4, data and cryptographic primitives are usually modelled symbolically. For instance, in the bank application case study described in Section 4.3, $\mathsf{SignCert}(\mathsf{SK}(\mathsf{CA}), \mathsf{C}, \mathsf{GIDC}, \mathsf{PK}(\mathsf{C}))$ stands for a cryptographic signature of the client's identification number GIDC and other data with the secret key $\mathsf{SK}(\mathsf{CA})$ of the certification authority. The reason for this representation is to keep verification feasible and to make the formal specifications easier to understand. The test driver must map the symbols corresponding e.g. to nonces or session keys to sequences of bits in a consistent way. Conversely, sequences of bits created and output by the component under test must be stored by the test driver and used in place of the symbols in the test data in the remainder of the execution. Besides, the test driver is responsible for applying the actual encryption, signature or hash algorithms.

- Sometimes, values (such as transaction numbers or time stamps) are abstracted away in formal specifications to simplify verification (and because they are seen to be independent from a security requirement at hand). These have to be re-included in the concrete test data in a consistent way.
- If encryption is used, the test driver must know the corresponding keys and encryption algorithms to be able to compute the encrypted input data and verify the encrypted output data.
- Hash values or message authentication codes contained in the output data cannot be verified unless the complete data necessary to compute the hash or message authentication code is available to the test driver. Likewise, encrypted output data cannot be verified unless the encryption key is available to the test driver.

Concretisation Mappings

In the following, we show how to address these issues and give formalisations in the AutoFocus/Quest framework. We fix a set TransV of transaction variables that are used to represent transaction data to be stored by the test driver, such as random nonces, time stamps or transaction identifiers. We define a concretisation of abstract messages by mapping each constructor C_i^k with $\mathsf{type}^k \notin \mathsf{T}_{\mathsf{Key}} \cup \{\mathsf{Int}\}$ to a sequence $\mathsf{concrete}(\mathsf{C}_i^k) = [d_1, d_2, \ldots, d_{n_{\mathsf{C}_i^k}}]$ of concretisation elements $d_j \in \mathbb{Z} \cup \mathsf{TransV} \cup \mathsf{Term}$. d_j can be

- an integer value,
- a transaction variable, or
- a QuestF expression, in which the abstract message $m = \mathsf{C}_i^k(\ldots)$ to which the concretisation is applied can be referenced by "this".

An integer value corresponds to a constant sequence of bits to be appended to the concrete test data. The transaction variables $\mathit{transv} \in \mathsf{TransV}$ are associated with a set $\mathsf{values}(\mathit{transv}) \subseteq \mathbb{Z}$ of possible values. A transaction variable in a concretisation is kept consistent with its actual value in the concrete test data. QuestF expressions are evaluated and the result is again concretised. This way, parts of an abstract message can be referred to in its concretisation. For instance, the global identification number GIDC in the above mentioned message SignCert(SK(CA), C, GIDC, PK(C)) is referred to by getGID(this).

To compute the concretisation of a symbolically represented cryptographic message, the test driver must be able to access its arguments. As an example, consider the data type definition

```
data TAgent = A | B | I;
data THashMsg = HashMsg(getAg1: TAgent, getAg2: TAgent);
```

The concretisation of a message $m = \mathsf{HashMsg}(\mathsf{A}, \mathsf{B})$ depends on the concretisation of A and B (and on other data that was possibly abstracted away and is included in concrete(HashMsg)). However, in the cryptographic interpretation for THashMsg, no selectors are defined to extract A and B from m.

Therefore, for the evaluation of the QuestF expressions in the concretisations concrete(m), we treat the data type definition of the model as an ordinary QuestF data type definition without special consideration of cryptographic data types as described in Section 4.2.6. A and B can then be extracted in the usual way, using getAg1(m) and getAg2(m).

Note that this is no contradiction to the properties of the cryptographic operators: the test driver does not extract A and B from the actual (concrete) hash corresponding to m but rather uses A and B from m to compute this hash. The algorithm for the test driver given below takes into account that no concrete hash can be computed when concrete values for the arguments are not available. The same argument applies for encryptions (i.e., the test driver can access the arguments of an encrypted abstract message using ordinary selectors).

Keys $k \in \bigcup_{\mathsf{type}^k \in \mathsf{T}_{\mathsf{Key}}} \mathsf{Value}(\mathsf{type}^k)$ are mapped directly to a single transaction variable $\mathsf{concrete}(k) = [d_1]$ with $d_1 \in \mathsf{TransV}$.

In addition, each data element has to be assigned a field length, and for constructors corresponding to encryption, signature, or hash computation, the algorithms to be used must be fixed. We omit this here for simplification. The actual concretisation, i.e. the mappings concrete and values, must be provided by the developer. See Section 6.5 for examples.

Test Driver Algorithm

An algorithm DO_TEST for the test driver is given in Figure 6.4, taking as input a test sequence σ; in addition, a component under test c, and the concretisation defined via the mappings concrete and values must be provided. It keeps a store *store* for the values of the transaction variables, which is prefilled with values of

```
VAR $store : \mathcal{P}(\mathsf{TransV} \times \mathbb{Z})$, $conditions : \mathcal{P}(\mathsf{Value} \times \mathbb{Z})$;

algorithm DO_TEST($\sigma : (\mathsf{Channel} \to \mathsf{Value})^*$): $\{\mathsf{Pass}, \mathsf{Fail}, \mathsf{Inconclusive}\}$
    $store \leftarrow \{(transv, y) : transv \in \mathsf{TransV} \wedge y \in \mathsf{values}(transv) \wedge |\mathsf{values}(transv)| = 1\}$
    $conditions \leftarrow \emptyset$
    for each step $\beta_i$ in test sequence $\sigma = \beta_1, \ldots, \beta_{|\sigma|}$ do
        for each $(ch, m) \in \beta_i$ do
            if $\mathsf{destP}(ch) \in \mathsf{inPorts}(c)$ then
                send GEN_SEQUENCE($m$) to $c$ via $\mathsf{destP}(ch)$
            else if $\mathsf{sourceP}(ch) \in \mathsf{outPorts}(c)$ then
                wait for output $s$ on $\mathsf{sourceP}(ch)$ (return Fail on timeout)
                if VERIFY_SEQUENCE($m, s$) = False then return Fail
            else
                return Inconclusive {illegal test sequence}

            {verify delayed conditions}
            $conditions' \leftarrow conditions$
            for each $(m, s) \in conditions'$ do
                $conditions \leftarrow conditions \setminus (m, s)$
                if VERIFY_SEQUENCE($m, s$) = False then return Fail

    if $conditions = \emptyset$ then
        return Pass
    else
        return Inconclusive
```

Figure 6.4.: Main Test Driver Algorithm

transaction variables *transv* that are already fixed (i.e., $|\mathsf{values}(transv)| = 1$). Besides, *conditions* contains the comparisons between abstract values and concrete bit sequences that have been delayed because necessary data has not been available.

The algorithm DO_TEST uses the algorithms GEN_SEQUENCE (Figure 6.5) and VERIFY_SEQUENCE (Figure 6.6) to generate concrete test data from abstract input messages in σ to the component c under test, respectively to compare abstract output messages in σ to output data received. In VERIFY_SEQUENCE, $\mathsf{first}(s)$ denotes the prefix of a bit sequence s corresponding to the current concretisation element d_j (depending on the field length of d_j, which we omitted for simplification) and $\mathsf{rest}(s)$ denotes the remaining part of s.

The idea of the algorithms GEN_SEQUENCE and VERIFY_SEQUENCE is as follows. Integer messages $m \in \mathsf{Value}(\mathsf{Int})$ and constant bit sequences $d_j \in \mathbb{Z}$ in $\mathsf{concrete}(\mathsf{C}_i^k)$ are passed directly to the implementation or are compared to the received data. If a transaction variable $d_j \in transv$ appears in $\mathsf{concrete}(m)$ in GEN_SEQUENCE, either a new concrete value is chosen for *transv* as its concretisation and added to the

```
algorithm GEN_SEQUENCE($m$ : Value): $\mathbb{Z}^*$
    {compute concrete data from abstract message $m$}
    if $\exists s' : (m, s') \in conditions$ then
        return $s'$
    $s \leftarrow \epsilon$
    if typeof$(m)$ = Int then
        append concrete bit sequence corresponding to $m$ to $s$
    else
        if $m \notin \bigcup_{\mathsf{type}^k \in \mathsf{T}_{\mathsf{Key}}} \mathsf{Value}(\mathsf{type}^k)$ then
            $\mathsf{C}_i^k \leftarrow \mathsf{head}(m)$
            $[d_1, \ldots, d_{n_{\mathsf{C}_i^k}}] \leftarrow \mathsf{concrete}(\mathsf{C}_i^k)$ ; $n_{\mathsf{concr}} \leftarrow n_{\mathsf{C}_i^k}$
        else
            $\mathsf{C}_i^k \leftarrow \perp$
            $[d_1] \leftarrow \mathsf{concrete}(m)$ ; $n_{\mathsf{concr}} \leftarrow 1$

        for $j \in \{1, \ldots, n_{\mathsf{concr}}\}$ do
            if $d_j \in \mathbb{Z}$ then
                append $d_j$ to $s$
            else if $d_j \in$ TransV then
                if $\exists y : (d_j, y) \in store$ then
                    append $y$ to $s$
                else
                    if $\exists (m', s') \in conditions : m \trianglelefteq m'$ then
                        terminate DO_TEST with result Inconclusive
                    else
                        choose $y \in \mathsf{values}(d_j)$
                        append $y$ to $s$
                        $store \leftarrow store \cup (d_j, y)$
            else if $d_j \in$ Term then
                append GEN_SEQUENCE($\mathsf{eval}_{[\mathbf{this} \leftarrow m]}(d_j)$) to $s$

    if $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}}$ then
        $concr\_key \leftarrow$ GEN_SEQUENCE($\mathsf{eval}(\mathsf{sel}_{i1}^k(m))$)
        $s \leftarrow$ encryption of $s$ with $concr\_key$
    else if $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Sign}}$ then
        $concr\_key \leftarrow$ GEN_SEQUENCE($\mathsf{eval}(\mathsf{sel}_{i1}^k(m))$)
        append signature of $s$ with key $concr\_key$ to $s$
    else if $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Hash}}$ then
        $s \leftarrow$ hash of $s$
    else if $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Mac}}$ then
        $concr\_key \leftarrow$ GEN_SEQUENCE($\mathsf{eval}(\mathsf{sel}_{i1}^k(m))$)
        $s \leftarrow$ MAC of $s$ with $concr\_key$
    return $s$
```

Figure 6.5.: Algorithm GEN_SEQUENCE

```
    algorithm VERIFY_SEQUENCE($m$ : Value, $s$ : $\mathbb{Z}^*$): {True, False}
        {verify concrete data $s$ w.r.t. abstract message $m$}
        if typeof($m$) = Int then
            compare concrete bit sequence corresponding to $m$ to $s$; return result
5:      if $m \notin \bigcup_{\mathsf{type}^k \in \mathsf{T}_{\mathsf{Key}}} \mathsf{Value}(\mathsf{type}^k)$ then
            $\mathsf{C}_i^k \leftarrow \mathsf{head}(m)$
            $[d_1, \ldots, d_{n_{\mathsf{C}_i^k}}] \leftarrow \mathsf{concrete}(\mathsf{C}_i^k)$ ; $n_{\mathsf{concr}} \leftarrow n_{\mathsf{C}_i^k}$
        else
            $\mathsf{C}_i^k \leftarrow \perp$ ; $[d_1] \leftarrow \mathsf{concrete}(m)$ ; $n_{\mathsf{concr}} \leftarrow 1$
10:
        if $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}} \cup \mathsf{C}_{\mathsf{Sign}}$ then
            $k \leftarrow \mathsf{eval}(\mathsf{inv}(\mathsf{sel}_{i1}^k(m)))$
            $[d_1'] \leftarrow \mathsf{concrete}(k)$
            if $\exists concr\_key : (d_1', concr\_key) \in store$ then
15:             if $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Encr}}$ then
                    $s \leftarrow$ decryption of $s$ with $concr\_key$
                if $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Sign}}$ then
                    check signature at end of $s$ using $concr\_key$; remove signature from $s$
                    if signature check failed then return False
20:         else
                $conditions \leftarrow conditions \cup (m, s)$ ; return True
        if $\mathsf{C}_i^k \in \mathsf{C}_{\mathsf{Hash}} \cup \mathsf{C}_{\mathsf{Mac}}$ then
            if GEN_SEQUENCE($m$) can be computed without changing store then
                $concr\_msg \leftarrow$ GEN_SEQUENCE($m$)
25:             return ($concr\_msg = s$)
            else {concrete data for $m$ not completely available}
                $conditions \leftarrow conditions \cup (m, s)$ ; return True

        for $j \in \{1, \ldots, n_{\mathsf{concr}}\}$ do
30:         if $d_j \in \mathbb{Z}$ then
                if $\mathsf{first}(s) \neq d_j$ then return False
                $s \leftarrow \mathsf{rest}(s)$
            else if $d_j \in \mathsf{TransV}$ then
                if $\exists y : (d_j, y) \in store$ then
35:                 if $\mathsf{first}(s) \neq y$ then return False
                    $s \leftarrow \mathsf{rest}(s)$
                else
                    if $\mathsf{first}(s) \notin \mathsf{values}(d_j)$ then return False
                    $store \leftarrow store \cup (d_j, \mathsf{first}(s))$
40:                 $s \leftarrow \mathsf{rest}(s)$
            else if $d_j \in \mathsf{Term}$ then
                if $\neg$ VERIFY_SEQUENCE($\mathsf{eval}_{[\mathbf{this} \leftarrow m]}(d_j), \mathsf{first}(s)$) then return False
                $s \leftarrow \mathsf{rest}(s)$

45:     return True
```

Figure 6.6.: Algorithm VERIFY_SEQUENCE

store, or an already chosen value from the store is taken. When data is received corresponding to *transv*, it is either compared to the value already chosen, or it is checked if the received value is within the specified range and if so, it is added to the store. QuestF expressions $d_j \in \mathsf{Term}$ are evaluated (after replacing this by the current message m) and the result is again concretised, respectively compared to the concrete received message.

Encryptions, signatures, hash or MAC computations are carried out in GEN_SEQUENCE after the concretisation depending on the cryptographic interpretation of the constructor C_i^k. In VERIFY_SEQUENCE, it is possible that the concrete bit sequence for a key or part of the data for which a hash or MAC is to be computed is not yet available to the test driver. In this case, instead the message and the corresponding bit sequence are added to *conditions* to be verified at later steps if the corresponding data becomes available. Otherwise, decryption or verification of signatures, MAC or hash values is performed within VERIFY_SEQUENCE.

If GEN_SEQUENCE is applied to a term contained in *conditions*, the concrete bit sequence received before from the implementation is returned as concretisation. Thus, the test driver acts in a similar way to an intruder, who can build his messages from parts of messages received, even without being able to completely "understand" them (because they are e.g. hashes or encryptions with an unknown key). The concrete bit sequences for proper subterms of a term contained in *conditions* can only be computed if their concretisation do not contain transaction variables for which no values have yet been chosen: the test driver cannot choose own values of the transaction variables because of the constraint in *conditions*, but would have to guess the actual ones – which is impossible if the used encryption and hash algorithms are indeed secure. In this case, an Inconclusive result is reported for the test sequence.

The processing of messages by the test driver is repeated for every step β_i of the test sequence σ and every sending and receiving action in β_i affecting the component c under test. After the processing of each message, the test driver tries to verify the delayed conditions. If this is possible (because a necessary secret has been received), the delayed condition is removed and potentially additional transaction data is extracted from the corresponding message.

As a test verdict, the test driver reports Fail if any verification of the received data failed. Pass is only reported if no delayed conditions remained in *conditions*. Otherwise, the test verdict is Inconclusive.

Generation and Verification of Asymmetric Keys

If asymmetric keys are used in a model, the test driver algorithm described above only works if the concrete values of both the private and the public key are previously available to the test driver, or if only one of them is necessary to compute and verify the test data. The reason for this is that the dependency between the private and the public key is not considered.

As an example, look at the EET of the bank application depicted in Figure 4.21 on page 74. This EET can be seen both as a test sequence for the Client component and for the Webserver component. Assume in the concretisation mappings we fixed concrete values for the keys SK(CA) and PK(CA) of the certification authority, but not for the client keys PK(C) and SK(C). If Client is the component under test, on reception of the message Data(Sign(SK(C), NonceS), SignCert(CA), C, GIDC, PK(C)) the test driver gets the concrete value for PK(C), which can be used to verify the signature. The concrete value for SK(C) is not needed to generate and verify the following messages. On the other hand, if Webserver is the component under test, the test driver must generate the Data(...) message and send its concretisation to the implementation. In this case, concrete values for PK(C) and SK(C) must not be generated independently. Another example where this dependency becomes important is testing a key generator which provides the user with a public/private key pair. If the dependency is not considered, the test driver cannot check if the public and private key belong together.

We show how this issue can be addressed by way of extending the algorithms GEN_SEQUENCE and VERIFY_SEQUENCE. In the algorithm GEN_SEQUENCE, concrete values for a key pair must be generated if the concrete value of one of the keys is not available. This is achieved by replacing lines 26-28 in Figure 6.5 by

```
if $m \in \bigcup_{\mathsf{type}^k \in \mathsf{T}_{\mathsf{Key}}} \mathsf{Value}(\mathsf{type}^k) \wedge \mathsf{eval}(\mathsf{inv}(m) == m) \neq \mathsf{True}$ then
    { $m$ corresponds to an asymmetric key }
    generate public/private key pair $y$, $y^{-1}$
    $[d'_1] \leftarrow \mathsf{concrete}(\mathsf{eval}(\mathsf{inv}(m)))$
    if $d_1$ represents public key then
        $store \leftarrow store \cup (d_1, y)$; $store \leftarrow store \cup (d'_1, y^{-1})$
    else
        if $\exists y' : (d'_1, y') \in store$ then
            terminate DO_TEST with result Inconclusive
        else
            $store \leftarrow store \cup (d_1, y^{-1})$; $store \leftarrow store \cup (d'_1, y)$
else
    { original code from lines 26-28 }
    ...
```

Note that the concretisation for a key only consists of one concretisation element d_1. Besides, note the special case when m is a private key and the corresponding public key is already contained in the store. In this case, an appropriate concrete value for m cannot be generated (otherwise, the cryptosystem would be insecure, as it must not be possible to derive the private key from the public key), and the result Inconclusive is reported. The other case does not occur: if m is a public key and a concrete value for the belonging private key is available, then a concrete value for m is available as well.

In the algorithm VERIFY_SEQUENCE, after line 39, the following statements have to be added:

```
if $d_j$ represents private key then
    compute value $y^{-1}$ for corresponding public key
    $[d'_1] \leftarrow \mathsf{concrete}(\mathsf{eval}(\mathsf{inv}(m)))$
    if $\exists y' : (d'_1, y') \in store$ then
        if $y' \neq y^{-1}$ then return False
    else
        $store \leftarrow store \cup (d'_1, y^{-1})$
```

Here, if a concrete value for a private key is received, we compute the corresponding public key, and add it to the store or check if the existing value in the store is correct.

With these extensions, treatment of asymmetric keys used in a model is fully supported by the test driver.

Remarks

Figure 6.7 summarises the described approach for executing security tests based on a concretisation mapping. Note that with the exception of the dependency between private and public keys addressed above, the presented test driver algorithm relies on the assumption that the values of concretisation elements are independent of each other and are concatenated in the concretisation. Examples where this is not the case are length fields preceding a bit sequence of variable length, checksums, cases where the same information is represented by two different formats (e.g. in the CEP specifications, there are two ways to represent a currency, which must be kept consistent in successive messages), or non-atomic keys constructed from other data elements via exclusive or (XOR) operations. Whereas the computation of values for length fields can be performed in a limited way by QuestF expressions, general treatment of dependencies and connections of the data elements other than by concatenation requires an extension of the described approach. For instance, a plugin interface to the test driver can be developed such that the execution of scripting code can be triggered if a certain data element is encountered.

Also, note that if more than one transaction is to be tested using a single test sequence, the store and conditions must be reset between the transactions. Besides, in some cases a fixed preamble to the test sequence may have to be generated by the test driver.

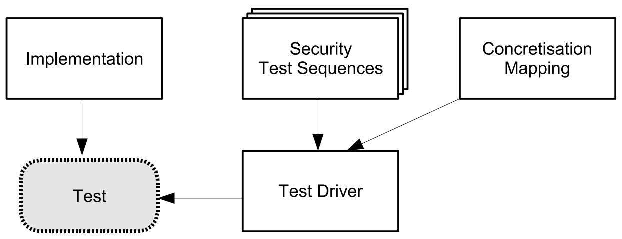


Figure 6.7.: Test Execution Based on Concretisation Mapping

6.5. Case Study: CEPS Load Transaction

In the following, we demonstrate security testing at the example of the CEPS load transaction introduced in Section 4.4.1. The component under test is a CEPS purse card application.

6.5.1. Generation of Test Sequences

Derivation of Test Case Specifications from Security Requirements

From the security requirements of the CEPS load transaction, we derived 7 test case specifications using the guidelines described in Section 6.3.1: two test case specifications Φ'_{pr} and $\Phi''^{(l)}_{pr}$ for $pr = \mathsf{SR.NONREP\_FAILED\_TRANS}$; three test case specifications $\Phi^{(l)}_{pr}$, Φ'_{pr} and $\Phi''^{(l)}_{pr}$ for $pr = \mathsf{SR.FAIREXC\_LACQ}$; and two test case specifications Φ'_{pr} and $\Phi''^{(l)}_{pr}$ for $pr = \mathsf{SR.NONREP\_LOAD\_AUTH}$. There is no test sequence that fulfils the test case specification $\Phi^{(l)}_{pr}$ for $pr = \mathsf{SR.FAIREXC\_LACQ}$, stating that at some execution step, a load approval is received by the LSAM, rl is included in the Credit message and the card reports an error, but no valid $\mathsf{R\_CEP}$ is sent by the card until step l. This is because in the model, the card always sends $\mathsf{R\_CEP}$ at the same time as it reports the error to the log. The other six test case specifications can be fulfilled. As the length of the test sequences, we chose $l = 18$. $l = 18$ is sufficient such that test sequences are computed for the six satisfiable test case specifications and the required computation time is still acceptable. Computing the test sequences took approximately 30s on average on the hardware described in Section 5.4.2.

To avoid the computation of trivial test sequences for $\Phi''^{(l)}_{pr}$ (for both the non-repudiation requirements and the fair exchange requirement), we added the conjunct $\exists i \geq 0 : \sigma(i)(\mathsf{Card.cLog}) \neq \perp$ to these test case specifications. This way, test sequences consisting of a complete transaction carried out by the card and finished by writing a log entry are computed. Otherwise executions where no input at all is sent to the card would fulfil the specifications Φ''_{pr}, as they state that certain outputs must *not* be produced by the card.

As an example, we take the security requirement pr = SR.NONREP_FAILED_TRANS. Remember that this requirement stated that if the Load Security Application Module LSAM received a load approval from the issuer and gets a message from the card containing a random number R_CEP that validates correctly, then the card must have reported an error to its log. Thus, Φ'_{pr} specifies test sequences where a load approval and a correctly validating R_CEP was received by the LSAM (to check if indeed an error was reported by the card), and $\Phi''^{(l)}_{pr}$ specifies test sequences where no error is reported by the card (to check that the LSAM does not receive a correctly validating R_CEP anyway).

Figure 6.8 shows an AutoFocus/Quest EET computed from the test case specification Φ'_{pr}. Note that the intruder has the control of the communication between the card, the LSAM and the issuer on the channels annotated with public in the security-enriched model. First, the card and the LSAM are initialised for the transaction amount 2 (messages 1 and 2). The intruder then stores the card's reply containing the hash of R_CEP (message 3) and generates a reply to the InquireCardInfo message sent by the LSAM (messages 4 and 5). After that, the intruder sends a Credit message with an incorrect MAC MacS2 to the card (message 7) and replies to the initialisation message from the LSAM (intended to be sent to the card) with the stored reply from the card (messages 6 and 8). The card now reports an error and includes R_CEP in its response to the Credit message (messages 9 and 10). In the following steps, the intruder forwards the load request from the LSAM to the issuer and the load approval back to the LSAM (messages 11,12,13,14). Finally, the LSAM (now in a state where a load approval has been received) sends a Credit message and receives from the intruder the stored reply reporting an error and including the R_CEP (messages 15,16). The remaining messages generated by the intruder are spurious messages that are ignored by the receiving components.

Inserting Faults

For simplicity, we demonstrate test sequence generation by inserting faults into the specification of the Card component by using only a small subset of the mutations listed in Table 6.1: we replace checks for equality by True and by False in boolean terms in the preconditions, and we replace keys appearing as arguments of the messages in the output expressions by other keys used by the card. Thus, ϵ' consists of the following mutations:

- $\mathsf{ST}_{==}$, and OR_f for $f \in \{\&\&, ||, \mathsf{not}, =>\}$ (so that $\mathsf{ST}_{==}$ can be applied to the operands of a boolean function); and
- KR, and $\mathsf{OR}_{\mathsf{MsgC}}$ (so that KR can be applied to the arguments of a message).

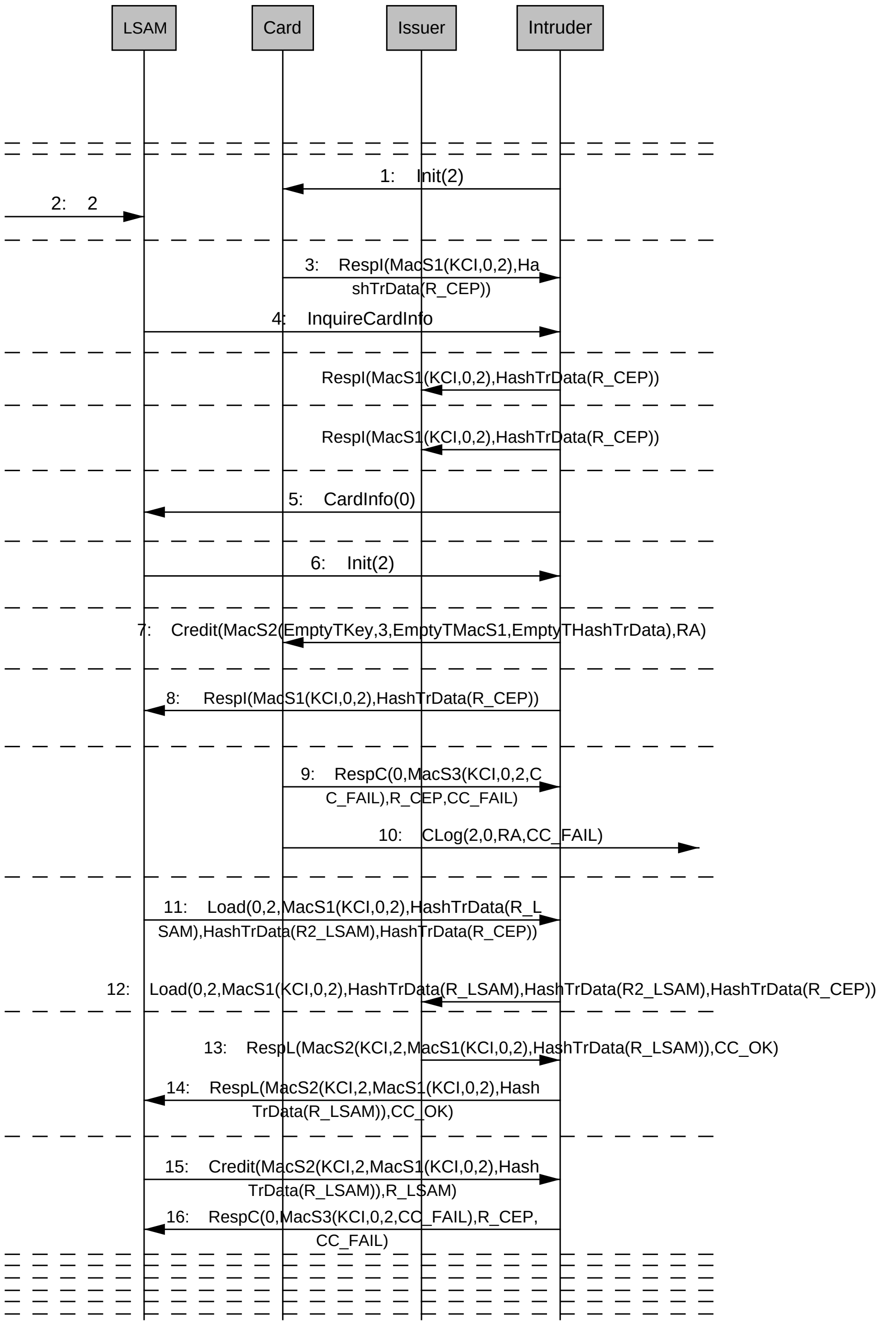

LSAM
Card
Issuer
Intruder
1: Init(2)
2: 2
3: RespI(MacS1(KCI,0,2),HashTrData(R_CEP))
4: InquireCardInfo
RespI(MacS1(KCI,0,2),HashTrData(R_CEP))
RespI(MacS1(KCI,0,2),HashTrData(R_CEP))
5: CardInfo(0)
6: Init(2)
7: Credit(MacS2(EmptyTKey,3,EmptyTMacS1,EmptyTHashTrData),RA)
8: RespI(MacS1(KCI,0,2),HashTrData(R_CEP))
9: RespC(0,MacS3(KCI,0,2,CC_FAIL),R_CEP,CC_FAIL)
10: CLog(2,0,RA,CC_FAIL)
11: Load(0,2,MacS1(KCI,0,2),HashTrData(R_LSAM),HashTrData(R2_LSAM),HashTrData(R_CEP))
12: Load(0,2,MacS1(KCI,0,2),HashTrData(R_LSAM),HashTrData(R2_LSAM),HashTrData(R_CEP))
13: RespL(MacS2(KCI,2,MacS1(KCI,0,2),HashTrData(R_LSAM)),CC_OK)
14: RespL(MacS2(KCI,2,MacS1(KCI,0,2),HashTrData(R_LSAM)),CC_OK)
15: Credit(MacS2(KCI,2,MacS1(KCI,0,2),HashTrData(R_LSAM)),R_LSAM)
16: RespC(0,MacS3(KCI,0,2,CC_FAIL),R_CEP,CC_FAIL)

Figure 6.8.: EET for Test Case Specification $\Phi'_{\mathsf{SR.NONREP\_FAILED\_TRANS}}$

With the above mutations, there are 20 mutants of the threat scenario. 6 of these allow executions violating one of the security requirements. As the length of the test sequences, we chose $l = 18$. Increasing l did not increase the number of counterexamples further. The required computation time was on average about 400s per mutant and security requirement. The reason why computing test sequences from mutants takes longer than by derivation of test case specifications from security requirements is in the cases where a security requirement is not violated by a mutant. Here, the bounded model checker cannot terminate early because a counterexample has been found.

As an example, consider the transition from state Init to state LoadSucc in the STD of the Card component (see Figure 4.26 on page 81). Replacing EmptyTKey by R_CEP in the message RespC sent to the LSAM makes possible an execution where the security requirement SR.NONREP_FAILED_TRANS (which we also dealt with in the previous section by deriving test case specifications from it) is violated.

Figure 6.9 shows the computed EET (spurious messages generated by the Intruder component but ignored by the receiving components are omitted in this figure). In messages 1 and 2, the intruder initialises the card for the transaction amount 1 and intercepts the card's reply. After that, he sends a Credit message with an empty S2 MAC EmptyTMacS2 to the card (message 3). Thus, the transition from state Init to state LoadSucc is taken. The card writes the condition code CC_OK to its log (message 4), but erroneously outputs R_CEP in the reply (message 5) because of the mutation. In the following, the LSAM is initialised with the same transaction amount 1 and the intruder performs an interaction with the LSAM answering the InquireCardInfo message correctly and sending the intercepted reply from the card as a reply to the initialisation message sent by the LSAM (messages 6-10). The load request issued by the LSAM and the load approval are then forwarded by the intruder between the LSAM and the issuer (messages 11-14). Finally, the LSAM sends a Credit message (message 15), to which the intruder replies with the intercepted reply from the card containing R_CEP (message 16). In the end, the LSAM is in a state where a load approval was received and a correctly validating R_CEP, but no error was reported by the card. This is a violation of the security requirement SR.NONREP_FAILED_TRANS.

In the test sequence for the card derived from this trace, the expected output, which is determined by the original (i.e., unmutated) specification of the card, contains EmptyTKey instead of R_CEP in the card's reply in message 5.

6.5.2. Concretisation

We show how test sequences for the Card component consisting of abstract QuestF messages are concretised for the purpose of testing an actual CEPS card application at the example of the messages 1 and 3 of the EET in Figure 6.8 (the Init message

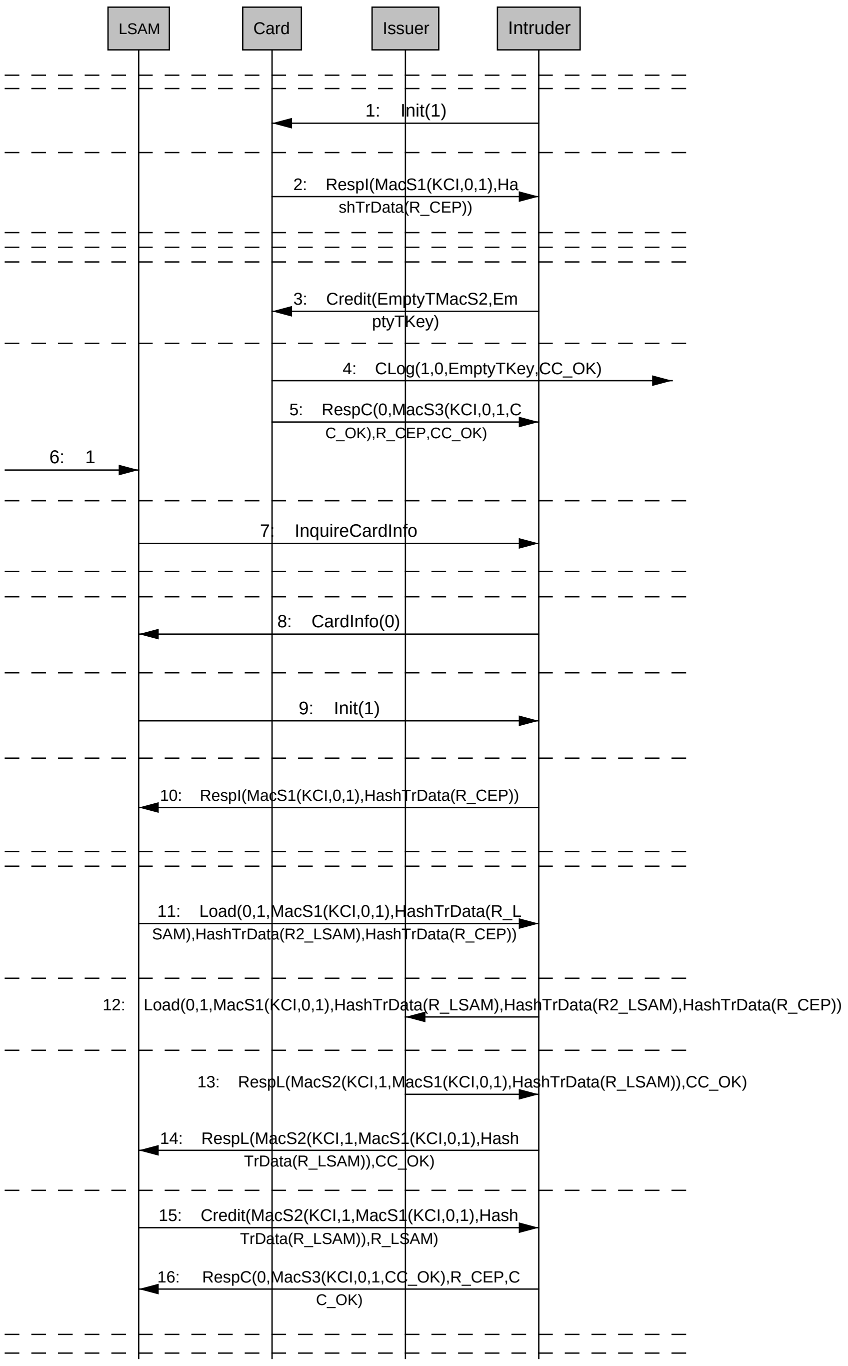

LSAM
Card
Issuer
Intruder
1: Init(1)
2: RespI(MacS1(KCI,0,1),HashTrData(R_CEP))
3: Credit(EmptyTMacS2,EmptyTKey)
4: CLog(1,0,EmptyTKey,CC_OK)
5: RespC(0,MacS3(KCI,0,1,CC_OK),R_CEP,CC_OK)
6: 1
7: InquireCardInfo
8: CardInfo(0)
9: Init(1)
10: RespI(MacS1(KCI,0,1),HashTrData(R_CEP))
11: Load(0,1,MacS1(KCI,0,1),HashTrData(R_LSAM),HashTrData(R2_LSAM),HashTrData(R_CEP))
12: Load(0,1,MacS1(KCI,0,1),HashTrData(R_LSAM),HashTrData(R2_LSAM),HashTrData(R_CEP))
13: RespL(MacS2(KCI,1,MacS1(KCI,0,1),HashTrData(R_LSAM)),CC_OK)
14: RespL(MacS2(KCI,1,MacS1(KCI,0,1),HashTrData(R_LSAM)),CC_OK)
15: Credit(MacS2(KCI,1,MacS1(KCI,0,1),HashTrData(R_LSAM)),R_LSAM)
16: RespC(0,MacS3(KCI,0,1,CC_OK),R_CEP,CC_OK)

Figure 6.9.: EET Computed by Inserting Faults

```
data TMessage = ...
              | Init(getIM: Int)
              | RespI(getRIS1: TMacS1, getRIHC: THashTrData)
              | ...;

data TMacS1   = EmptyTMacS1
              | MacS1(getS1Key: TKey, getS1Bal: Int,
                      getS1M: Int);

data THashTrData = EmptyTHashTrData
                 | HashTrData(getHashTrKey: TKey);

data TKey = EmptyTKey | KCI | R_CEP | ...;
```

Figure 6.10.: CEPS Load Transaction: Part of Data Type Definition

to the card and the corresponding response RespI). The relevant part of the data type definition is depicted in Figure 6.10.

Table 6.2 shows a part of the definition of the concretisation of the abstract QuestF messages. The sequences $\mathsf{concrete}(\mathsf{C}_i^k)$ of concretisation elements and the sets $\mathsf{values}(\mathit{transv})$ of possible values of the transaction variables were chosen such that the test driver generates and expects data conforming to the CEP specifications.

In the example, the message Init(2) to the card is translated e.g. to the bit sequence

$$\text{0x 90 50 00 00 17 16 0503160901 0CCC0E 1234FFFF 123456780FFF } \underline{\text{00000002}} \text{ 00}$$

Here, "0x" indicates that the bit sequence is given in hexadecimal notation. The constructor of the message is Init, so the test driver looks up the definition of concrete(Init) (see Table 6.2). The first six bytes are constants, for the transaction variable $\mathsf{DTHR}_{\mathsf{LDA}}$ a correctly encoded date/time value 0x0503160901 is chosen, etc. The underlined part corresponds to the transaction amount 2, retrieved via getIM(Init(2)).

As the reply $\mathsf{RespI}(\mathsf{MacS1}(\mathsf{KCI}, 0, 2), \mathsf{HashTrData}(\mathsf{R\_CEP}))$, the test driver expects

$$\text{0x 21 123123FF 1122334455FF 050404 } \underline{\mathsf{NT}_{\mathsf{CEP}}}\ \underline{S1}\ \underline{h} \text{ 0a } \underline{\mathsf{DD}_{\mathsf{CEP}}} \text{ 9000}$$

$\mathsf{NT}_{\mathsf{CEP}}$, the transaction identifier assigned by the card, and $\mathsf{DD}_{\mathsf{CEP}}$ (discretionary data) are read from the reply and added to the store for later use. The MAC $S1$ can be verified after $\mathsf{DD}_{\mathsf{CEP}}$ was read, because then all data that is part of $S1$ and

Table 6.2.: Concretisation of Abstract CEPS Messages

Constructor C_i^k	$\mathsf{concrete}(\mathsf{C}_i^k)$
Init	[0x90, 0x50, 0x00, 0x00, 0x17, 0x16, $\mathsf{DTHR}_{\mathsf{LDA}}$, $\mathsf{CURR}_{\mathsf{LDA}}$, $\mathsf{ID}_{\mathsf{LACQ}}$, $\mathsf{ID}_{\mathsf{LDA}}$, getIM(this), 0x00]
RespI	[0x21, $\mathsf{ID}_{\mathsf{ISS}}$, $\mathsf{ID}_{\mathsf{CEP}}$, $\mathsf{DEXP}_{\mathsf{CEP}}$, $\mathsf{NT}_{\mathsf{CEP}}$, getRIS1(this), getRIHC(this), 0x0A, $\mathsf{DD}_{\mathsf{CEP}}$, 0x90 00]
MacS1	[0x0C, $\mathsf{NT}_{\mathsf{CEP}}$, $\mathsf{ID}_{\mathsf{ISS}}$, $\mathsf{ID}_{\mathsf{CEP}}$, getS1Bal(this), $\mathsf{BALMAX}_{\mathsf{CEP}}$, getS1M(this), $\mathsf{CURR}_{\mathsf{LDA}}$, $\mathsf{DTHR}_{\mathsf{LDA}}$, $\mathsf{DEXP}_{\mathsf{CEP}}$, $\mathsf{DD}_{\mathsf{CEP}}$, $\mathsf{ID}_{\mathsf{LACQ}}$, $\mathsf{ID}_{\mathsf{LDA}}$]
HashTrData	[$\mathsf{ID}_{\mathsf{LACQ}}$, $\mathsf{ID}_{\mathsf{LDA}}$, $\mathsf{ID}_{\mathsf{ISS}}$, $\mathsf{ID}_{\mathsf{CEP}}$, $\mathsf{NT}_{\mathsf{CEP}}$, getHashTrKey(this)]
R_CEP	[$\mathsf{R}_{\mathsf{CEP}}$]
...	...
transv	values(*transv*)
$\mathsf{DTHR}_{\mathsf{LDA}}$	$\{x \in \mathbb{Z} : x$ is correctly encoded date/time value$\}$
$\mathsf{ID}_{\mathsf{LACQ}}$	{0x 12 34 FF FF}
$\mathsf{NT}_{\mathsf{CEP}}$	$\{x \in \mathbb{Z} : x$ is 32 bit integer$\}$ (transaction identifier from card)
KCI	{0x 2A 0C 40 ... BC} (192 bit key between card and issuer)
$\mathsf{R}_{\mathsf{CEP}}$	$\{x \in \mathbb{Z} : x$ is 128 bit integer$\}$ (random number from card)
...	...

the key KCI are known to the test driver. The hash h cannot be verified because the value of R_CEP is not available. Thus, h is added to the constraints in *conditions* and will be checked at a later execution step. The other transaction variables such as $\mathsf{ID}_{\mathsf{ISS}}$ were assumed to have a fixed value (only one element in values(*transv*)) that is checked by the test driver.

In the model of the Card component, it is assumed that the card's balance is initially zero. Thus, before each test sequence is applied, the test driver must send a preamble such that the balance stored on the card to be tested is reset. We do that by reading the balance and sending a command sequence to withdraw the respective amount. Besides, to obtain the log entry (message 10 in Figure 6.8), an additional request message must be sent to the card, which for simplification has been omitted in the model and included in the test driver.

6.5.3. Test Execution and Evaluation

Using an implementation of the test driver described in Section 6.4, we conducted tests of the unlinked load functionality of a CEPS purse card application with test sequences computed from the AutoFocus/Quest model. The tested CEPS purse card application was developed in JavaCard and deployed on a JavaCard-based smart card. It is described in more detail in [Hei03].

As a first step, we computed test sequences from the threat scenario using general testing criteria (see the remarks under "Emphasis on Completeness" in Section 6.3). In particular, we covered all critical transitions of the Card component with test sequences and generated a test sequence for each alternative in the data type definition of the type TMessage (e.g. Init, Credit and InquireCardInfo) and each critical control state of the automaton of the Card component such that a message belonging to the respective alternative is received by the Card component while the automaton is in the respective control state (an extension of channel coverage). Conducting these tests led to the discovery of a number of inconsistencies between the model and the implementation. For instance, it was specified in the model that the card can process an InquireCardInfo message between the processing of the Init and Credit messages, whereas in the implementation, the transaction state is reset when an InquireCardInfo message is received. The CEP specifications do not fix a particular behaviour in this case. This and similar inconsistencies were eliminated by either changing the model or the implementation. In the described case, the implementation was changed by removing the code where the transaction state was reset. After the changes were carried out, the test driver reported a pass verdict for all applied test sequences. Cryptographic expressions were generated and validated correctly.

In addition, we performed vulnerability tests using the 12 test sequences whose generation was described in Section 6.5.1 (6 test sequences generated by derivation of test case specifications from security requirements and 6 test sequences generated by inserting faults). For this purpose, we planted faults into the CEPS purse card application by modifying the code and monitored the capability of the test sequences to detect these faults. We chose three kinds of faults:

- general condition validation faults: omission of a conditional statement corresponding to validation of a condition that should in normal executions *not* be fulfilled (e.g. not setting the condition code to "fail" if the received $S2$ MAC was incorrect), respectively replacing conditions or parts of conditions that should in normal executions be fulfilled with true (e.g. always crediting the transaction amount rather than only if the condition code is "ok"). 9 such modifications are possible in the Java method in the CEPS purse card application where the Credit message is processed.

- manually selected faults which make executions possible that violate the security requirements within one transaction and which concern aspects of the implementation that were not omitted in the model (e.g. writing "fail" to the log even though the transaction completed successfully). We selected 9 such faults, 3 for each security requirement.

- faults concerning aspects of the implementation that were abstracted in the model and dealt with by the test driver: for all 24 occurrences of transaction variables in the concretisations of Init, RespI, MacS1 and HashTrData (see Table 6.2; terms built with the latter two constructors can appear as subterms

Table 6.3.: Results of Testing CEPS Load Application

	#faults planted	#faults detected		total # failed tests	
		t.c. spec. derivation (6 t.s.)	fault insertion (6 t.s.)	t.c.spec. derivation (6 t.s.)	fault insertion (6 t.s.)
condition validation	9	4	4	8 out of 54	9 out of 54
violations of security requirements	9	9	9	22 out of 54	27 out of 54
corruption of abstracted transaction data	24	24	24	122 out of 144	134 out of 144

of terms built with Respl), we changed the implementation such that a wrong value for the corresponding data element is received or sent.

For each of the 42 faults, a modified version of the implementation was built and the 12 test sequences were applied. Table 6.3 shows the results. It lists the number of faults detected by the test sequences generated by derivation of test case specifications from security requirements and by inserting faults, respectively. Moreover, the total number of tests leading to a “fail” result (i.e., detecting the fault) is given for both cases.

4 out of the 9 condition validation faults were detected. All of the 4 detected faults introduced vulnerabilities with respect to the security requirements (e.g., allowing to credit the card without a valid $S2$ MAC). Of the remaining 5 planted faults, three are generally not detectable, because they do not affect the behaviour of the implementation (one concerns the repetition of a condition check done before, for robustness reasons; the other two concern erroneously trying to read a non-existing data element of the Init message which is not used later on). The fourth fault involves checking the APDU for correct length. It is not detected because we do not consider faults at the APDU level . The last fault could actually be detected by an appropriate test sequence: here, contrary to the specification, the R_CEP is output by the card even though no R_LSAM was provided by the LSAM. However, this fault does not correspond to a violation of the security requirements, which only state that on failure of a transaction, R_CEP must be output if R_LSAM was received but not that it must not be output if R_LSAM was not received. Approximately the same number of tests lead to the detection of faults for both derivation of test case specifications from security requirements and fault insertion.

All 9 planted faults leading to violations of the security requirements were detected. Thus, the generated test sequences cover well the security-critical functionality of the implementation. It must be noted though that not all faults leading to the violation of a security requirement pr were detected by test sequences generated from test case specifications derived from pr, respectively by test sequences generated by inserting faults into the model leading to a violation of pr. Sometimes, test sequences generated with respect to the other (related) security requirements detected these faults. To improve the fault detection with respect to a single security requirement, a larger number of test sequences must be generated per test case specification (we generated only one) using an additional coverage criterion such as channel coverage; or a more comprehensive mutation function must be used in the case of inserting faults. More tests lead to the detection of faults for fault insertion than for derivation of test case specifications from security requirements. This is because in testing using fault insertion, also information about possible faults (i.e., the mutation function) is used for test sequence generation, thus the fault detection capability of the test sequences is better if the mutation function was selected appropriately. However, computing the test sequences takes considerably longer.

Finally, all 24 planted faults concerning the corruption of abstracted transaction data were detected. Usually, such a result cannot be achieved, because the test driver can only verify the consistency of the transaction data with values specificed in the concretisation, generated, or received previously. However, in the RespI message of the CEPS load protocol, all transaction data sent and received is contained again in the $S1$ MAC. Therefore, the test driver can detect a single corruption by verifying $S1$. Corruptions of the transaction data in the hash contained in the RespI message are also detected, but only in test sequences that cause the card to output R_CEP. Otherwise, the hash cannot be verified. The total number of failed tests is comparatively high because all generated test sequences contain an Init-RespI interaction. There are more failed tests in the fault insertion case because more of the respective test sequences cause the card to output R_CEP.

The CEPS load application case study demonstrates that specification-based security testing based on a security-enriched model can indeed be successfully applied for the generation and execution of application-specific vulnerability tests. An important advantage is that small additional effort is needed if a verified security-enriched model is already available. Further empirical evaluation of the given strategies for the generation of vulnerability tests, necessary for a more rigorous assessment of their effectiveness, is subject of ongoing work. As a part of this evaluation, our approach will be applied for test sequence generation for security-critical service-based systems in the project Mewadis [Mew04].

6.6. Related Work

Model-Based Generation of Test Sequences

General references to research into model-based test sequence generation have been given in Section 6.2. In [Pre03] (p. 76ff), a description is given of possibilities to derive test case specifications from universal properties for the purpose of safety testing. This is related to our approach for the derivation of test case specifications from security requirements (see Section 6.3.1), whereas we start from the negations of the requirements, focus on patterns for common security requirements and formally state the respective derived test case specifications. In [ABM98], an approach for the generation of test sequences using model checking is presented where mutations are introduced into a model and the model checker is used to generate traces that distinguish these mutations from the original model. [ADX01] describes generation of test sequences for safety properties (invariants) using mutation analysis. The general mutation operators used both in [ABM98] and [ADX01] are described in [BOY00].

Security Testing

In [CB03], a model-based approach for security functional testing is described based on models in the specification language SCR. Test case specifications are determined by a generic coverage criterion (domain convergence path, a kind of condition coverage). The approach is focused on access control. The use of cryptography in input and output data is not considered. The AVA approach [VM01] is designed to allow the assessment of the susceptibility of code against perturbations of the data state. For each of a defined set of perturbations, the implementation is tested if a violation of security requirements becomes possible. If this is the case, a potential vulnerability has been detected. Likewise, [DM00] suggests perturbations of the environment (e.g. file attributes). No specific testing criteria are used ([VM01] employs statistical testing), cryptography is not considered and a formal specification is not assumed to be given. In the PROTOS approach [Kak01] already mentioned in Section 6.3, security-critical protocols are specified in the form of grammars. Mutations of the grammars are carried out such that illegal message parts are sent and it is monitored if this causes the implementation to crash by covering the executions of the protocol involving the illegal message parts. Therefore, the PROTOS approach is particularly suitable for testing of implementation-level vulnerabilities such as buffer overflow.

In [JW01c] (joint paper with J. Jürjens), we describe a model-based approach for firewall testing.

Concretisation

Usually, the concretisation of abstract test sequences gained through model-based test sequence generation is performed by an application-specific test driver, often implemented in a scripting language. In [PPS$^+$03] (where an application of the testing method from [Pre03, Löt03] is described), Python is used for this purpose. Issues arising because of the use of cryptography in the model are also addressed in this paper, by verifying concrete cryptographic data or abstracting from the concrete content of a message if sufficient information for its verification is not available. In contrast, our approach described in Section 6.4 and [WJ02] aims at providing a generic concretisation mapping, and supporting consistency checks for abstracted transaction data and delayed verification of cryptographic messages. In the model-based testing approach presented in [DBG01], test concretisation is achieved by the definition of a mapping from variable assignments (as parts of a test sequence) to macros that are translated to concrete command sequences to be sent to and received from the implementation. Security-specific issues are not addressed. In the PROTOS approach [Kak01], for test concretisation grammar symbols are mapped by rules to scripting commands that are executed by the test driver. The examples given do not involve the use of cryptography.

6.7. Summary and Discussion

We explained how a threat scenario generated from a security-enriched model can be used for the purpose of **security testing** to gain confidence in the security of an implementation. To this end, we adapted methods from model-based testing to the domain of security-critical systems. Both security functional testing and vulnerability testing are supported.

We presented concepts for the **generation of test sequences for security-critical systems** from the threat scenario. Which test sequences are to be generated is determined by test case specifications resulting from **testing criteria**. For security functional testing, classical structural and functional coverage criteria can be applied. The annotations in the security-enriched model make it possible to focus on critical parts. For vulnerability testing, we gave two testing criteria where the security requirements are taken into account and which should lead to test sequences likely to detect possible vulnerabilities of the implementation. By **derivation of test case specifications from security requirements**, we obtain test sequences that fulfil properties that can contribute to the violation of security requirements. By **inserting faults** into a model, we obtain test sequences leading to the violation of security requirements, which we use to test if the implementation reacts correctly in these cases. Fault insertion corresponds to a model transformation depending on a **mutation function** which specifies how the terms in a model can be modified.

We gave a mutation function for terms in the functional language QuestF taking into account the security-specific extensions described in Section 4.2.6.

The messages in the generated test sequences are terms in QuestF. To execute a test, these abstract messages must be concretised. We explained how this can be achieved by way of a **concretisation mapping** and a **test driver** using a generated test sequence and a concretisation mapping as its input for performing a test of the implementation. Our approach supports the use of cryptography, in particular the verification of the results of cryptographic operations, which is delayed if the necessary information is not available. In addition, the test driver deals with data which was modelled symbolically in the abstract model or which was left out to reduce the model's complexity.

For an evaluation, functionality for the generation of test sequences based on bounded model checking and a test driver have been implemented as plugins for the AutoFocus/Quest tool. We **demonstrated model-based security testing** at the example of a **CEPS purse card application** and evaluated its capability of detecting faults planted into the implementation. Our approach worked well both in detecting faults in the security-critical behaviour reflected in the model and in the handling of symbolically modelled or omitted transaction data.

According to the current state of research, there seem to be no general criteria for the assessment of the quality of test suites [Pre03], with the exception of conformance tests under severely restrictive assumptions about the underlying system model (see e.g. [Ura92, LY96]; here the underlying system model are (non-extended) finite state machines, where the size of the state space of the implementation must be known and relatively small). The main benefit of the testing criteria to derive test sequences described in this chapter is that test sequences can be generated from the security-enriched model under consideration of the security requirements with low additional effort needed on part of the test engineer. In particular, a model of the environment to control the test generation need not be manually built, but is generated automatically together with appropriate test case specifications. On the other hand, the test engineer is free to refine the test case specifications or to generate additional test sequences using structural testing criteria. Empirical evaluation of the presented method with the help of additional case studies is subject of further work.

Note that our model-based testing approach does not aim at replacing conventional vulnerability testing techniques completely. Rather, we reduce the necessary effort by automating the test of application-specific vulnerabilities reflected on the level of the available security-enriched model and its concretisation.

7. Security Mechanisms: Layered Protocols

In top-down oriented development of security-critical systems, one begins with an abstract specification of the security-relevant aspects, which is concretised in a stepwise fashion by selecting appropriate security mechanisms. In this chapter, we show how such an approach can be supported by model-based development techniques. Based on abstract assumptions annotated to model elements to restrict the intruder's capabilities, the appropriate security mechanisms are applied by introducing them via model transformations. In particular, we give model transformations that preserve security and thus make it unnecessary to repeat the verification of the security requirements. We focus on security mechanisms realising assumptions on communication channels by the introduction of additional protocol layers.

This chapter is structured as follows. In Section 7.1, we describe the general concepts of the application of security mechanisms as a part of our model-based approach to the development of security-critical systems. A model transformation for the insertion of layered protocols into communication channels is presented in Section 7.2. Section 7.3 is devoted to model transformations preserving security. We show what conditions must be ensured such that the security requirements are still fulfilled after such a transformation, and show how these conditions can be proven, both manually and partly automated using SAT solving. Moreover, we give examples for security patterns representing common uses of encryption and/or signatures.

In Section 7.4, we demonstrate our stepwise approach at the example of the bank application case study, where the communication between client and server is tunnelled through an SSL connection. References to related work are given in Section 7.5, and we conclude in Section 7.6 with a summary and discussion.

Early stages of the presented work have been published in [JPW02, GHJW03].

7.1. Application of Security Mechanisms

Stepwise, top-down oriented development of security-critical systems eases reasoning about their security: one can start with a set of global security requirements and an abstract specification and modularise the justification that the security requirements are fulfilled in the low-level design by tracing them through the different

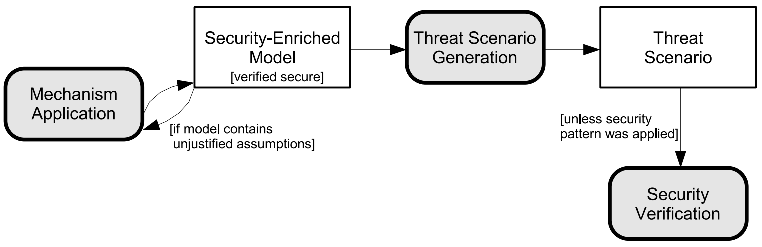

Mechanism Application
Security-Enriched Model
[verified secure]
[if model contains unjustified assumptions]
Threat Scenario Generation
Threat Scenario
[unless security pattern was applied]
Security Verification

Figure 7.1.: Mechanism Application

abstraction levels. The decisions on the use of concrete security mechanisms such as particular authentication protocols or access control methods, or even on key sizes or cryptographic algorithms, should be left open in the first stages of design to allow for maximum flexibility and to be able to reason about the security of the mechanisms and cryptographic algorithms separately. A top-down oriented process especially accounts for the fact that security in general affects the whole system at all levels of abstraction.

In our approach, abstract specifications of security-relevant aspects of a system are integrated into a model in the form of assumptions restricting the intruder's capabilities (see Section 4.2.4). For example, we state that a communication channel cannot be eavesdropped (by the annotation secret) or a component cannot be accessed (by the annotation node), but we do not fix how these assumptions are enforced. In later development stages, it must be decided if

- the assumptions are fulfilled by the environment or by measures out of scope of the model (for example, the communication channel is implemented by a dedicated communication link or the component is placed on a smart card), if
- the corresponding threats can be tolerated (for example, because their realisation requires excessive effort), or if
- an appropriate security mechanism must be applied.

In the context of model-based development, the application of a security mechanism can be seen as a model transformation by which abstract assumptions are eliminated by a mechanism countering the corresponding threats. The model transformation can be carried out manually, or automatically based on additional annotations specifying the security mechanism to be applied (see Section 7.2).

In general, after the application of a security mechanism, the security verification must be repeated, because additional vulnerabilities could have been introduced. For this purpose, a new threat scenario must be derived from the transformed security-enriched model and the security requirements must be verified (see Chapter 5).

Under certain conditions, model transformations for the application of security mechanisms can be defined that preserve the validity of the security requirements. We call these transformations "security patterns", described in further detail in Section 7.3.

Security mechanisms are applied until no more assumptions are left that are not either fulfilled by the environment or by measures out of scope of the model, except if they correspond to risks that can be tolerated. The final design acts as a basis for coding or code generation. Figure 7.1 summarises the described process.

Note that here we are mainly concerned with the application of mechanisms to models that are verified secure. Models that violate a security requirement can be corrected by directly applying security mechanisms (such as the insertion of a protocol layer described in Section 7.2) as well, but also by the revision of security requirements or by weakening threats or strengthening assumptions (see Section 5.3). In this case, the security verification must always be repeated.

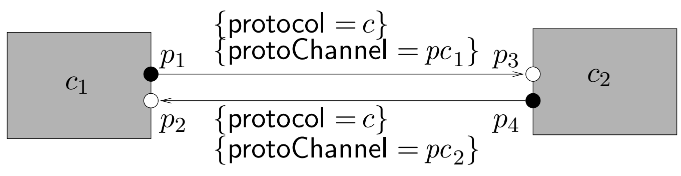
Testing at Different Abstraction Levels

The different stages of the design can also be utilised for testing of an implementation at different levels of abstraction. For example, one can first test the interaction of two components via a channel annotated with secret without testing the mechanism that enforces the confidentiality of the transferred data (for this purpose, the test driver must have access to the messages before they are processed according to the mechanism). If no errors are found, tests also considering the applied mechanism can be performed, by using the transformed model for test sequence generation, for example with the help of test sequences on the level of the encrypted link. Besides, if a separate specification is available, the security mechanism can also be tested separately (in our example, the general establishment of and communication on the encrypted link).

7.2. Insertion of Protocol Layer

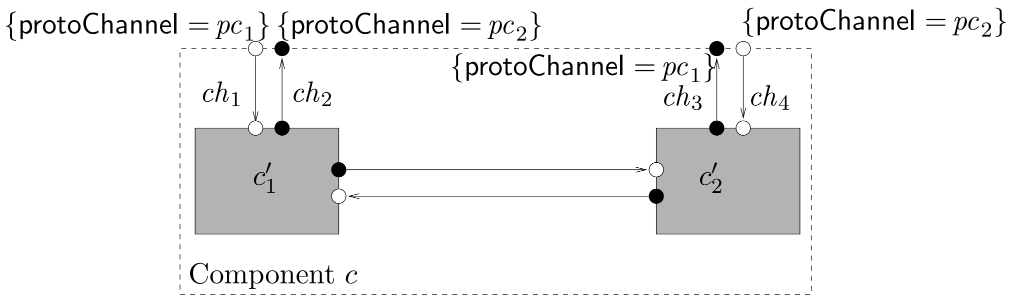
In this section, we describe a model transformation to insert an additional protocol layer into a model over which the communication via selected channels is tunnelled and show at an example how this transformation can be used to realise abstract security assumptions on a channel.

For this purpose, in Section 4.2.5 we presented annotations to specify lower-level protocol layers and their use in a model. Figure 7.2 shows how the model transformation is performed based on the annotations. The annotations $\{\textsf{protocol} = c\}$ on the channels connecting the components c_1 and c_2 in the original model (see



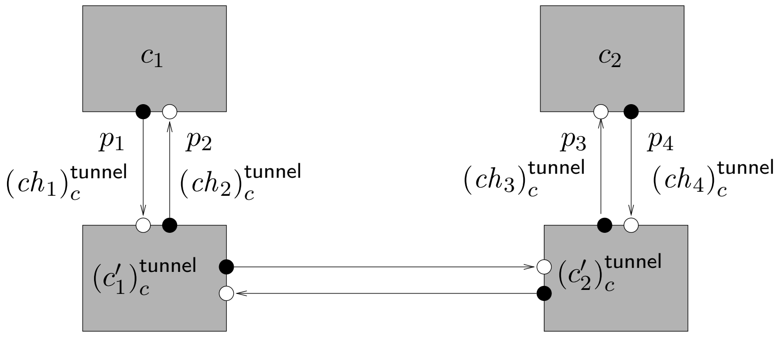
c_1
c_2
p_1
p_2
p_3
p_4
$\{\text{protocol} = c\}$
$\{\text{protoChannel} = pc_1\}$
$\{\text{protocol} = c\}$
$\{\text{protoChannel} = pc_2\}$

(a) Original Model



$\{\text{protoChannel} = pc_1\}$
$\{\text{protoChannel} = pc_2\}$
$\{\text{protoChannel} = pc_2\}$
$\{\text{protoChannel} = pc_1\}$
ch_1
ch_2
ch_3
ch_4
c'_1
c'_2
Component c

(b) Protocol Component with Protocol Layer to be Inserted



c_1
c_2
p_1
p_2
p_3
p_4
$(ch_1)_c^{\text{tunnel}}$
$(ch_2)_c^{\text{tunnel}}$
$(ch_3)_c^{\text{tunnel}}$
$(ch_4)_c^{\text{tunnel}}$
$(c'_1)_c^{\text{tunnel}}$
$(c'_2)_c^{\text{tunnel}}$

(c) Result of Transformation

Figure 7.2.: Model Transformation for Insertion of Protocol Layer

Figure 7.2(a)) indicate that the messages sent via these channels are to be tunneled using the protocol realised by the protocol component c. The protocol component c is depicted in Figure 7.2(b). The protoChannel annotations on the pairs of input and output ports of the component c indicate protocol channels and are referred to by corresponding annotations on the channels of the original model. In the transformed model, a message originally sent via a channel with a $\{\mathsf{protoChannel} = n\}$ annotation is sent to the source port of the corresponding protocol channel, processed by the protocol and then (possibly) delivered to the receiving component via the destination port of the protocol channel. Figure 7.2(c) shows the result of the transformation. The $(c'_i)_c^{\mathsf{tunnel}}$ and $(ch_i)_c^{\mathsf{tunnel}}$ denote copies of the components and channels in c.

In the following, we give a formal definition of the structural part of the transformation "insertion of protocol layer". Let $\mathcal{M}$ be the original model and Main be its top level component. For simplicity, we assume that a library of protocol components is already part of the model and that the protocol components have a non-hierarchical structure. Also, we assume that the Main component contains only channels with the annotation $\{\mathsf{protocol} = c\}$ for the insertion of one instance of the tunnelling protocol specified via the protocol component c and that each channel annotated with $\{\mathsf{protocol} = c\}$ is also annotated with $\{\mathsf{protoChannel} = n\}$ for some n. The insertion of multiple instance of a protocol and the omission of the protoChannel annotation will be addressed later.

In the transformation, the subcomponents of c and their ports, as well as the ports of c not annotated with protoChannel, are copied into Main:

$$\begin{array}{lcl}
\mathsf{Component}^{\mathcal{M}'} & = & \mathsf{Component}^{\mathcal{M}} \cup \{c'^{\mathsf{tunnel}}_c : c' \in \mathsf{subComponents}^{\mathcal{M}}(c)\} \\
\mathsf{subComponents}^{\mathcal{M}'}(\mathsf{Main}) & = & \mathsf{subComponents}^{\mathcal{M}}(\mathsf{Main}) \cup \\
 & & \{c'^{\mathsf{tunnel}}_c : c' \in \mathsf{subComponents}^{\mathcal{M}}(c)\} \\
\mathsf{Port}^{\mathcal{M}'} & = & \mathsf{Port}^{\mathcal{M}} \cup \\
 & & \{p^{\mathsf{tunnel}}_c : p \in \bigcup_{c' \in \mathsf{subComponents}^{\mathcal{M}}(c)} \mathsf{inPorts}^{\mathcal{M}}(c') \cup \\
 & & \qquad\qquad\qquad \mathsf{outPorts}^{\mathcal{M}}(c')\} \cup \\
 & & \{p^{\mathsf{tunnel}}_c : p \in \mathsf{inPorts}^{\mathcal{M}}(c) \cup \mathsf{outPorts}^{\mathcal{M}}(c) \wedge \\
 & & \quad \neg\exists n : (\mathsf{protoChannel}, n) \in \mathsf{tags}^{\mathcal{M}}(p)\} \\
\forall c' \in \mathsf{subComponents}^{\mathcal{M}}(c) : & & \\
\quad \mathsf{inPorts}^{\mathcal{M}'}(c'^{\mathsf{tunnel}}_c) & = & \{p^{\mathsf{tunnel}}_c : p \in \mathsf{inPorts}^{\mathcal{M}}(c')\} \\
\quad \mathsf{outPorts}^{\mathcal{M}'}(c'^{\mathsf{tunnel}}_c) & = & \{p^{\mathsf{tunnel}}_c : p \in \mathsf{outPorts}^{\mathcal{M}}(c')\}
\end{array}$$

Here, the copies of the subcomponents c' of c and of the ports p of the subcomponents c' or of c itself that are created by the insertion of the protocol layer c are denoted with c'^{tunnel}_c and p^{tunnel}_c respectively. The ports of c annotated with protoChannel are not copied, because the channels connected to them are instead connected to the ports of the channels to be tunnelled. Let $\mathsf{TunnelledChannels}(\mathcal{M}, c)$ be the set of channels of Main to be tunnelled via the layered protocol specified by c:

$$\mathsf{TunnelledChannels}(\mathcal{M}, c) \quad = \quad \{ch \in \mathsf{channels}^{\mathcal{M}}(\mathsf{Main}) : (\mathsf{protocol}, c) \in \mathsf{tags}^{\mathcal{M}}(ch)\}$$

Then, the set of channels of the transformed model is the set of channels of the original model with the tunnelled channels removed and copies ch_c^{tunnel} of the channels of c added:

$$\begin{aligned}\mathsf{Channel}^{\mathcal{M}'} \quad = \quad & \mathsf{Channel}^{\mathcal{M}} \setminus \mathsf{TunneledChannels}(\mathcal{M}, c) \cup \\ & \{ch_c^{\mathsf{tunnel}} : ch \in \mathsf{channels}^{\mathcal{M}}(c)\}\end{aligned}$$

It remains to show how the added channels are to be connected. If the source or destination port of a channel ch in the protocol component c is annotated with $\mathsf{protoChannel}$, its copy ch_c^{tunnel} is connected to the source or destination port of the tunnelled channel with the same annotation. Otherwise, ch_c^{tunnel} is connected to the copy of its original source or destination port. Formally:

$$\forall ch \in \mathsf{channels}^{\mathcal{M}}(c) :$$

$$\mathsf{sourceP}^{\mathcal{M}'}(ch_c^{\mathsf{tunnel}}) \quad = \quad \begin{cases} \mathsf{sourceP}^{\mathcal{M}}(ch'), \text{ if} \\ \qquad \exists ch' \in \mathsf{TunneledChannels}(\mathcal{M}, c) : \exists n : \\ \qquad\qquad (\mathsf{protoChannel}, n) \in \mathsf{tags}^{\mathcal{M}}(ch') \wedge \\ \qquad\qquad (\mathsf{protoChannel}, n) \in \mathsf{tags}^{\mathcal{M}}(\mathsf{sourceP}^{\mathcal{M}}(ch)) \\ p_c^{\mathsf{tunnel}}, \text{ otherwise, for } p = \mathsf{sourceP}^{\mathcal{M}}(ch) \end{cases}$$

$$\mathsf{destP}^{\mathcal{M}'}(ch_c^{\mathsf{tunnel}}) \quad = \quad \begin{cases} \mathsf{destP}(ch'), \text{ if} \\ \qquad \exists ch' \in \mathsf{TunneledChannels}(\mathcal{M}, c) : \exists n : \\ \qquad\qquad (\mathsf{protoChannel}, n) \in \mathsf{tags}^{\mathcal{M}}(ch') \wedge \\ \qquad\qquad (\mathsf{protoChannel}, n) \in \mathsf{tags}^{\mathcal{M}}(\mathsf{destP}^{\mathcal{M}}(ch)) \\ p_c^{\mathsf{tunnel}}, \text{ otherwise, for } p = \mathsf{destP}^{\mathcal{M}}(ch) \end{cases}$$

In the example in Figure 7.2, the ports p_1 and p_3 become the source and destination ports of the channels $(ch_1)_c^{\mathsf{tunnel}}$ and $(ch_3)_c^{\mathsf{tunnel}}$, and the ports p_2 and p_4 become the destination and source ports of the channels $(ch_2)_c^{\mathsf{tunnel}}$ and $(ch_4)_c^{\mathsf{tunnel}}$.

This concludes the formal definition of the structural part of the transformation "insertion of protocol layer". Besides, the automata assigned to the subcomponents of the protocol layer must be copied together with their states, transitions and local variables. Ports appearing in the input and output patterns must be replaced by their copies, but no other changes need to be made. A hierarchical substructure of the components of the protocol layer would merely require that the substructure of the respective components is copied into the transformed model as well. A plugin for AutoFocus/Quest also supporting hierarchy was implemented to carry out the described transformation automatically.

Omitted protoChannel Annotations and Insertion of Multiple Protocol Instances

If the protoChannel annotations are omitted, appropriate annotations can be inserted automatically before the transformation. As pointed out in Section 4.2.5, this is only possible if there is only one protocol channel (i.e. one pair of input and output ports) in the protocol component c and only one tunnelled channel in the original model. In this case, the tunnelled channel and the ports of the protocol channel are both annotated with the same (arbitrary) protoChannel annotation.

If multiple instances of a protocol layer are to be inserted, a sequence of transformations as described above is performed. As the set $\mathsf{TunnelledChannels}(\mathcal{M}, c)$, the subset of the channels annotated with $(\mathsf{protocol}, c)$ with the same $(\mathsf{protoInstance}, n)$ tag is taken. For each instance n, new copies of the components, ports and channels of the protocol component c must be created.

Type-Independent Protocol Channels

Often, the behaviour of a protocol layer is independent of the actual type of messages to be transferred via a protocol channel, because the received messages are only forwarded, possibly embedded into larger messages. This makes it possible to insert a protocol layer into a model regardless of the types of messages on the channels to be tunnelled, but requires a careful adaptation of the transformed model.

In a protocol component c, we call a protocol channel of type type^k *type-independent* if no constructors, selectors or discriminators of type type^k appear in the terms on the transitions of the automata of the subcomponents of c. In addition, if a subcomponent of c has a local variable whose values can contain (sub-)terms of type type^k, it must be ensured that a value is written to the variable before it is first read (so that its initial value does not affect the subcomponent's behaviour). The latter condition can in general only be verified dynamically. To be able to verify it statically, one can for instance require additionally that there is a boolean variable indicating if the variable of type type^k has already been written to and that there are preconditions that only allow read access if the boolean variable evaluates to true.

In the transformation, the type of the type-independent protocol channel can be replaced by the type of the tunnelled channel throughout the copied parts of the protocol component c. If there are data type definitions of types $\mathsf{type}^{k'}$ depending on type^k, they must also be copied, type^k must be replaced by the type of the tunnelled channel and $\mathsf{type}^{k'}$ must be replaced by the copy. If necessary, this must be repeated recursively until no more replacements are to be performed.

If there is more than one type-independent protocol channel, it must be ensured that the replacement is carried out consistently. I.e., if two type-independent protocol channels have the same type, the types of the tunnelled channels that replace the types of the protocol channels must also be the same.

For simplicity, in the following we only consider protocol layers where the protocol channels already are of the same type as the channels to be tunnelled through them, such that the above described replacements do not need to be performed.

Example: Shared Key Encryption Layer

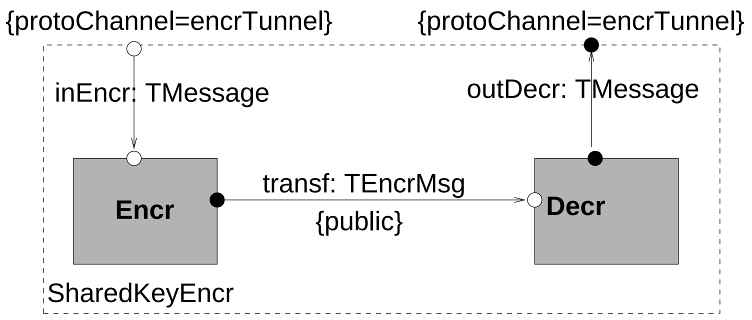
As a simple example of the application of protocol layers in security-critical systems, Figure 7.3 shows the specification of a shared key encryption layer transmitting messages over an unreliable channel. The protocol component SharedKeyEncr (see Figure 7.3(a)) has two subcomponents Encr and Decr, whose behaviour is specified by the STDs in Figure 7.3(b) and Figure 7.3(c) respectively. The data type definitions and function definitions are given in Figure 7.3(d).

The Encr component receives messages of type TMessage from the source port of the protocol channel encrTunnel, encrypts them with the key KED shared between Encr and Decr, and transmits them as encrypted messages of type TEncrMsg to the component Decr via the channel transf, which is annotated with public (i.e., the intruder has full access to this channel). Decr attempts to decrypt the received messages, and if this succeeds forwards them to the destination port of the protocol channel encrTunnel. We assume the key KED is not known to the intruder, as specified by the function declaration of knowsIntruderTKey.

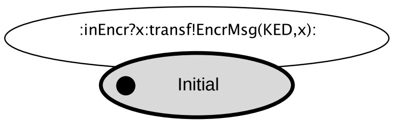
Figure 7.4 shows how this protocol layer can be used to realise a channel requiring confidential and authentic communication. In Figure 7.4(a), a simple model is depicted with a channel AtoB between two components A and B. The annotations secret and auth on the channel state that data sent via this channel is only readable by component B, and if component B receives data via this channel, it is assured that the data has indeed been sent by component A.

If in the implemented system, there can only be an unreliable channel between A and B, the described assumptions can be realised by the shared key encryption layer. In Figure 7.4(b), appropriate annotations have been assigned to the channel for the insertion of the shared key encryption layer by the above described model transformation.

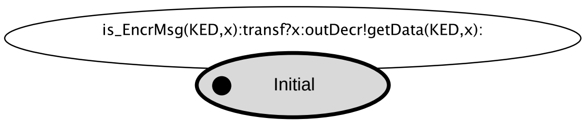
Figure 7.4(c) shows the result of the transformation, where the shared key encryption layer has been inserted (for simplicity, we use the original channel names instead of the copies). Note that here the assumptions secret and auth are not present anymore — the intruder has full access to the channel transmitting the messages between Encr and Decr. However, the intruder must not have access to



(a) Shared Key Encryption Layer (SSD)



(b) Shared Key Encryption Layer (STD Encr)

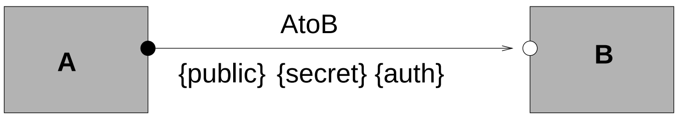


(c) Shared Key Encryption Layer (STD Decr)

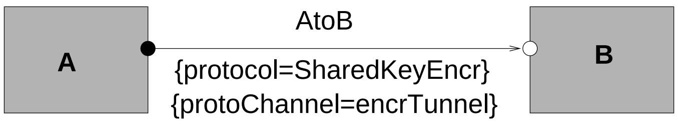
```
data TMessage = EmptyTMessage | MsgA | MsgB;
data TKey = KA | KED | EmptyTKey;
data TEncrMsg = EncrMsg(TKey,getData:TMessage) | EmptyTEncrMsg;
fun invTKey(x:TKey) = x;
fun knowsIntruderTKey(x) = not(x==KED);
```

(d) Shared Key Encryption Layer (DTD)

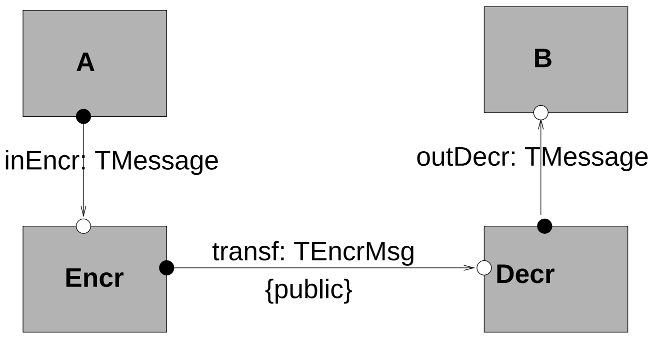
Figure 7.3.: Shared Key Encryption Layer



(a) Original Model (with Assumptions)



(b) Annotations for Insertion of Protocol Layer



(c) Result of Transformation

Figure 7.4.: Application of Shared Key Encryption Layer

the channels between A and Encr, and between Decr and B respectively; otherwise, the intruder could intercept and manipulate the messages directly on these channels. This is reflected by the fact that there are no public annotations on these channels. Also note that the protoChannel annotation could have been omitted as there is only one way of inserting the protocol layer into the channel between A and B.

7.3. Security Patterns

The shared key encryption layer described in the previous section is an example for a security pattern: under specific conditions, a channel annotated with public, secret and auth can be replaced by the shared key encryption layer without it being necessary to repeat the verification of the security requirements.

In general, we understand by a *security pattern* a description of a way of transforming a model such that the validity of the security requirements is preserved. We call the transformation itself a *security-preserving transformation*. By the application of a security pattern, the reasoning about the security of the resulting system is modularised into the reasoning about the security of the original system and the reasoning about the security of the pattern (which needs to be performed only once). This leads to a reduction of the verification effort.

In the following, we describe security-preserving transformations and conditions for proving the preservation of security requirements in more detail, show how such proofs can be supported by bounded model checking, and give small examples for security patterns and how they are applied.

7.3.1. Security-Preserving Transformations

Definition 7.3.1. A *security-preserving transformation* of a model $\mathcal{M}$ with top-level component Main is a transformation such that the set of security requirements does not change and the security requirements that are fulfilled in the threat scenario of $\mathcal{M}$ are also fulfilled in the threat scenario of the transformed model $\mathcal{M}'$.

Let $\mathcal{M}^{\mathsf{T}}$ and $\mathcal{M}'^{\mathsf{T}}$ be the threat scenarios of $\mathcal{M}$ and $\mathcal{M}'$, resulting from the threat scenario generation transformation described in Chapter 5. We need to show that for each attack on the system given by the Main component in $\mathcal{M}'^{\mathsf{T}}$ there is an attack on the corresponding system in $\mathcal{M}^{\mathsf{T}}$.

This condition is fulfilled if the behaviour of Main in $\mathcal{M}'^{\mathsf{T}}$ is a refinement of the behaviour of Main in $\mathcal{M}^{\mathsf{T}}$ with respect to the common state variables. In general, it is desirable to have a refinement instead of mere preservation of the security requirements, because in this case, all other universal trace properties of the original model are preserved as well. In fact, in most cases the transformed model should actually exhibit the *same* behaviour as before (possibly with some delay introduced), such that no functionality is lost. Our examples for security patterns have this property, but corresponding proof strategies are out of scope of this work. Similar techniques as used in Section 5.3.3 for the completeness proof of the specialised intruder model can be applied for the consideration of delays (i.e., systems with a request/reply architecture and attacks with delay).

Additional Intruder Knowledge

The intruder knowledge must be treated in a special way in the refinement. We allow the intruder knowledge in the threat scenario of the original model to be larger than the intruder knowledge in the threat scenario of the transformed model, to account for possible delays introduced by the security mechanisms. In this case, the security requirements are still preserved, provided that properties referring to the intruder knowledge only occur as confidentiality requirements of the form $\mathsf{never}(\mathsf{learnedIntruder}(x))$ (more generally, $\mathsf{learnedIntruder}(x)$ must occur under an odd number of negations). However, often the intruder gains additional knowledge when a security mechanism has been applied, such that there is no trace in the threat scenario of the original model with the same or larger intruder knowledge as in threat scenario of the transformed model. For example, from eavesdropping the public channel in the shared key encryption layer, the intruder learns the encrypted messages. This is not dangerous in itself, but we need to show that the additional knowledge cannot be exploited.

We allow additional knowledge if

- the additional knowledge does not change the values the intruder can send to the other connected components (for instance, because their types do not match the types on the components' interfaces), such that the intruder cannot interfere with the other components, and if
- none of the values in the additional knowledge are mentioned in the security requirements.

If this is not the case, one must add the additional knowledge to the knowledge of the intruder in the threat scenario of the original model and then carry out another verification of the threat scenario of the original model to show that no security requirements are violated. Because the additional knowledge has been included in the threat scenario of the original model, we can now find for each trace in the threat scenario of the transformed model a trace in the threat scenario of the original model with the same or larger intruder knowledge.

Formalisation

Let $[\![\mathsf{Main}]\!]^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}}} = (V^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}};\mathsf{Main}}, I^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}};\mathsf{Main}}, T^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}};\mathsf{Main}})$ be the semantics of the Main component in $\mathcal{M}^{\mathsf{T}}$, where the behaviour of the Intruder component is given by the generic intruder behaviour described in Section 5.2.2 . $\Psi_{[\![\mathsf{Main}]\!]^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}}}}$ is the predicate characterising its set of computations. Besides, we use the same notation for the variables, initial states and transition relation in the semantics of the Intruder component in $\mathcal{M}^{\mathsf{T}}$ and in the semantics of the Main component and of the Intruder component in $\mathcal{M}'^{\mathsf{T}}$. For instance, the set of state variables of the (generic) Intruder component in $\mathcal{M}'^{\mathsf{T}}$ is denoted as $V^{\mathsf{G}}_{\mathcal{M}'^{\mathsf{T}};\mathsf{Intruder}}$. We define a refinement relation $\mathsf{RefineRel}^{\mathsf{G}}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ between traces σ' of the Main component in $\mathcal{M}'^{\mathsf{T}}$ and traces σ of the Main component in $\mathcal{M}^{\mathsf{T}}$ as follows:

$$\begin{aligned}
&\mathsf{RefineRel}^{\mathsf{G}}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}}) = \\
&\quad \forall i : \\
&\qquad \forall v \in (V^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}};\mathsf{Main}} \cap V^{\mathsf{G}}_{\mathcal{M}'^{\mathsf{T}};\mathsf{Main}}) \setminus (V^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}};\mathsf{Intruder}} \cup V^{\mathsf{G}}_{\mathcal{M}'^{\mathsf{T}};\mathsf{Intruder}}) : \\
&\qquad\qquad \sigma(i)(v) = \sigma'(i)(v) \wedge \\
&\qquad \sigma(i)(\mathsf{K}_{\mathsf{Intr}}) \cup \mathsf{AddK}_{\mathsf{Intr}} \supseteq \sigma'(i)(\mathsf{K}_{\mathsf{Intr}})
\end{aligned}$$

I.e., $\mathsf{RefineRel}^{\mathsf{G}}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ holds if σ and σ' have the same values of the common state variables except the variables of the intruder, and the intruder has at most some additional knowledge $\mathsf{AddK}_{\mathsf{Intr}}$ in σ'.

Theorem 7.3.2. *The transformation of $\mathcal{M}$ to $\mathcal{M}'$ is a security-preserving transformation if $\{pr \in \mathsf{properties}^{\mathcal{M}}(\mathsf{Main}) : \mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}}(pr)\} = \{pr \in \mathsf{properties}^{\mathcal{M}'}(\mathsf{Main}) : \mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}'}(pr)\}$ and for all computations σ' of the* Main *component of $\mathcal{M}'^{\mathsf{T}}$, there is a computation σ of the* Main *component of $\mathcal{M}^{\mathsf{T}}$ with $\mathsf{RefineRel}^{\mathsf{G}}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$, and security requirements referring to the intruder knowledge are of the form $\mathsf{never}(\mathsf{learnedIntruder}(x))$, where x is not contained in the additional knowledge $\mathsf{AddK}_{\mathsf{Intr}}$ of the intruder in the original model:*

$$\begin{aligned}
&\exists \mathsf{AddK}_{\mathsf{Intr}} : \\
&\quad \forall \sigma' : \Psi_{[\![\mathsf{Main}]\!]^{\mathsf{G}}_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \Rightarrow \exists \sigma : \Psi_{[\![\mathsf{Main}]\!]^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}}}}(\sigma) \wedge \mathsf{RefineRel}^{\mathsf{G}}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}}) \wedge \\
&\quad \forall pr \in \mathsf{properties}^{\mathcal{M}}(\mathsf{Main}) : (\mathsf{SecRequirement} \in \mathsf{tags}^{\mathcal{M}}(pr) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad \exists x : \mathsf{learnedIntruder}(x) \trianglelefteq pr) \Rightarrow \\
&\qquad \exists x : pr = \mathsf{never}(\mathsf{learnedIntruder}(x)) \wedge x \notin \mathsf{AddK}_{\mathsf{Intr}}
\end{aligned}$$

Proof Let pr be a security requirement. If σ' is a computation violating pr in the system given by the Main component in $\mathcal{M}'^{\mathsf{T}}$, by the above definition one can derive a computation σ violating pr in the threat scenario of the system given by the Main component in $\mathcal{M}^{\mathsf{T}}$. This is because pr must refer to the common state variables or to the intruder knowledge (otherwise, pr cannot be evaluated in both the original and the transformed model). The common state variables have the same values, because $\mathsf{RefineRel}^{\mathsf{G}}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ holds. Besides, if a property of the form $pr = \mathsf{never}(\mathsf{learnedIntruder}(x))$ is violated in σ', x is contained in the intruder knowledge $\sigma'(i)(\mathsf{K}_{\mathsf{Intr}})$ in $\mathcal{M}'^{\mathsf{T}}$ at some execution step i. As $\mathsf{RefineRel}^{\mathsf{G}}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ holds and we required that x is not contained in $\mathsf{AddK}_{\mathsf{Intr}}$, x must be contained in the intruder knowledge $\sigma(i)(\mathsf{K}_{\mathsf{Intr}})$, and thus pr is also violated in $\mathcal{M}^{\mathsf{T}}$. Therefore, according to Definition 7.3.1, the transformation of $\mathcal{M}$ to $\mathcal{M}'$ is security-preserving. □

7.3.2. Security Patterns for Layered Protocols

To show that the application of a layered protocol such as of the shared key encryption layer shown in Figure 7.4 is a security pattern, one must show that the corresponding model transformation is security-preserving. However, in general this is dependent on the possible sequences of messages that are sent over the tunnelled channels (in the example, from component A to component B) and on the possible interference with other attacked parts of the model.

To be able to show that a particular layered protocol is always applicable, we assume the most general components to be connected to the tunneled channels and prove that the corresponding model transformation is a security-preserving transformation for *any* sequence of messages transferred on each of the channels.

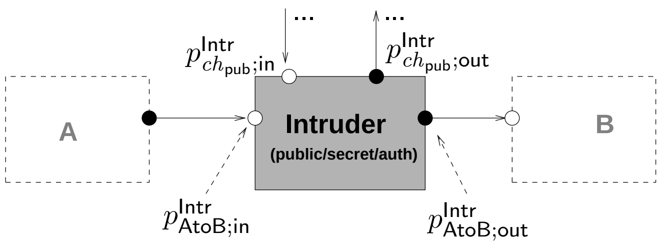
Besides, to account for the other attacked parts of the model, we assume there is a set of additional public channels where arbitrary sequences of messages are transferred for each type of messages for which we assume there can be channels in the original model that can be attacked. Additional assumptions on the public channels, such as secret, auth, integrity or noreplace, only restrict the actions the intruder can perform on (i.e., the data he can read from or write to) such a channel. Thus, if the transformation is security-preserving without such assumptions, it is also security-preserving when the assumptions are present and thus the additional assumptions need not be considered.

Example: Shared Key Encryption Layer

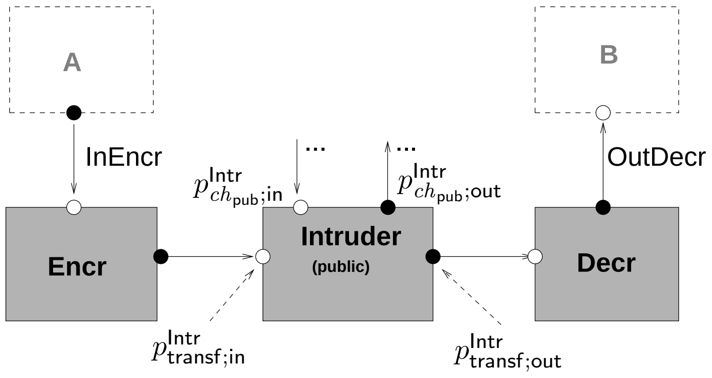
Using the above described concepts, we can show that the insertion of the shared key encryption layer into a channel annotated with public, secret and auth is a security pattern, provided that

- the secret key KED is not derivable from the initial intruder knowledge and from the messages the intruder receives via other attacked channels,
- the security requirements that refer to the intruder knowledge are of the form $\mathsf{never}(\mathsf{learnedIntruder}(x))$,
- the security requirements do not refer to the encrypted terms, i.e. do not contain subterms of the form $\mathsf{learnedIntruder}(\mathsf{Encr}(\mathsf{KED}, x))$, and
- the type TEncrMsg is not used in the original model.

Proof Sketch Figure 7.5(a) shows the threat scenario $\mathcal{M}^{\mathsf{T}}$ of the original model depicted in Figure 7.4(a), including a number of ports for the intruder to read from and write to additional public channels of type TMessage, denoted with $p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{in}}$ and


A
Intruder
(public/secret/auth)
B
$p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{in}}$
$p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{out}}$
$p^{\mathsf{Intr}}_{\mathsf{AtoB;in}}$
$p^{\mathsf{Intr}}_{\mathsf{AtoB;out}}$

(a) Threat Scenario $\mathcal{M}^{\mathsf{T}}$ of Original Model $\mathcal{M}$


A
B
InEncr
OutDecr
Encr
Intruder
(public)
Decr
$p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{in}}$
$p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{out}}$
$p^{\mathsf{Intr}}_{\mathsf{transf;in}}$
$p^{\mathsf{Intr}}_{\mathsf{transf;out}}$

(b) Threat Scenario $\mathcal{M}'^{\mathsf{T}}$ of Transformed Model $\mathcal{M}'$

Figure 7.5.: Threat Scenarios for Application of Shared Key Encryption Layer

$p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{out}}$. We denote the set of additional public channels ch_{pub} by AddPubChannels. Figure 7.5(b) shows the threat scenario $\mathcal{M}'^{\mathsf{T}}$ of the transformed model depicted in Figure 7.4(c), with the same ports for the additional channels. Component A can send and component B can receive any sequence of messages of type TMessage.

Let σ' be a computation of Main in $\mathcal{M}'^{\mathsf{T}}$. From the semantics of the Intruder component in the generic threat scenario described in Section 5.2.2 and from the semantics of the Encr and Decr components, we can conclude that for any $i \geq 0$:

$$\begin{aligned}
&\sigma'(i)(\mathsf{M}_{p^{\mathsf{Intr}}_{\mathsf{transf;in}}}) \subseteq \{\mathsf{EncrMsg}(\mathsf{KED}, x) : x \in \textstyle\bigcup_{0\leq j\leq i-2}\{\sigma'(j)(\mathsf{InEncr})\} \wedge x \neq \perp\}\\
&\sigma'(i)(\mathsf{K}_{\mathsf{Intr}}) \subseteq\\
&\qquad \mathsf{derivable}(\{\mathsf{EncrMsg}(\mathsf{KED}, x) : x \in \textstyle\bigcup_{0\leq j\leq i-2}\{\sigma'(j)(\mathsf{InEncr})\} \wedge x \neq \perp\} \cup\\
&\qquad\qquad \textstyle\bigcup_{0\leq j\leq i-1, ch_{\mathsf{pub}}\in\mathsf{AddPubChannels}}\{\sigma'(j)(p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{in}})\} \cup \mathsf{IK}(\mathcal{M}))
\end{aligned}$$

I.e., the message store $\mathsf{M}_{p^{\mathsf{Intr}}_{\mathsf{transf;in}}}$ for the messages received via $p^{\mathsf{Intr}}_{\mathsf{transf;in}}$ contains the encryptions of the messages that the Encr component received via the channel InEncr until execution step $i-2$ (because the Encr and $\mathsf{Intruder}$ components introduces one time tick delay each). Besides, the intruder knowledge $\mathsf{K}_{\mathsf{Intr}}$ consists of the values that can be derived from these encrypted messages, the messages received from the other attacked channels (with one time tick delay) and the initial intruder knowledge.

As according to the assumptions KED cannot be derived from the messages received from the other attacked channels and the initial intruder knowledge, and because $\mathsf{TEncrMsg}$ does not appear in the original model $\mathcal{M}$, no additional messages can be derived from the encrypted messages $\mathsf{EncrMsg}(\mathsf{KED}, x)$, and thus

$$\begin{aligned}
\sigma'(i)(\mathsf{K}_{\mathsf{Intr}}) \subseteq \{&\mathsf{EncrMsg}(\mathsf{KED}, x) : x \in \textstyle\bigcup_{0\le j\le i-2}\{\sigma'(j)(\mathsf{InEncr})\} \wedge x \neq \perp\} \cup \\
&\mathsf{derivable}(\textstyle\bigcup_{0\le j\le i-1, ch_{\mathsf{pub}}\in\mathsf{AddPubChannels}}\{\sigma'(j)(p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{in}})\} \cup \mathsf{IK}(\mathcal{M}))\\
\sigma'(i)(p^{\mathsf{Intr}}_{\mathsf{transf;out}}) \in \{\perp\} \cup &\\
&\{\mathsf{EncrMsg}(\mathsf{KED}, x) : x \in \textstyle\bigcup_{0\le j\le i-1}\{\sigma'(j)(\mathsf{InEncr})\} \wedge x \neq \perp\} \cup \\
&\mathsf{derivable}(\textstyle\bigcup_{0\le j\le i, ch_{\mathsf{pub}}\in\mathsf{AddPubChannels}}\{\sigma'(j)(p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{in}})\} \cup \mathsf{IK}(\mathcal{M}))
\end{aligned}$$

I.e., the intruder knowledge $\mathsf{K}_{\mathsf{Intr}}$ can be separated into the encrypted messages $\mathsf{EncrMsg}(\mathsf{KED}, x)$ and the values that can be derived from the messages from the other attacked channels and the initial intruder knowledge. From these values, the output of the intruder on port $p^{\mathsf{Intr}}_{\mathsf{transf;out}}$ is chosen. Outputs are performed immediately (see Section 5.2.2), thus the delay of one time tick introduced by the intruder for storing the messages is compensated for by taking the values of $\mathsf{M}_{p^{\mathsf{Intr}}_{\mathsf{transf;in}}}$ and $\mathsf{K}_{\mathsf{Intr}}$ in the following execution step.

The Decr component only accepts messages of the form $\mathsf{EncrMsg}(\mathsf{KED}, x)$ and forwards the decrypted part x. As no messages of the form $\mathsf{EncrMsg}(\mathsf{KED}, x)$ can be derived from the messages from the other attacked channels and the initial intruder knowledge, we have

$$\sigma'(i)(\mathsf{OutDecr}) \in \{\perp\} \cup \textstyle\bigcup_{0\le j\le i-2}\{\sigma'(j)(\mathsf{InEncr})\}$$

I.e., on the channel $\mathsf{OutDecr}$ exactly any of the messages can appear that have been sent to the channel InEncr until up to two time ticks earlier. Finally, as the additional channels are of type $\mathsf{TMessage}$, the messages $\mathsf{EncrMsg}(\mathsf{KED}, x)$ cannot be sent to these channels. Thus,

$$\begin{aligned}
\sigma'(i)(p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{out}}) \in \{\perp\} \cup &\,\mathsf{derivable}(\textstyle\bigcup_{0\le j\le i, ch_{\mathsf{pub}}\in\mathsf{AddPubChannels}}\{\sigma'(j)(p^{\mathsf{Intr}}_{ch_{\mathsf{pub}};\mathsf{in}})\} \cup \\
&\mathsf{IK}(\mathcal{M}))
\end{aligned}$$

I.e., any values derivable from the messages received from the additional channels and the initial knowledge can be sent to the additional channels, but not the values received via the tunneled channel InEncr.

This is the same behaviour as can be derived from the semantics of the threat scenario of the original model with the abstract channel annotated with public, secret and auth depicted in Figure 7.5(a): the intruder cannot add the values sent to him from component A to his knowledge and cannot insert own values into the messages sent to component B. The only possibility are replays. The only difference is that the intruder in the original model can additionally replay the values received during the last two time ticks, because there is no delay as was introduced by the components Encr and Decr. Besides, the intruder in the transformed model has as additional knowledge at most $\mathsf{AddK}_{\mathsf{Intr}} = \{\mathsf{EncrMsg}(\mathsf{KED}, x) : x \in \mathsf{Value}(\mathsf{TMessage})\}$. We assumed that these values do not appear in the security requirements.

Therefore, we can derive from σ' a trace σ of the Main component of $\mathcal{M}^{\mathsf{T}}$ with the same values of the common state variables except the variables of the intruder, and only the additional knowledge $\mathsf{AddK}_{\mathsf{Intr}}$. For this reason, the condition given in Theorem 7.3.2 is fulfilled and thus the introduction of the shared key encryption layer is a security pattern. □

7.3.3. Proof Support Using SAT Solving

The proof that a given model transformation is a security-preserving transformation can be partly automated with the help of SAT solving. For this purpose, we use SAT solving in a similar way as for test sequence generation (see Section 6.2) to compute a trace of the threat scenario of the transformed model that has no corresponding trace in the threat scenario of the original model. If no such trace can be computed, we conclude that the transformation is indeed security-preserving.

First, we choose a suitable set of values for the additional intruder knowledge $\mathsf{AddK}_{\mathsf{Intr}}$, such that none of the values in $\mathsf{AddK}_{\mathsf{Intr}}$ appear in the security requirements.

Using the terminology from Section 7.3.1, traces σ and σ', where σ' is a trace in the transformed threat scenario and σ is a trace which corresponds to σ' via the refinement relation but is no trace of the original threat scenario, fulfil the following predicate:

$$\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \wedge \mathsf{RefineRel}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}}) \wedge \neg\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma)$$

Here, the threat scenarios are the specialised ones (see Section 5.3), to be able to apply SAT solving. I.e., $[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}$ and $[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}$ are used as the semantics of the threat scenario of the transformed and original model rather than $[\![\mathsf{Main}]\!]^{\mathsf{G}}_{\mathcal{M}'^{\mathsf{T}}}$ and $[\![\mathsf{Main}]\!]^{\mathsf{G}}_{\mathcal{M}^{\mathsf{T}}}$. The refinement relation $\mathsf{RefineRel}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ is the adaptation of $\mathsf{RefineRel}^{\mathsf{G}}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ to the state variables of the specialised intruder model.

With the help of SAT solving, we try to compute traces σ, σ' fulfilling the above predicate for a fixed bound l on their length. If this is not possible, the transformation of $\mathcal{M}$ to $\mathcal{M}'$ is security-preserving for traces with length less than or equal to l, as we will show below. However, the converse is not true. If there is more than one trace σ that fulfils $\mathsf{RefineRel}(\sigma, \sigma', \mathsf{AddK_{Intr}})$ for given σ' and $\mathsf{AddK_{Intr}}$, the SAT procedure can determine such a σ that is not a trace of the original threat scenario even though there is another such σ that is. In this case, σ and σ' do not demonstrate that the transformation of $\mathcal{M}$ to $\mathcal{M}'$ is not security-preserving.

This problem can occur because there can be local variables of the intruder in the original model that have no correspondence in the transformed model (such as the $\mathsf{replace}_c$ variables), or because the intruder in the original model has been modified, for example by introducing delays (see below). For this reason, we strengthen $\mathsf{RefineRel}(\sigma, \sigma', \mathsf{AddK_{Intr}})$ to a refinement relation $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}})$ that implies $\mathsf{RefineRel}(\sigma, \sigma', \mathsf{AddK_{Intr}})$ and fixes all state variables in σ for given σ' and $\mathsf{AddK_{Intr}}$. Besides, for any σ', there must always be a σ such that $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}})$ is fulfilled. The strengthened $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}})$ must be determined manually such that the "right" σ is fixed for a given σ' and thus the SAT solver does not produce spurious solutions. If there is no solution, we have a security-preserving transformation, because

$$\begin{gathered}
\neg\left(\exists\sigma' : \exists\sigma : \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \wedge \mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}}) \wedge \neg\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma)\right) \\
\Rightarrow \\
\forall\sigma' : \forall\sigma : \neg\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \vee \neg\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}}) \vee \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma) \\
\Rightarrow \\
\forall\sigma' : \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \Rightarrow \forall\sigma : \mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}}) \Rightarrow \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma) \\
\Rightarrow^{(*)} \\
\forall\sigma' : \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \Rightarrow \exists\sigma : \mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}}) \wedge \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma) \\
\Rightarrow^{(**)} \\
\forall\sigma' : \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \Rightarrow \exists\sigma : \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma) \wedge \mathsf{RefineRel}(\sigma, \sigma', \mathsf{AddK_{Intr}})
\end{gathered}$$

The last line corresponds to the condition for security-preserving transformations given in Theorem 7.3.2 using the generic threat scenarios and refinement relation. The implication marked (*) is fulfilled because as assumed above, for any σ' there is a σ such that $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}})$ is fulfilled, and $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}}) \Rightarrow \Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma)$ is fulfilled for all σ. The implication marked (**) is fulfilled because $\mathsf{RefineRel}'$ implies $\mathsf{RefineRel}$.

Propagation to Generic Threat Scenario

If the specialised intruder models are sound and complete with respect to the threat scenarios $\mathcal{M}^{\mathsf{T}}$ and $\mathcal{M}'^{\mathsf{T}}$ (see Section 5.3.3) and l was chosen sufficiently large such that no attacks are missed, the result from SAT solving can be propagated to the generic threat scenario. This is because for each trace σ'^{G} violating a security requirement *pr* in the generic threat scenario of the transformed model, there is a trace σ' in the specialised threat scenario of the transformed model such that *pr* is violated (because of completeness of the specialised intruder model with respect to the threat scenario of the transformed model). Due to the result from SAT solving and Theorem 7.3.2 (adapted to the specialised versions of the intruder and the refinement relation), there is a trace σ of the specialised threat scenario of the original model where *pr* is violated. Therefore, there is a trace σ^{G} of the generic threat scenario of the original model where *pr* is violated (because of soundness of the specialised intruder model with respect to the threat scenario of the original model), and thus the transformation is security-preserving with respect to the generic threat scenario as well.

Restriction of Intruder Behaviour

In some cases, finding an appropriate refinement relation can be made easier by first showing that the transformation is security-preserving with respect to a restricted version of the intruder in the original model. For instance, if the security pattern introduces delays into the communication on a channel, a restricted version of the intruder in the original model can be specified that introduces the same delays. If the transformation is security-preserving with respect to a restricted version of the intruder in the original model, it is also security-preserving with respect to the non-restricted version of the intruder in the original model. We will demonstrate this strategy below, at the example of the shared key encryption layer.

Summary

Altogether, to prove that a transformation is security-preserving, the following steps are to be taken:

- Choose the additional intruder knowledge in the transformed model $\mathsf{AddK}_{\mathsf{Intr}}$ appropriately.
- Possibly restrict the intruder's behaviour in the original model.
- Define a strengthened refinement relation $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ that fixes all state variables in σ for given σ'.
- Use SAT solving to compute solutions with length up to bound l for

$$\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \wedge \mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}}) \wedge \neg\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma)$$

- If there are no solutions, the transformation of $\mathcal{M}$ to $\mathcal{M}'$ is security-preserving for traces with length less than or equal to l. Otherwise, σ' is a trace in the transformed model which has no corresponding trace (via the refinement relation) in the original model. Either the additional knowledge, the strengthened refinement relation or the transformation itself need to be modified and the given steps need to be repeated. Note that this way, errors in the additional conjuncts of the strengthened refinement relation are detected, and thus the validity of the proof does not depend on their correctness. A good strategy to find errors is to gradually decrease l to find the shortest trace for which a counterexample is produced, and to carefully examine the last step of this counterexample, which must be the cause for the error.

Example: Shared Key Encryption Layer

In the following, we apply the above described method to the insertion of the shared key encryption layer.

We choose the same additional intruder knowledge $\mathsf{AddK_{Intr}}$ as in the manual proof in Section 7.3.2:

$$\mathsf{AddK_{Intr}} = \{\mathsf{EncrMsg}(\mathsf{KED}, x) : x \in \mathsf{Value}(\mathsf{TMessage})\}$$

Besides, we restrict the intruder's behaviour in the threat scenario of the original model such that a value sent by component A is only available to the intruder after an additional delay of one time tick, and likewise component B can only receive a value from the intruder after an additional delay of one time tick. This makes it easier to specify a correspondence between the state variables of the intruder in the original model and the intruder in the transformed model, where the same delay is introduced by the Encr and Decr components. Figure 7.6 shows the restricted threat scenario of the original model. Here, the components D1 and D2 accept messages on their input ports and forward them with one time tick delay. For the threat scenario of the transformed model, see Figure 7.5(b) on page 185.

For simplicity, we do not consider additional public channels, because the generation of the specialised intruder model currently requires that all attacked channels have the same type (see Section 5.3.2). However, the treatment of additional public channels would not introduce additional complexity, but merely require that the correspondence of the values on the ports of these channels in the original model and in the transformed model is stated in the refinement relation.

We now explain the conjuncts in the strengthened refinement relation $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK_{Intr}})$. For all $i \leq l$ (where l is the maximum trace length), the values sent by component A, respectively received from component B, must be the same in the threat scenario of the original model as in the threat scenario of the transformed model:

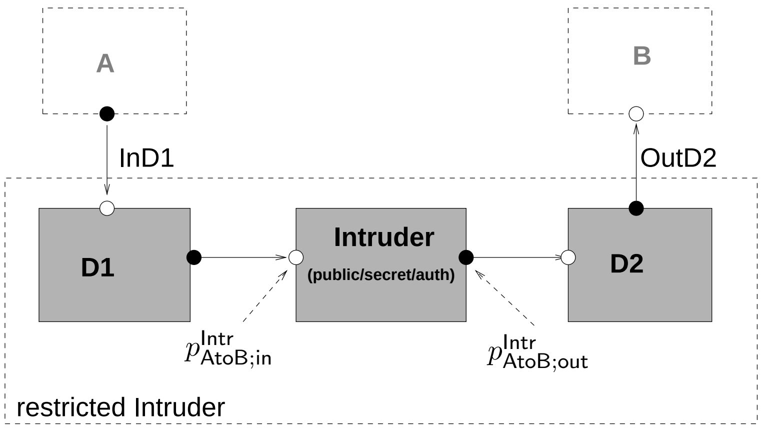

A
B
InD1
OutD2
D1
Intruder
(public/secret/auth)
D2
$p^{\mathsf{Intr}}_{\mathsf{AtoB;in}}$
$p^{\mathsf{Intr}}_{\mathsf{AtoB;out}}$
restricted Intruder

Figure 7.6.: Shared Key Encryption Layer: Threat Scenario $\mathcal{M}^{\mathsf{T}}$ for Original Model $\mathcal{M}$ (restricted)

$$\begin{array}{lcl} \sigma(i)(\mathsf{InD1}) & = & \sigma'(i)(\mathsf{InEncr}) \\ \sigma(i)(\mathsf{OutD2}) & = & \sigma'(i)(\mathsf{OutDecr}) \end{array}$$

The knowledge of the intruder in the threat scenario of the original model must be the same as the knowledge of the intruder in the threat scenario of the transformed model, with the exception of the messages in $\mathsf{AddK}_{\mathsf{Intr}}$:

$$\begin{array}{lcl} \sigma(i)(\mathsf{store}_{\mathsf{TMessage};1}) & = & \sigma'(i)(\mathsf{store}_{\mathsf{TMessage};1}) \\ \sigma(i)(\mathsf{store}_{\mathsf{TKey};1}) & = & \sigma'(i)(\mathsf{store}_{\mathsf{TKey};1}) \\ \sigma(i)(\mathsf{store}_{\mathsf{TEncrMsg};1}) & = & \begin{cases} \mathsf{EmptyTEncrMsg}, & \text{if } \sigma'(i)(\mathsf{store}_{\mathsf{TEncrMsg};1}) \in \mathsf{AddK}_{\mathsf{Intr}} \\ \sigma'(i)(\mathsf{store}_{\mathsf{TEncrMsg};1}), & \text{otherwise} \end{cases} \end{array}$$

In the example, the specialised intruder in the threat scenarios for both the original and the transformed models can store one message for each type in the variables $\mathsf{store}_{\mathsf{TMessage};1}$, $\mathsf{store}_{\mathsf{TKey};1}$ and $\mathsf{store}_{\mathsf{TEncrMsg};1}$ (see Section 5.3.2). Messages in $\mathsf{AddK}_{\mathsf{Intr}}$ (which are of type $\mathsf{TEncrMsg}$) do not need to be contained in the knowledge of the intruder of the threat scenario of the original model and thus are mapped to $\mathsf{EmptyTEncrMsg}$.

The conjunction of the above terms forms the refinement relation $\mathsf{RefineRel}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$. As mentioned above, to be able to use SAT solving to verify if we have a security-preserving transformation, all state variables in the trace σ of the original model must be fixed. For this purpose, we strengthen $\mathsf{RefineRel}(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ to $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ by adding the following conjuncts:

$$\begin{array}{lcl}
\sigma(i)(p^{\mathsf{Intr}}_{\mathsf{AtoB;in}}) & = & \begin{cases} \bot\,, \quad \text{if } \sigma'(i)(p^{\mathsf{Intr}}_{\mathsf{transf;in}}) = \bot \\ \mathsf{getData}(\mathsf{KED}, \sigma'(i)(p^{\mathsf{Intr}}_{\mathsf{transf;in}})), \quad \text{otherwise} \end{cases} \\
\sigma(i)(p^{\mathsf{Intr}}_{\mathsf{AtoB;out}}) & = & \begin{cases} \bot\,, \quad \text{if } \sigma'(i)(p^{\mathsf{Intr}}_{\mathsf{transf;out}}) = \bot \\ \mathsf{getData}(\mathsf{KED}, \sigma'(i)(p^{\mathsf{Intr}}_{\mathsf{transf;out}})), \quad \text{otherwise} \end{cases} \\
\sigma(i)(\mathsf{M}_{p^{\mathsf{Intr}}_{\mathsf{AtoB;in}};1}) & = & \mathsf{getData}(\mathsf{KED}, \sigma'(i)(\mathsf{store}_{\mathsf{TEncrMsg};1}))
\end{array}$$

I.e., if a message encrypted with KED is sent by the Encr component respectively received by the Decr component in the threat scenario of the transformed model, in the corresponding trace in the threat scenario of the original model the cleartext is sent respectively received by the delay components D1 and D2. Besides, the capability of the intruder in the threat scenario of the transformed model to replay a message of type TEncrMsg encrypted with KED (stored in $\mathsf{store}_{\mathsf{TEncrMsg};1}$) corresponds to the capability of the intruder in the threat scenario of the original model to replay the cleartext (stored in $\mathsf{M}_{p^{\mathsf{Intr}}_{\mathsf{AtoB;in}};1}$). Messages not encrypted with KED are blocked by the Decr component in the threat scenario of the transformed model and thus in the corresponding trace in the threat scenario of the original model, no message is sent.

Finally, we used SAT solving to find solutions for

$$\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}'^{\mathsf{T}}}}(\sigma') \wedge \mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}}) \wedge \neg\Psi_{[\![\mathsf{Main}]\!]_{\mathcal{M}^{\mathsf{T}}}}(\sigma)$$

with the help of an extension of the translation of AutoFocus/Quest models to formulas in propositional logic described in [WLPS00]. As no solutions can be found, we conclude that the introduction of the shared key encryption layer is indeed a security-preserving transformation. In case of this (admittedly simple) example, performance is not an issue — even for the bound $l = 50$ the SAT solver Chaff terminates in less than 3s on the used system.

7.3.4. Further Patterns for Encryption/Signature

In this section, we give further examples for modelling common uses of encryption and/or signature of transferred messages as layered protocols and state the abstract assumptions on the tunnelled channels these layered protocols can be inserted into. These are examples for security patterns a designer applying a model-based development methodology can be provided with as part of a library from which appropriate security patterns can be selected during the development. A more complex security pattern is described in Section 7.4.

A summary of the security patterns described in this and the previous sections is shown in Table 7.1. In addition to the name of the security pattern, the abstract

Table 7.1.: Encryption/Signature Patterns for Abstract Channels

Security Pattern	Realised Assumptions	Further Conditions
Shared Key Encryption	secret, auth, integrity	confidentiality of shared key in $\mathcal{M}$
Public Key Encryption	secret	confidentiality of secret key in $\mathcal{M}$
Message Authentication Code	auth, integrity	confidentiality of MAC key in $\mathcal{M}$
Public Key Signature	auth, integrity	confidentiality of secret key in $\mathcal{M}$
Public Key Signature + Encryption	secret, auth integrity	confidentiality of secret encryption / signature keys in $\mathcal{M}$

assumptions on a channel it can be applied to are given, as well as further conditions that need to be ensured such that the corresponding model transformation is security-preserving. Besides, as in case of the shared key encryption layer described in the previous sections, we require that the encrypted or signed messages transferred by the protocol layer are not derivable from the initial intruder knowledge and the messages the intruder receives via the other attacked channels (i.e., the encrypted or signed messages are confidential in the original model $\mathcal{M}$), and that the type of these messages is not used in the original model.

All patterns have a similar architecture as the shared key encryption layer (see Figure 7.3): the messages to be sent via the tunnelled channel are received from the sender via a channel not accessible by the intruder, encrypted and/or signed and transferred via a public channel the intruder has full access to. The messages received from the public channel are decrypted and/or their signature is verified and then are possibly forwarded to the receiver via a channel not accessible by the intruder.

Using the concepts described in Section 7.3.2, we proved that the application of each of the patterns is a security-preserving transformation provided that the given conditions are fulfilled. For space reasons, we confine ourselves in the following to giving a brief informal description and justification for each security pattern.

Shared Key Encryption This pattern was described in detail in the previous sections.

Public Key Encryption In the public key encryption layer, the messages are encrypted with a public key, transferred via the public channel and then decrypted with the corresponding secret key. If the secret key is confidential in the original model, the messages transferred on the public channel cannot be decrypted by the intruder, i.e. the transferred data stays confidential (corresponding to the assumption secret on the tunnelled channel in the original model). Authenticity or integrity is not provided by the public key encryption layer, because the intruder can encrypt own messages with the public key, which he may possess. Availability is not provided either, as the intruder on the public channel can intercept the transferred messages. If availability should be provided on the level of the abstract channel, the public channel over which the messages are transferred by the protocol layer must provide availability as well and the notion of availability used on the level of the abstract channel must allow for the delays introduced by the protocol layer. The same observations with regard to availability apply to the other patterns described in this section. Therefore, we omit this aspect in the following.

Message Authentication Code Here, the messages are transferred in clear, together with a message authentication code depending on a shared key. On reception, the message authentication code is verified. The message authentication code acts as a kind of signature ensuring that if the shared key is confidential in the original model, the intruder can only insert replays into the public channel. This corresponds to the assumption auth on the tunnelled channel. Integrity (assumption integrity) is provided as well, because in our definition it is implied by authenticity. Confidentiality of the messages themselves is not provided.

Public Key Signature If the messages transferred via the public channel include the cleartext and a signature based on a secret key, the layered protocol realises the assumptions auth and integrity as well, presumed that the secret key is confidential in the original model. The main difference to the use of a message authentication code is that it is ensured that the receiving component cannot generate signed messages on its own, and therefore that if a signed message is received, its cleartext must indeed have been passed to the tunnelled channel by the sender, even if the receiver has been compromised (non-repudiation). Such a scenario can be examined by additionally giving the intruder access to the secrets of the receiving component in the layered protocol. In this case, the public key signature still realises the assumptions auth and integrity on the tunnelled channel, but the message authentication code does not. Again, confidentiality is not provided.

Public Key Signature + Encryption This pattern is a combination of public key encryption and public key signature, i.e. the messages transferred are signed based on a secret key and then encrypted with a public key (using two different key pairs). Because the intruder can neither decrypt the messages on the public channel nor insert messages other than replays, both the assumptions auth and integrity and the assumption secret on the tunnelled channel are realised. As in the public key signature pattern, non-repudiation is provided as well.

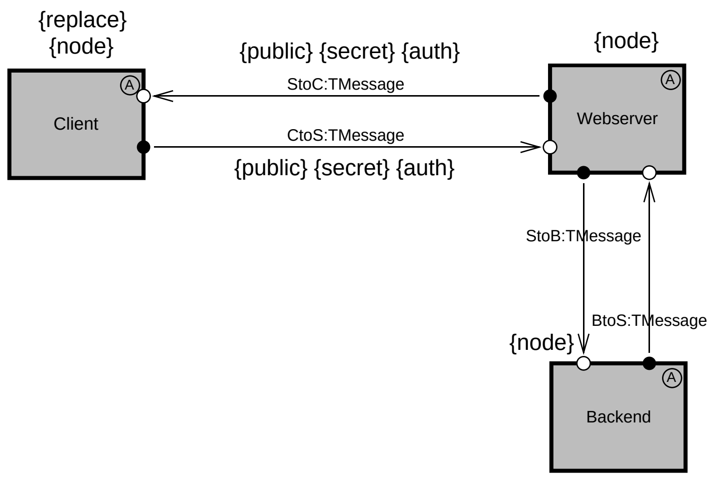


Figure 7.7.: Bank Application: Security-Enriched Model (SSD, modified)

7.4. Case Study: The Bank Application

In the following, we demonstrate the techniques described in this chapter at the example of a more comprehensive case study than the shared key encryption layer: we describe modelling and verification of the bank application presented in Section 4.3 together with an SSL protocol layer over which the communication between the client and the Web server is tunnelled.

7.4.1. The Bank Application Revisited

Reconsider the abstract model of the bank application described in Section 4.3 for requesting and managing electronic order forms, structured into a component Client representing the client's computer and Web browser, a component Webserver, representing the Web server and application server, and a component Backend for the back-end system and its data bases.

As specified in the abstract security-enriched model (cf. Figure 7.7, a slightly modified version of Figure 4.24 on page 78 for reasons given below), we assumed that the connection between Client and Webserver is confidential, integrity protected and authenticated on the server side, reflected by the assumption replace annotated to the Client component indicating that the only possible threat is an intruder masquerading as a legitimate client. We verified the threat scenario of the abstract security-enriched model using threat scenario generation and model checking and showed that the stated security requirements are fulfilled (see Section 5.4.1).

However, in the implemented system the client and the Web server communicate over the Internet. As ordinary TCP/IP connections do not ensure the required assumptions, an appropriate secure connection must be established first. In the bank application, the communication between the client and the Web server is tunnelled over an SSL connection with server authentication.

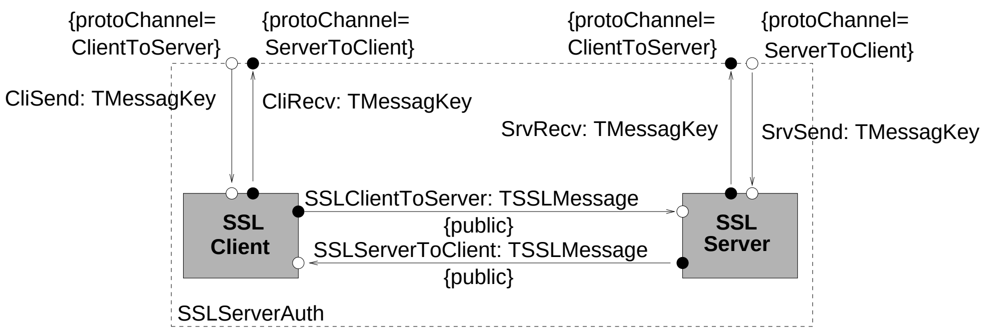
In the forthcoming sections, we describe a model of an SSL protocol layer to be inserted into the channel between the Client and Webserver components and show that this transformation preserves the bank application's security. Unlike the actual SSL protocol, the modelled SSL protocol does not prevent replays, because it is designed to transfer an unbounded number of messages but the number of possible values for sequence numbers necessary to prevent replays would have to be finite to allow model checking. Therefore, to achieve a security-preserving transformation, we also allow replays in the original security-enriched model. This is achieved by annotating the channels between Client and Webserver with public, secret and auth. I.e., if a legitimate client communicates with the Web server, the intruder can access these channels, but only to insert replays, not to insert own messages or to add transferred messages to his knowledge. As the model of the bank application describes a single transaction, the security requirements are still fulfilled with this modification. We verified this fact using model checking.

7.4.2. SSL Layer (Server Authentication)

Figure 7.8 shows the AutoFocus/Quest model of the SSL layer with server authentication. For simplicity, we focus on the essential parts of the SSL handshake and record (i.e. transmission) protocols (see [FKK96]). Error handling, time stamps, session identifiers, negotiation of a cipher suite and compression algorithm, derivation of the session key from a pre-master key and the transmission of a server certificate have been omitted. Some of these aspects are treated in the AutoFocus/Quest model of the SSL protocol described in [Grü03]. Besides, sequence numbers have been left out to allow the transfer of an unbounded number of messages and still keep the data types in the model finite such that it can be verified by model checking or translated to propositional logic. As pointed out above, this enables the possibility of replays, which we therefore also assumed possible in the channels between client and Web server in the threat scenario of the model of the bank application.

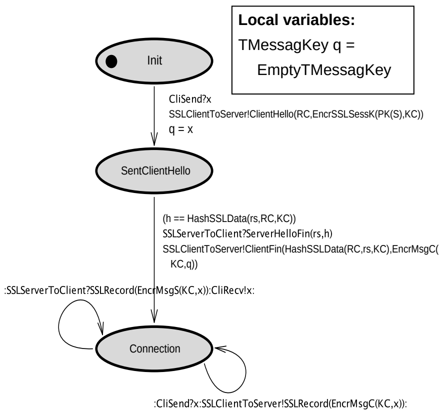
Model

The SSL layer consists of a client side (component SSLClient) and a server side (component SSLServer), connected by a pair of public channels, which can be accessed and manipulated by the intruder. The client side and the server side accept messages of type TMessagKey from the client, respectively from the server, to be transmitted to the other end of the connection. The SSL protocol messages ex-

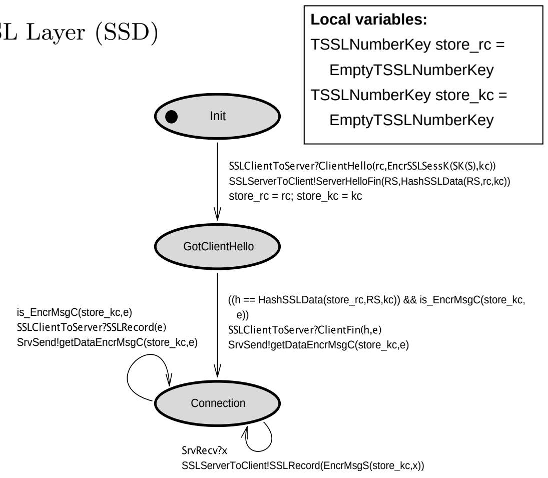


{protoChannel= ClientToServer}
{protoChannel= ServerToClient}
{protoChannel= ClientToServer}
{protoChannel= ServerToClient}
CliSend: TMessagKey
CliRecv: TMessagKey
SrvRecv: TMessagKey
SrvSend: TMessagKey
SSL Client
SSL Server
SSLClientToServer: TSSLMessage
{public}
SSLServerToClient: TSSLMessage
{public}
SSLServerAuth

(a) SSL Layer (SSD)



Local variables:
TMessagKey q = EmptyTMessagKey
Init
CliSend?x
SSLClientToServer!ClientHello(RC,EncrSSLSessK(PK(S),KC))
q = x
SentClientHello
(h == HashSSLData(rs,RC,KC))
SSLServerToClient?ServerHelloFin(rs,h)
SSLClientToServer!ClientFin(HashSSLData(RC,rs,KC),EncrMsgC(KC,q))
:SSLServerToClient?SSLRecord(EncrMsgS(KC,x)):CliRecv!x:
Connection
:CliSend?x:SSLClientToServer!SSLRecord(EncrMsgC(KC,x)):



Local variables:
TSSLNumberKey store_rc = EmptyTSSLNumberKey
TSSLNumberKey store_kc = EmptyTSSLNumberKey
Init
SSLClientToServer?ClientHello(rc,EncrSSLSessK(SK(S),kc))
SSLServerToClient!ServerHelloFin(RS,HashSSLData(RS,rc,kc))
store_rc = rc; store_kc = kc
GotClientHello
((h == HashSSLData(store_rc,RS,kc)) && is_EncrMsgC(store_kc, e))
SSLClientToServer?ClientFin(h,e)
SrvSend!getDataEncrMsgC(store_kc,e)
is_EncrMsgC(store_kc,e)
SSLClientToServer?SSLRecord(e)
SrvSend!getDataEncrMsgC(store_kc,e)
Connection
SrvRecv?x
SSLServerToClient!SSLRecord(EncrMsgS(store_kc,x))

(b) SSL Layer (STD Client) (c) SSL Layer (STD Server)

data TSSLAgent = SSL_C | SSL_S | SSL_A;
data TSSLAKey = PK_SSL(TSSLAgent) | SK_SSL(TSSLAgent) | EmptyTSSLAKey;
data TSSLNumberKey = EmptyTSSLNumberKey | RC | RS | RA | KC | KA;
data TMessagKey = EmptyTMessagKey | MsgA | MsgB | MsgC;

data TSSLMessage = EmptyTSSLMessage
| ClientHello(TSSLNumberKey, TEncrSSLSessK)
| ServerHelloFin(TSSLNumberKey, THashSSLData)
| ClientFin(THashSSLData,TSSLRecord)
| SSLRecord(TSSLRecord);
data TEncrSSLSessK = EncrSSLSessK(TSSLAKey,TSSLNumberKey);
data THashSSLData = HashSSLData(TSSLNumberKey,TSSLNumberKey,TSSLNumberKey);
data TSSLRecord = EncrMsgC(TSSLNumberKey,getDataEncrMsgC: TMessagKey)
| EncrMsgS(TSSLNumberKey,getDataEncrMsgS: TMessagKey);

(d) SSL Layer (DTD)

Figure 7.8.: SSL Layer (Server Authentication)

changed between the client and the server side have the type TSSLMessage.

At the beginning of the execution, the client side of the SSL layer waits for the reception of a message from the client to be transmitted to the server. The received message is stored in a local variable q and the SSL handshake is initiated by sending a message ClientHello(RC, EncrSSLSessK(PK_SSL(SSL_S), KC) towards the server side of the SSL layer via a public channel. Here, RC represents a nonce, and EncrSSLSessK(PK_SSL(SSL_S), KC) is the client's session key KC encrypted with the server's public key PK_SSL(SSL_S). The server side waits for such a ClientHello message. If the encrypted part can be decrypted with the server's secret key SK_SSL(SSL_S), the client's key and the nonce are stored and a random nonce RS is sent towards the client side together with a hash of RS and the client's nonce and key (message ServerHelloFin). On reception of this message and successful verification of the hash, the client side sends a message ClientFin, which includes a hash of both nonces and the session key (in another order to prevent replays). In addition, in this message also the first data record is transferred: EncrMsgC(KC, q), i.e. the message stored in q encrypted with the session key KC. When the server side receives this message with a correct hash and the data record can be decrypted with the received session key, the cleartext is forwarded to the server. After this, the SSL handshake is finished and both the client and server side are in the Connection state. In the following, messages are received from the client, transmitted by the client side of the SSL protocol to the server side encrypted with the session key (SSLRecord messages), decrypted by the server side and forwarded to the server if the decryption succeeds. In the same way, messages are transmitted from the server to the client. Note that different constructors EncrMsgC and EncrMsgS are used for the encrypted messages generated by the client side and by the server side, such that it is possible to distinguish by which side an encrypted message was generated. Even without sequence numbers, this prevents the intruder from reflecting a message to the sender. In the actual SSL protocol, two different keys derived from the same session key are used for this purpose.

Security Analysis

We assume that the intruder knows all public keys PK_SSL(x), his own secret key SK_SSL(SSL_A), an empty/illegal asymmetric key EmptyTSSLAKey, a nonce RA, a session key KA and an empty/illegal symmetric key or nonce EmptyTSSLNumberKey. In addition, the intruder can generate an empty/illegal message EmptyTMessagKey. Other data is only available to the intruder if it can be derived from messages obtained from attacked channels. In particular, the messages MsgA, MsgB, and MsgC of type TMessagKey that can be transmitted between the client and the server are not guessable, because TMessagKey $\in \mathsf{T}_{\mathsf{Key}}$. The intruder knowledge is specified by appropriate function specifications for knowsIntruder^k functions, which we left out in Figure 7.8.

The model of the SSL protocol layer was analysed using the method for threat scenario generation and verification by model checking described in Chapter 5. We verified that the key KC can never be obtained by the intruder (confidentiality of KC) and that a message MsgA, MsgB or MsgC can only be received by the client if it was sent before by the server (server authentication). Client authentication is not provided, which is demonstrated by an appropriate counterexample. Performance figures are given in Table 5.1 on page 120.

However, this is not sufficient to show that the insertion of the SSL protocol layer into the model of the bank application indeed preserves the security requirements. For example, we did not show that the client will not receive any messages at all once the server has accepted a message from the intruder.

7.4.3. Insertion of SSL Layer

The SSL layer is inserted into the model of the bank application by annotating the channels StoC and CtoS with (protocol, SSLServerAuth) and by annotating the channel CtoS with (protoChannel, ClientToServer) and the channel StoC with (protoChannel, ServerToClient). The application of the transformation described in Section 7.2 results in a combined model where the messages exchanged between the components Client and Webserver are tunnelled over the SSL layer. For this purpose, the message type TMessagKey must be replaced by the type TMessage used by the bank application for messages to be transferred between Client and Webserver.

The resulting model can be used for simulation, verification, code generation and classical test sequence generation purposes. Figure 7.9 shows the beginning of an EET of a simulation run of the combined model, corresponding to the EET in Figure 4.21 on page 74.

On the other hand, the threat scenario generated from the combined model is too complex to allow model checking or generation of vulnerability tests using the available model checker / SAT solver interfaces.

7.4.4. Layered Verification

To show that the model of the bank application after the insertion of the SSL layer preserves the security requirements, we proved that the insertion of the SSL layer into general systems with the same assumptions concerning the tunnelled channels (annotations public, secret and auth) and the components connected to them (annotation replace on the component connected to the client side) is a security pattern. For this purpose, we used the techniques described in Section 7.3, in particular the SAT solving approach described in Section 7.3.3.

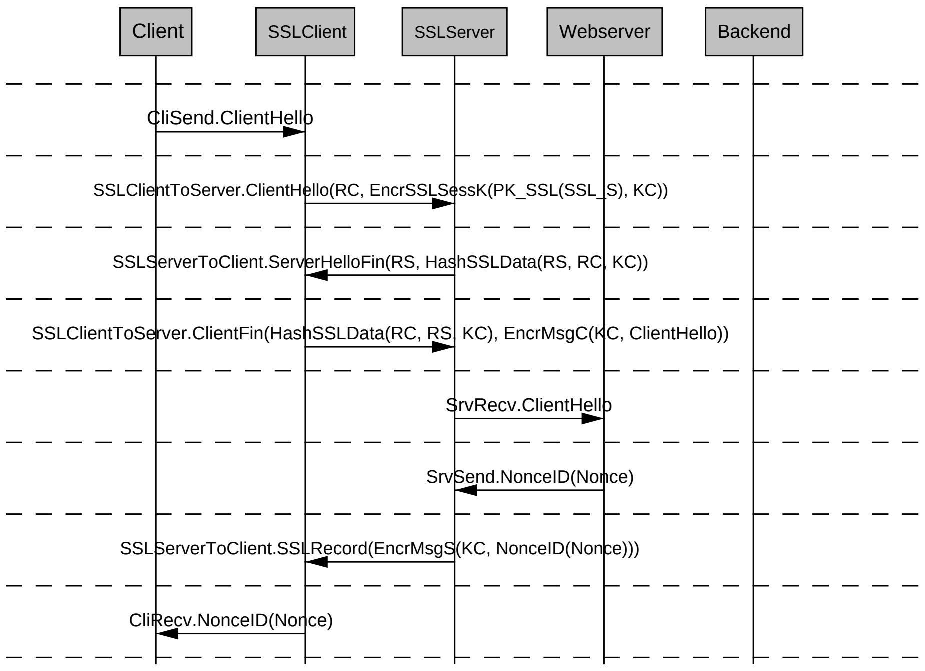

Client
SSLClient
SSLServer
Webserver
Backend
CliSend.ClientHello
SSLClientToServer.ClientHello(RC, EncrSSLSessK(PK_SSL(SSL_S), KC))
SSLServerToClient.ServerHelloFin(RS, HashSSLData(RS, RC, KC))
SSLClientToServer.ClientFin(HashSSLData(RC, RS, KC), EncrMsgC(KC, ClientHello))
SrvRecv.ClientHello
SrvSend.NonceID(Nonce)
SSLServerToClient.SSLRecord(EncrMsgS(KC, NonceID(Nonce)))
CliRecv.NonceID(Nonce)

Figure 7.9.: Bank Application / SSL Layer: Combined Model (EET)

Additional Knowledge

The additional knowledge the intruder can gain from eavesdropping on the channels between SSLClient and SSLServer consists at most of the values of the additional types introduced by the SSL layer:

$$\begin{aligned}\mathsf{AddK}_{\mathsf{Intr}} = \;& \mathsf{Value}(\mathsf{TSSLMessage}) \cup \mathsf{Value}(\mathsf{TEncrSSLSessK}) \cup \\ & \mathsf{Value}(\mathsf{THashSSLData}) \cup \mathsf{Value}(\mathsf{TSSLRecord}) \cup \\ & \mathsf{Value}(\mathsf{TSSLNumberKey}) \cup \mathsf{Value}(\mathsf{TSSLAKey}) \cup \mathsf{Value}(\mathsf{TSSLAgent})\end{aligned}$$

We assume that the values in $\mathsf{AddK}_{\mathsf{Intr}}$ are not used in the model the SSL layer is to be inserted into. In this case, the additional knowledge does not change the values the intruder can send to components connected via other attacked channels in the original model, and the truth values of the security requirements are not influenced. For the model of the bank application, this assumption is fulfilled.

Restricted Intruder

We restricted the intruder's behaviour in the threat scenario of the original model in a similar way as in case of the shared key encryption layer (see Section 7.3.3), by adding two components D1 and D2 that cause delays corresponding to the delays caused by SSLClient and SSLServer.

In particular, D1 can store a message and release it later (to simulate the behaviour of SSLClient, which stores the first message and sends its encryption only after a successful handshake). The only additional functionality of D1 and D2 is to forward messages with one tick delay or to block them.

Besides, the intruder in the threat scenario of the original model can wait for some time performing no interactions, before the $\mathsf{replace}_c$ variable is assigned indicating whether or not the component connected to the client side is replaced by the intruder. This corresponds to the time needed by the SSL layer to perform the first stages of the SSL handshake, where it is still not determined if a connection is established with the legitimate client or with an intruder. In the intruder model, an unconditional self-loop in the Init state was added to the Medium component (see Figure 5.9 on page 104) for this purpose.

Refinement Relation

The strengthened refinement relation $\mathsf{RefineRel}'(\sigma, \sigma', \mathsf{AddK}_{\mathsf{Intr}})$ we used is analogous to the strengthened refinement relation specified for the shared key encryption layer in Section 7.3.3.

We focus on the aspects which are special for the relation between the threat scenario of the SSL layer and the threat scenario of the original model with its replace annotation on the component connected to the client side. In particular, we need to define the correspondence between the $\mathsf{replace}_c$ variable in the intruder in the threat scenario of the original model and the state variables of the SSL layer, and we need to state which messages can be replayed by the intruder and which messages belong to the intruder knowledge in the threat scenario of the original model.

The variable $\mathsf{replace}_c$ is assigned the value True if $\mathsf{SSLServer.store\_kc} \neq \mathsf{KC}$ and SSLServer is not in the control state Init, and False otherwise. If SSLServer is in the control state Init, the intruder in the threat scenario of the original model waits and performs no interaction. The reason for this choice is that in the SSL layer, the commitment if the server establishes a connection with the legitimate client or with the intruder is exactly made when the SSLServer accepts the session key, stores it to store_kc and changes to the GotClientHello state.

The message stores M_p for replays of messages sent from the client to the server are assigned the cleartext x of messages $\mathsf{EncrMsgC}(\mathsf{KC}, x)$, if the intruder knowledge in the threat scenario of the SSL layer contains such messages. The message stores for replays of messages sent from server to client are fixed analogously with the cleartext of messages $\mathsf{EncrMsgS}(\mathsf{KC}, x)$. This reflects the fact that the replay capability of the intruder in the threat scenario of the original model is only relevant if the SSL connection has been established between the legitimate client and the server (i.e., $\mathsf{replace}_c = \mathsf{False}$ and thus the intruder in the threat scenario of the original model cannot insert messages other than replays corresponding to replays of messages encrypted with KC on the SSL layer).

The knowledge of the intruder in the threat scenario of the original model only consists of values of type TMessagKey. The other possible values in the knowledge of the intruder in the threat scenario of the SSL layer belong to $\mathsf{AddK}_{\mathsf{Intr}}$. The variables storing the knowledge of values of type TMessagKey in the intruder in the threat scenario of the original model are assigned to the corresponding values of the variables in the intruder in the threat scenario of the SSL layer. This also affects variables in the intruder in the threat scenario of the SSL layer whose values contain subterms of type TMessagKey that can potentially be extracted (because the necessary keys are known or can be generated).

Proof via SAT Solving

By SAT solving, we showed that the insertion of the SSL layer into a pair of channels annotated with public, secret and auth, and where the component connected to the client side is annotated with replace, is indeed a security pattern.

For a bound $l = 20$, the SAT solver Chaff takes 6210s on the used system to establish that there are no solutions to the corresponding propositional formula. $l = 20$ is sufficiently large to cover all relevant interactions between the components of the SSL layer and the intruder. All counterexamples we obtained because of initial errors in the refinement relation had length $l = 15$ or smaller.

Together with the fact that the stated security requirements were verified in the threat scenario of the original model of the bank application using model checking (see Section 5.4.1), therefore the transformed model of the bank application after the insertion of the SSL layer fulfils the stated security requirements as well.

7.5. Related Work

Application of Security Mechanisms in Top-Down Oriented Development

A top-down oriented approach for the development of security-critical systems by stepwise concretisation is presented in [Eck98]. As an example, the concretisation of a communication channel is described. In the context of models in the formal method Focus, [Lot97] gives security-specific development activities, including (security) mechanism embedment, here to be applied if the verification of global security requirements fails, rather than as a concretisation of abstract assumptions. [Rud01] describes a top-down oriented design method for cryptographic protocols, where abstract representations of protocols are derived from a formal specification of the security service a protocol should provide, using synthesis rules. Cryptographic primitives are first modelled in terms of abstract secure channels. [VWW02] shows how to integrate security-specific development activities into a waterfall process to fulfil the requirements stated by the Common Criteria.

Layered Protocols

[Bro04, HB05] describes a general formal model for layered architectures based on the Focus method. Focus also features several notions of refinement (see [BS01]). The property that the insertion of a protocol layer is a security-preserving transformation is related to the notion of interface refinement, i.e. a refinement relationship between an abstract specification (in our case, the intruder behaviour in the threat scenario of the original model) and a concrete specification together with representation specifications translating between the abstract and concrete levels (in our case, the intruder behaviour in the threat scenario of the transformed model together with the components realising the protocol).

The Cadence SMV model checker [McM99] supports refinement verification by specification of a refinement relationship. Refinement between state variables of the different layers is verified separately for each state variable. Bounded model checking is not supported.

EAP-TTLS [FBW02] is an example for a layered security protocol, proposed in the Internet Engineering Task Force (IETF). Here, a (client) authentication protocol (EAP, extensible authentication protocol) is tunnelled over a previously established TLS channel providing confidentiality and server authentication. [ANN03] describes a man-in-the-middle attack on such protocols, arising if the client authentication protocol can also be used in an untunnelled form. In our approach, we can allow or rule out such attacks by appropriate annotations in the security-enriched model of the protocol components.

Security Patterns

General work on security patterns includes [YB97, FP01, Sch03]. Following the design patterns movement [GHJV95], in the context of that work security patterns are regarded as well-understood solutions to recurring information security problems, described in a structured way. In contrast, we focus on formally defined transformations preserving the security requirements.

[Jür01] addresses secrecy-preserving refinement using a formal model and applies it to a secure channel implemented by a handshake protocol (TLS). [Jür04] contains an example where UMLsec is used for the modelling and analysis of an abstract channel preserving secrecy and of its realisation by encryption. [AFG02] describes how to abstractly model secure channels in the join calculus and gives a security-preserving translation of such models to the sjoin calculus, which includes cryptographic primitives, by mapping communication on abstract secure channels to encrypted communication. The respective proofs are carried out manually.

The approach presented in [HL01] goes in the opposite direction, by giving simplifying transformations for security protocols (e.g. removing encryptions or fields) such that if no attacks are found in the simplified protocol, then there are no attacks in the original version.

7.6. Summary and Discussion

We presented a **model-based approach for the application of security mechanisms** within a model of a security-critical system, **as part of a top-down oriented development process**. Here, the purpose of security mechanisms is to counter threats ruled out in earlier stages by corresponding assumptions. The central part of our approach are **model transformations for the introduction of security mechanisms** into a model, based on the abstract assumptions annotated to the model elements.

We gave a **model transformation for the insertion of layered protocols** into communication channels, and demonstrated it at the example of a shared-key encryption layer realising the abstract assumptions confidentiality and authenticity. The model transformation was implemented as a plugin to the AutoFocus/Quest tool set.

In general, the verification of security requirements must be repeated after the application of a security mechanism. We introduced the concept of **security patterns**, describing transformations of a model for the application of a security mechanism such that security requirements that are fulfilled in the original model are also fulfilled in the transformed model. For this purpose, we formally defined a **condition for security-preserving transformations**, which is related to refinement with special treatment of the intruder models in the original and transformed model. Thus, the security verification can be modularised into the verification of a model including abstract assumptions and the verification if the application of security mechanisms to realise the assumptions is security-preserving, which leads to a reduction of the verification effort.

We explained how to show that the insertion of a particular layered protocol at an appropriate place in a model is a security pattern and exemplarily carried out such a proof for the shared-key encryption layer.

To **facilitate the verification if a given model transformation is security-preserving** and to make it less error prone, we described an **automated approach based on SAT solving**. For this purpose, the designer must state the relationship between the state variables of the abstract and concrete intruder models. Again, we applied the approach to the example of the shared-key encryption layer and in particular showed how to account for delays introduced by the components handling the protocol. The automated verification support has been integrated into the AutoFocus/Quest tool set.

We gave examples for **further security patterns representing common uses of encryption and/or signature** as layered protocols and stated the abstract assumptions they realise. Developing and evaluating a larger collection of security patterns is subject of future work. Note that it may not always be possible or desirable to elaborate appropriate abstract assumptions corresponding to a particular security mechanism such that a security-preserving transformation can be defined. In particular, we required that the security mechanism is independent of the rest of the system, in the sense that the additional protocol messages that can be eavesdropped do not affect the communication between the other components. Besides, there are application-specific security mechanisms, such as the payment protocols described in this work, that provide very specific security services for which it may be difficult to give appropriate abstract characterisations. In addition, even if appropriate abstract characterisations are found, it is questionable if they can be re-used in the development of other systems.

Finally, we demonstrated the practical applicability of the presented concepts at a more complex example, taken from a real-life case study: we describe the **layered verification of the bank application model** introduced in Chapter 4, where a pair of channels in a system with appropriate assumptions is replaced by an SSL connection preserving the security requirements.

8. Conclusion

This chapter concludes the thesis. We summarise and discuss the main results and give an outlook on future work.

8.1. Summary and Discussion

In this thesis, we presented concepts for an application of model-based development techniques to the development of security-critical systems. Thereby we aimed to address the lack of integration of security in current development methodologies, in particular the lack of the support of integrated security testing and application of security mechanisms based on formally defined models. An important concern was the need to improve the accessibility of formally-based specification, verification, and testing techniques for security-critical systems to non-expert developers in the area of security, to lower the threshold for the use of such techniques in an industrial context.

The main results of this work are:

- concepts for an integrated, model-based methodology for the development of security-critical systems, based on design models enriched with security-related information;
- model-based approaches for security verification, security testing and application of security mechanisms;
- the provision of tool support within a general-purpose CASE tool offering easy-to-understand graphical description techniques; and
- the demonstration of the practical applicability of the presented concepts in several case studies.

As an example specification language and tool, we used AutoFocus/Quest, which provides general description techniques to model distributed systems as networks of communicating components specified via finite state machines.

Modelling Security-related information to be integrated into design models includes global security requirements, threats, assumptions, and security mechanisms. We explained how to define and manipulate such security-enriched models at the example of AutoFocus/Quest. Within the extended description techniques, the use of cryptographic operations and data is supported by special constructs added to the underlying functional language. A formally defined semantics ensures that the models precisely specify the required behaviour and can thus be used for verification and testing and as well as a part of the documentation for achieving certification at high assurance levels (level 5 or higher in the Common Criteria for security evaluation). We focused on the specification of communication-related threats, assumptions and mechanisms. Other aspects, such as access control rules, or more detailed information can be integrated in a similar manner, if necessary by additional description techniques. However, we explicitly did not define a completely new specification language for security-related behaviour (as done e.g. in protocol specification languages), to be able to take advantage of existing experience of the developers, of the use of a domain-specific language and of existing tool support.

Verification We explained how security verification can be performed based on threat scenarios that can be automatically generated from security-enriched models. A threat scenario represents the behaviour of a system under attack. It is given by a model in the same specification language as the original security-enriched model, such that available verification support can be used for verifying security requirements and the threat scenario can be easily adapted to analyse customised threats. A disadvantage of this approach is that idiosyncrasies of the specification language must be dealt with when generating a model of the intruder behaviour (in the case of AutoFocus/Quest, e.g. introduced communication delays and the fact that transitions in state machines are deterministic). We addressed this problem by additionally defining a generic intruder model on the semantical level and justifying the soundness and completeness of the intruder model in the generated threat scenario with respect to the generic intruder model.

Testing Even if a verified model is available, testing is necessary to gain confidence in the security of an implementation, in particular if reliable code generation is not supported by the used tool or the generated code is not suitable for deployment. We presented a first approach for integrated model-based security testing. Two main activities for model-based security testing were addressed: the generation of test sequences from a threat scenario with the aim to discover possible vulnerabilities, and the concretisation of test sequences given in terms of the model, where usually an abstract notion of messages and cryptography is employed to make verification feasible. The concretisation is performed by a test driver translating the abstract messages in the test sequences to bit sequences to be passed to and expected from the implementation. For test sequence generation, we presented two testing criteria for the selection of appropriate test sequences. We evaluated our testing approach based on a case study. The results were promising, nevertheless the actual given testing criteria should be primarily regarded as a demonstration of the underlying concepts (manipulation of the security properties respectively of the model for the purpose of security test sequence generation). Extension and adaption of the testing criteria for particular systems is not a problem.

Mechanisms Security-critical systems are often developed in a top-down oriented manner, by first specifying security aspects abstractly and subsequently including appropriate mechanisms. We explained the utilisation of such a development strategy within a model-based development approach for security-critical systems, by formulating the application of a security mechanism as a model-based development activity given by a model transformation. We presented proof concepts to show that a given model transformation preserves the stated security requirements, and provided automation support for such proofs based on SAT solving. Thereby, modular verification is made possible, i.e. splitting the verification task into the verification of the original model containing abstract assumptions and the verification of security mechanisms realising these assumptions. We defined the notion of security patterns as generic descriptions of such security-preserving transformations.

Methodology and Evaluation We put the above described development activities into the general context of a model-based development methodology for the development of security-critical systems.

The presented approaches were successfully applied in a number of case studies from the domain of electronic business and cryptographic protocols. According to feedback from industrial project partners (in particular [HSG$^+$03]), this work provides a solid basis for the more widespread use of model-based and formally-defined methodologies in the development of security-critical systems. We demonstrated that all necessary design, verification, and testing activities could be carried out within a model-based development methodology, could be automated to a large extent, and rigorous completeness and soundness justifications could be applied.

Therefore, we are convinced that by the support of verification, testing and stepwise development within a general formally-based CASE tool the quality of developed security-critical systems is considerably improved.

8.2. Future Work

Currently, follow-up work and further evaluation of the approaches described in this thesis is carried out within the project Mewadis in cooperation with the German car manufacturer BMW. The Mewadis project [Mew04] is concerned with techniques for the analysis, modelling, and validation of reliable, adaptive, context-aware services, and with process models for their development. The results are prototypically implemented in the domain of automotive systems. Because of the high criticality of systems in the automotive domain, security is an important issue. In the following, we give an outlook on possible future work on the topics described in this thesis.

When, during the development of a system, additional classes of security requirements, threats, and assumptions are encountered, it is useful to extend the respective means of specification (property patterns or tag definitions) within the security-enriched models. For instance, for the analysis of availability requirements, a "fair" intruder may be specified who cannot block the communication indefinitely. Of particular interest are access control requirements, which we only marginally considered because of our focus on communication-related aspects. First work on the specification and verification of access control requirements using the tool AutoFocus/Quest is described in [DGJW04], within the context of service-based models considered in the Mewadis project.

It would also be interesting to adapt the presented approaches to other specification languages, such as the UML (and particularly, the UMLsec approach) or the UML-RT, respectively the parts of the UML2.0 derived from the UML-RT (structured classes with ports and connectors). Structured classes in the UML2.0 are conceptually close to AutoFocus/Quest system structure diagrams. The concepts underlying the approaches described in this thesis do not depend on the choice of AutoFocus/Quest, but the given concrete realisations and formalisations would have to be adapted for this purpose.

For security verification, the use of other model checking tools beside SMV or bounded model checking via SAT solving can be evaluated, to decrease the necessary time and memory resources, respectively to enlarge the class of models that can be verified. Towards this end, currently tool connections of AutoFocus/Quest to the model checkers Spin [Hol97], and SAL [dMOR+04] are developed. SAL also provides features for the verification of infinite-state systems based on constraint solving techniques. Besides, the translation of particular AutoFocus/Quest models to the input language of efficient special-purpose protocol model checkers such as Athena [SBP01] may be considered.

The empirical results on the quality of generated security test sequences with respect to their potential to find faults were promising (see Chapter 6). Further case studies must be carried out to substantiate these results and if necessary to refine the developed testing criteria. Besides, it would be interesting to integrate the PROTOS approach [Kak01], mainly targeted on the detection of low-level vulnerabilities such as buffer overflows, into our work.

To support top-down oriented development of security-critical systems, a larger collection of security patterns may be developed. For instance, in [DGJW04], we describe a security pattern for the synchronisation of authentication information to ensure the validity of the stated access control requirements.

Security-enriched models can be used as a basis for code generation. For this purpose, similar concretisations of abstract data must be performed as in case of security testing, i.e. the mapping of symbolical representations of cryptographic operations and data to actual calls to a framework providing cryptography within the used programming language. The code generator must be adapted to the requirements of the platform the system is to be deployed on. For instance, the Java dialect JavaCard, in which applications can be written for the deployment on smart cards containing a Java virtual machine, only supports a subset of the features of the Java programming language, for which a code generator is available as part of the AutoFocus/Quest tool set.

If the generated code cannot be used directly to deliver the required functionality (for example, in the event whereby some aspects were abstracted away in the model), a promising approach would be to employ it as a wrapper for the purpose of intrusion detection. I.e., the generated code is executed together with the manually written code, and the generated code serves to detect violations of the security requirements. If a violation is detected, it can be reported and the current transaction can be aborted if necessary. Another possibility is the automatic generation of security-related deployment information along the lines of [BDL03], for example an access control policy for Enterprise Java Beans (EJB) or a set of permissions in the Java-based service framework OSGi considered in the Mewadis project.

Finally, earlier development phases such as security requirements engineering should be integrated into our methodology, possibly with the help of a recently developed extension of AutoFocus/Quest supporting requirements specification and requirements tracing. [DGP$^+$04] describes a combined approach, where in the early phases UML models are used (in particular, activity diagrams, use case diagrams and sequence diagrams), within the context of service-based development. The use cases are extended with the result of a first security analysis. From these models, AutoFocus/Quest specifications of services are derived for verification, code generation and testing.

Bibliography

[ABM98] P. E. Ammann, P. E. Black, and W. Majurski. Using Model Checking to Generate Tests from Specifications. In *Proc. 2nd International Conference on Formal Engineering Methods (ICFEM)*, 1998.

[ADX01] P. Ammann, W. Ding, and D. Xu. Using a Model Checker to Test Safety Properties. In *Proc. 7th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2001.

[AFG02] M. Abadi, C. Fournet, and G. Gonthier. Secure Implementation of Channel Abstractions. *Information and Computation*, 174(1):37–83, 2002.

[AG99] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.

[AJ01] M. Abadi and J. Jürjens. Formal Eavesdropping and its Computational Interpretation. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, 2001.

[AJSW00] N. Asokan, P. Janson, M. Steiner, and M. Waidner. The State of the Art in Electronic Payment Systems. In M. Zellkowitz, editor, *Advances in Computers*, volume 53. Academic Press, 2000.

[And01] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* Wiley Computer Publishing, 2001.

[ANN03] N. Asokan, V. Niemi, and K. Nyberg. Man-in-the-Middle in Tunneled Authentication Protocols. In *Proc. International Workshop on Security Protocols (SPW)*, 2003.

[APS99] V. Apostolopoulos, V. Peris, and D. Saha. Transport Layer Security: How Much Does It Really cost? In *Proc. Conference on Computer Communications (IEEE INFOCOM)*, 1999.

[AS85] B. Alpern and F. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[AVB$^+$02] A. Armando, V. Vigneron, D. Basin, et al. The AVISS Security Protocol Analysis Tool. In *Proc. 14th International Conference on Computer Aided Verification (CAV)*, 2002.

[BBH$^+$03] R. Breu, K. Burger, M. Hafner, J. Jürjens, G. Popp, G. Wimmel, and V. Lotz. Key Issues of a Formally Based Process Model for Security Engieering. In *Proc. 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA)*, 2003.

[BBHP04] R. Breu, K. Burger, M. Hafner, and G. Popp. Towards a Systematic Development of Secure Systems. In *Proc. 2nd International Workshop on Security in Information Systems (WOSIS)*, 2004.

[BCC$^+$03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. In M. Zellkowitz, editor, *Advances in Computers*, volume 58. Academic Press, 2003.

[BDL03] D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security for Process-Oriented Systems. In *Proc. 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2003.

[Bei90] B. Beizer. *Software Testing Techniques.* International Thomson Computer Press, 2nd edition, 1990.

[BHS99] M. Broy, F. Huber, and B. Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 14(3), 1999. (in German).

[Bis03] M. Bishop. *Computer Security — Art and Science.* Pearson Education, 2003.

[BMV03] D. Basin, S. Mödersheim, and L. Viganò. An On-the-Fly Model Checker for Security Protocol Analysis. In *Proc. 8th European Symposium on Research in Computer Security (ESORICS)*, 2003.

[BOY00] P. E. Black, V. Okun, and Y. Yesha. Mutation Operators for Specifications. In *Proc. 15th International Conference on Automated Software Engineering (ASE)*, 2000.

[Bra03] P. Braun. *Metamodellbasierte Kopplung von Werkzeugen in der Softwareentwicklung.* Dissertation, Technische Universität München, Nov 2003. (in German).

[Bro04] M. Broy. Services, Components, Interfaces and Layered Architectures — Specification, Composition, and Refinement. In *Lecture Notes of Marktoberdorf Summer School*, 2004.

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems.* Springer, 2001.

[CB03] R. Chandramouli and M. Blackburn. Model-based Automated Security Functional Testing. In *Proc. 7th Workshop on Distributed Objects and Components Security (DOCSEC)*, 2003. Presentation slides.

[CCG$^+$02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. 14th International Conference on Computer-Aided Verification (CAV)*, 2002.

[CEP01] CEPSCO. Common Electronic Purse Specifications, 2001. Business Requirements Version 7.0, Functional Requirements Version 6.3, Technical Specification Version 2.3, available from http://www.cepsco.com.

[Com99] Common Criteria for Information Technology Security Evaluation Version 2.1. Technical report, 1999. URL: `http://www.commoncriteria.org/docs/index.html`.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0, Jan 1999. Internet Draft RFC 2246.

[DAC99] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. 21st International Conference on Software Engineering (ICSE)*, 1999.

[DBG01] J. Dushina, M. Benjamin, and D. Geist. Semi-Formal Test Generation with Genevieve. In *Proc. 38th Design Automation Conference (DAC)*, 2001.

[DF93] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *FME '93: Industrial-Strength Formal Methods*, pages 268–284, 1993.

[DGJW04] M. Deubler, J. Grünbauer, J. Jürjens, and G. Wimmel. Sound Development of Secure Service-based Systems. In *Proc. 1st International Conference on Service Oriented Computing (ICSOC)*, 2004.

[DGP$^+$04] M. Deubler, J. Grünbauer, G. Popp, G. Wimmel, and C. Salzmann. Towards a Model-Based and Incremental Development Process for Service-Based Systems. In *Proc. IASTED International Conference on Software Engineering (IASTED SE)*, 2004.

[Die04] R. Dierstein. Sicherheit in der Informatikstechnik–der Begriff IT-Sicherheit. *Informatik Spektrum*, 27(4):343–353, 2004. (in German).

[DM00] W. Du and A. Mathur. Testing for Software Vulnerability Using Environment Perturbation. In *Proc. International Conference on Dependable Systems and Networks (DSN), Workshop on Dependability Despite Malicious Faults*, 2000.

[dMOR$^+$04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *LNCS*, pages 496–500. Springer, 2004.

[DW99] D. F. D'Souza and A. C. Wills. *Components and Frameworks with UML, The Catalysis Approach.* Addison-Wesley, 1999.

[DY83] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[Eck98] C. Eckert. Sichere, verteilte Systeme – Konzepte, Modelle und Systemarchitekturen, 1998. Professorial thesis, Technische Universität München (in German).

[Eck01] C. Eckert. *IT-Sicherheit, Konzepte – Verfahren – Protokolle*. Oldenbourg Wissenschaftsverlag, 2001. (in German).

[Eme90] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[FBW02] P. Funk and S. Blake-Wilson. EAP Tunneled TLS Authentication Protocol (EAP-TTLS), 2002. IETF pppext working group draft draft-ieft-pppext-eap-ttls-01.txt.

[FH97] E. B. Fernandez and J. C. Hawkins. Extending Use Cases and Interaction Diagrams to Develop System Architecture Requirements. Technical Report TR-CSE-97-47, Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Florida, 1997.

[FKK96] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol Version 3.0, 1996. URL: `http://home.netscape.com/eng/ssl3/index.html`.

[FP01] E.B. Fernandez and R.Y. Pan. A Pattern Language for Security Models. In *Proc. 8th Conference on Pattern Languages of Programs (PloP)*, 2001.

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & Sons, 2003.

[GGK$^+$04] R. Grimm, K.E. Großpietsch, H. Keller, I. Münch, K. Rannenberg, and F. Saglietti. Technische Sicherheit und Informationssicherheit — Unterschiede und Gemeinsamkeiten. Terminologie Workshop des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V., `http://www-sec.uni-regensburg.de/begriffeWSMai2004/`, 2004. (in German).

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software*. Addison-Wesley, 1995.

[GHJW03] J. Grünbauer, H. Hollmann, J. Jürjens, and G. Wimmel. Modelling and Verification of Layered Security Protocols: A Bank Application. In *Proc. 22nd International Conference of Computer Safety, Reliability and Security (SAFECOMP)*, 2003.

[GL97] F. Germeau and G. Leduc. Model-based Design and Verification of Security Protocols using LOTOS. In *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[GLLR04] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. 2004 CSI/FBI Computer Crime and Security. Technical report, Computer Security Institute, 2004.

[Grü03] J. Grünbauer. Modellbasierte Sicherheitsanalyse einer Bankapplikation. Diplomarbeit, Technische Universität München, 2003. (in German).

[GSG99] S. Gritzalis, D. Spinellis, and P. Georgiadis. Security Protocols over Open Networks and Distributed Systems: Formal Methods for their Analysis, Design, and Verification. *Computer Communications*, 22(8):695–707, 1999.

[HB05] D. Herzberg and M. Broy. Modelling Layered Distributed Communication Systems. *Formal Aspects of Computing*, 2005. Online First Issue, DOI 10.1007/s00165-004-0051-8.

[Hei03] A. Heider. Implementierung des CEPS Bezahlsystems auf einer Java Card, 2003. Individual Project (*Systementwicklungsprojekt*, in German).

[Her03] P. Herzog. Open Source Security Testing Methodology Manual (OSSTMM) 2.1. Technical report, Institute for Security and Open Methodologies (ISECOM), 2003.

[HL01] M. L. Hui and G. Lowe. Fault-Preserving Simplifying Transformations for Security Protocols. *Journal of Computer Security*, 9(1,2):3–46, 2001.

[HMR$^+$98] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported Specification and Simulation of Distributed Systems. In *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems (PPSE)*, 1998.

[HNS97] S. Helke, T. Neustupny, and T. Santen. Automating Test Case Generation from Z Specifications with Isabelle. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. ZUM '97: The Z Formal Specification Notation*, volume 1212 of *LNCS*. Springer, 1997.

[Hol97] G. J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HSE97] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In *Proc. 4th International Symposium of Formal Methods Europe (FME)*, 1997.

[HSG$^+$03] H. Hollmann, K. Schmidt, J. Grünbauer, J. Jürjens, and G. Wimmel. Untersuchung zur Einbindung von Sicherheitsinfrastrukturen in Applikationsarchitekturen und Kommunikationsprotokollen. Technical report, Technische Universität München, 2003. Internal project report (in German).

[IE92] RTCA Inc. and EUROCAE. DO178B / ED-12B: Software Considerations in Airborne Systems and Equipment Certification, 1992.

[Irv00] C. Irvine. Security: Where Testing Fails. *ITEA Journal*, June 2000.

[ITU96] ITU. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JLW05] J. Jürjens, M. Lehrhuber, and G. Wimmel. Model-Based Design and Analysis of Permission-Based Security. In *Proc. 10th International Conference on Complex Computer Systems (ICECCS)*, 2005. To appear.

[Jon93] M. P. Jones. *Release notes for Gofer 2.28*. February 1993. Included as part of the standard Gofer distribution.

[JPW02] J. Jürjens, G. Popp, and G. Wimmel. Towards Using Security Patterns in Model-based System Development. In *Proc. 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2002. Security Focus Group.

[JPW03] J. Jürjens, G. Popp, and G. Wimmel. Use Case Oriented Development of Security-Critical Systems. *Information Security Bulletin*, 8(2):55–60, March 2003.

[JS04] J. Jürjens and P. Shabalin. Automated Verification of UMLsec Models for Security Requirements. In *Proc. 7th International Conference on the Unified Modeling Language (UML)*, 2004.

[Jür01] J. Jürjens. Secrecy-preserving Refinement. In *Proc. 10th International Symposium of Formal Methods Europe (FME)*, 2001.

[Jür02] J. Jürjens. *Principles of Secure Systems Design*. PhD thesis, Oxford University Computing Laboratory, Trinity Term 2002.

[Jür04] J. Jürjens. *Secure Systems Design with UML*. Springer, 2004.

[JW01a] J. Jürjens and G. Wimmel. Formally Testing Fail-Safety of Electronic Purse Protocols. In *Proc. 16th International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2001.

[JW01b] J. Jürjens and G. Wimmel. Security Modelling for Electronic Commerce: the Common Electronic Purse Specifications. In *Proc. 1st IFIP Conference on E-Commerce, E-Business, and E-Government (I3E)*. Kluwer, 2001.

[JW01c] J. Jürjens and G. Wimmel. Specification-based Testing of Firewalls. In *Proc. Andrei Ershov 4th International Conference "Perspectives of System Informatics" (PSI)*, LNCS. Springer, 2001.

[KAH99] J. Kirby, M. Archer, and C. Heitmeyer. SCR: A Practical Approach to Building a High Assurance COMSEC System. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, 1999.

[Kak01] R. Kaksonen. *A Functional Method for Assessing Protocol Implementation Security.* VTT Publications, 2001.

[Kru00] P. Kruchten. *The Rational Unified Process – an Introduction.* Addison-Wesley Longman, 2000.

[LBD02] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: a UML-based Modeling Language for Model-Driven Security. In *Proc. 5th International Conference on the Unified Modeling Language*, 2002.

[Lot97] V. Lotz. Threat Scenarios as a Means to Formally Develop Secure Systems. Technical Report TUM-I9709, Technische Universität München, 1997.

[Lot00] V. Lotz. Formally Defining Security Properties with Relations on Streams. In Steve Schneider and Peter Ryan, editors, *Electronic Notes in Theoretical Computer Science*, volume 32. Elsevier Science Publishers, 2000.

[Löt03] H. Lötzbeyer. *Modellbasierte Testfallermittlung für eingebettete Systeme in sicherheitskritischen Anwendungen.* Dissertation, Technische Universität München, Jun 2003. (in German).

[Low96] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In Margaria and Steffen, editors, *Proc. 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[Low97] G. Lowe. A Hierarchy of Authentication Specifications. In *Proc. 10th Computer Security Foundations Workshop (CSFW)*, 1997.

[Low98] G. Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1,2):53–84, 1998.

[Low99] G. Lowe. Towards a Completeness Result for Model Checking of Security Protocols. *Journal of Computer Security*, 7(2,3):89–146, 1999.

[LPvB94] G. Luo, A. Petrenko, and G. v. Bochmann. Test Selection based on Communicating Nondeterministic Finite State Machines using a Generalized Wp-Method. *IEEE Transactions on Software Engineering*, 20(2):149–162, 1994.

[LY96] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[McG99] G. McGraw. Software Assurance for Security. *IEEE Computer*, 32(4):103–105, 1999.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

[McM99] K.L. McMillan. Getting Started with SMV: User's Manual. Technical report, Cadence Berkeley Laboratories, Berkeley, CA, 1999.

[Mew04] Mewadis Team. Mewadis Project Website, 2004. URL: `http://www4.informatik.tu-muenchen.de/~mewadis/`.

[MIE$^+$05] Technische Universität München, IABG, EADS, Siemens AG, Technische Universität Kaiserslautern, and 4soft. V-Modell XT, 2005. URL: `http://www.v-modell-xt.de/`.

[Mil05] J. Millen. CAPSL – Common Authentication Protocol Specification Language, 2005. URL: `http://www.csl.sri.com/users/millen/capsl/`.

[MLS78] R.A. De Millo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[MMZ$^+$01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th Design Automation Conference (DAC)*, 2001.

[NRT04] J. Niemela, S. Rautiainen, and K. Tocheva. F-Secure Computer Virus Information Pages: Cabir. `http://www.f-secure.com/v-descs/cabir.shtml`, 2004.

[Nta88] S. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.

[Off95] J. Offutt. Practical Mutation Testing. In *Proc. 12th International Conference on Testing Computer Software*, 1995.

[OL02] D. Oheimb and V. Lotz. Formal Security Analysis with Interacting State Machines. In *Proc. 7th European Symposium on Research in Computer Security (ESORICS)*, 2002.

[OMG03] OMG. OMG Unified Modeling Language Specification Version 1.5, 2003. Available at `http://www.omg.org/uml/`.

[oNSSC03] Committee on National Security Systems (CNSS). National Information Assurance (IA) Glossary (CNSS Instruction 4009), 2003.

[Ope02] OpenSSH Team. OpenSSH Security Advisory (adv.channelalloc). `http://www.openbsd.org/advisories/ssh_channelalloc.txt`, 2002.

[OXL99] J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-Based Tests. In *Proc. 1st IEEE Conference on Engineering of Complex Computer Systems (ICECCS)*, 1999.

[Pal01] PalME Team. PalME Project Website, 2001. URL: `http://www4.informatik.tu-muenchen.de/~palme/`.

[Pau98] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1,2):85–128, 1998.

[Pet00] A. Petrenko. Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In *Proc. of the Summer School MOVEP2000, Modelling and Verification of Parallel Processes*, 2000.

[Pfi98] B. Pfitzmann. Higher Cryptographic Protocols, 1998. Lecture Notes, Universität des Saarlandes.

[PJWB03] G. Popp, J. Jürjens, G. Wimmel, and R. Breu. Security-Critical System Development with Extended Use Cases. In *Proc. 10th Asia-Pacific Software Engineering Conference (APSEC)*, 2003.

[PLP01] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 11th IEEE International Workshop on Rapid System Prototyping (RSP)*, 2001.

[Poh04] H. Pohl. Taxonomie und Modellbildung in der Informationssicherheit. *Datenschutz und Datensicherheit*, 28(11):678–685, 2004. (in German).

[Pop05] G. Popp. *Methode zur Integration von Sicherheitsanforderungen in die Entwicklung zugriffssicherer Systeme.* Dissertation, Technische Universität München, 2005. (in German; to appear).

[PPS$^+$03] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-Based Test Case Generation for Smart Cards. In *Proc. 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, 2003.

[Pre03] A. Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme.* Dissertation, Technische Universität München, Aug 2003. (in German).

[PS97] J. Peleska and M. Siegel. Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal*, 19:53–77, 1997.

[PS99] J. Philipps and O. Slotosch. The Quest for Correct Systems: Model Checking of Diagrams and Datatypes. In *Proc. 6th Asia Pacific Software Engineering Conference (APSEC)*, 1999.

[PSW00] B. Pfitzmann, M. Schunter, and M. Waidner. Cryptographic Security of Reactive Systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 32, 2000.

[RSG$^+$00] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols.* Addison-Wesley, 2000.

[Rud01] C. Rudolph. *A Model for Secure Protocols and its Application to Systematic Design of Cryptographic Protocols.* PhD thesis, Queensland University of Technology, Dec 2001.

[Rud03] M. Rudorfer. IT-Trends im Fahrzeug. In *Workshop Embedded IT-Security in Cars (ESCAR)*, 2003. (in German).

[SBP01] D. Song, S. Berezin, and A. Perrig. Athena, a Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, 9(1,2):47–74, 2001.

[Sch96] B. Schneier. *Applied Cryptography.* Wiley, 1996.

[Sch01] B. Schätz. The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA. Technical Report TUM-I0111, Technische Universität München, 2001.

[Sch03] M. Schumacher. *Security Engineering with Patterns.* Springer, 2003.

[Shm98] V. Shmatikov. Efficient Finite-State Analysis for Large Security Protocols. In *Proc. 11th IEEE Computer Security Foundations Workshop (CSFW)*, 1998.

[SK03] S. Sendall and W. Kozaczynski. Model Transformation – the Heart and Soul of Model-Driven Software Development. *IEEE Software, Special Issue on Model Driven Software Development*, 5(20):42–45, 2003.

[SPHP02] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-Based Development. Technical Report TUM-I0204, Technische Universität München, 2002.

[Sto02] S. Stoller. A Bound on Attacks on Authentication Protocols. In *Proc. 2nd IFIP International Conference on Theoretical Computer Science (TCS)*, 2002.

[Ura92] H. Ural. Formal Methods for Test Sequence Generation. *Computer Communications*, 15(5):311–325, 1992.

[Ver04] Verisoft Project Consortium. Verisoft Project Website, 2004. URL: `http://www.verisoft.de/index_en.html`.

[VM01] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, 2001.

[VWW02] M. Vetterling, G. Wimmel, and A. Wißpeintner. Secure Systems Development Based on the Common Criteria. In *Proc. 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, 2002.

[Wim00] G. Wimmel. Specification Based Determination of Test Sequences in Embedded Systems. Diplomarbeit, Technische Universität München, 2000.

[WJ02] G. Wimmel and J. Jürjens. Specification-Based Test Generation for Security-Critical Systems Using Mutations. In *Proc. 4th International Conference on Formal Engineering Methods (ICFEM)*, 2002.

[WLPS00] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Journal on Software Testing Verification and Reliability*, 10(4):229–248, 2000.

[Woo93] M. Woodward. Errors in Algebraic Specifications and an Experimental Mutation Testing Tool. *Software Engineering Journal*, 8(4):211–224, July 1993.

[WW01] G. Wimmel and A. Wißpeintner. Extended Description Techniques for Security Engineering. In *Trusted Information, the New Decade Challege. Proc. 16th International Conference on Information Security (IFIP/Sec)*, 2001.

[YB97] J. Yoder and J. Barcalow. Architectural Patterns for Enabling Application Security. In *Proc. 4th Conference on Pattern Languages of Programs (PloP)*, 1997.

A. Frequently Used Notation

A.1. Sets

$\emptyset$	empty set
$A_1 \cup A_2$	union of A_1 and A_2
$A_1 \cap A_2$	intersection of A_1 and A_2
$A_1 \setminus A_2$	subtraction of A_2 from A_1
$A_1 \subseteq A_2$	A_1 is a subset of A_2
$x \in A$	x is an element of A
$\lvert A \rvert$	cardinality of A
$\{x_1, x_2, \ldots, x_n\}$	set consisting of $x_1, x_2, \ldots, x_n$
$\{x : P(x)\}$	set of all x such that $P(x)$
$\{f(x) : P(x)\}$	set of all $f(x)$ such that $P(x)$
$\mathcal{P}(A)$	set of all subsets of A
$\mathbb{N}$	set of natural numbers (including 0)
$\mathbb{Z}$	set of integers

A.2. Tuples

$A_1 \times A_2 \times \ldots A_n$	cross product of $A_1, A_2, \ldots, A_n$
$(x_1, x_2, \ldots, x_n)$	tuple consisting of $x_1, x_2, \ldots, x_n$

A.3. Functions

$f : A_1 \to A_2$	function of type $A_1 \to A_2$
$f(x)$	f applied to x
$\mathsf{dom}\, f$	domain of f
$\mathsf{rng}\, f$	range of f
$\{v_1 \mapsto x_1, \ldots, v_n \mapsto x_n\}$	function yielding x_j when applied to v_j
$f\vert_A$	restriction of f to domain A
$\perp$	undefined

A.4. Sequences

A^*	finite sequences over A
A^+	non-empty finite sequences over A
A^∞	infinite sequences over A
A^ω	$A^* \cup A^\infty$
$[x_0, x_1, \ldots, x_n]$	sequence consisting of $x_0, x_1, \ldots, x_n$
$\sigma_1 \circ \sigma_2$	concatenation of sequences σ_1 and σ_2
$\sigma(i)$	ith element of sequence σ ($i \geq 0$)
σ^i	suffix of σ starting with ith element
$\sigma\|_A$	pointwise application of $\|_A$ to elements of the function sequence σ

A.5. Logic

False	inconsistency
True	validity
$Q_1 \wedge Q_2$	Q_1 and Q_2
$Q_1 \vee Q_2$	Q_1 or Q_2
$\neg Q$	Q negated
$Q_1 \Rightarrow Q_2$	Q_1 implies Q_2
$\exists x : Q$	there is an x such that Q
$\forall x : Q$	Q holds for all x
$\exists x \in A : Q$	there is an $x \in A$ such that Q
$\forall x \in A : Q$	Q holds for all $x \in A$

A.6. Modelling Formalism

$e^{\mathcal{M}}$	entity set name e interpreted with respect to model $\mathcal{M}$
$f^{\mathcal{M}}$	function identifier f interpreted with respect to model $\mathcal{M}$
$[\![c]\!]_{\mathcal{M}}$	semantics of component c in model $\mathcal{M}$ (as discrete system)
$V_{\mathcal{M};c}$, $I_{\mathcal{M};c}\, T_{\mathcal{M};c}$	state variables, initial condition, and transition relation of $[\![c]\!]_{\mathcal{M}}$
$[\![pr]\!]$	semantics of property pr (as trace predicate)
$[\![c]\!]^{\mathsf{G}}_{\mathcal{M}}$	semantics of component c in model $\mathcal{M}$ (generic intruder)
$V^{\mathsf{G}}_{\mathcal{M};c}$, $I^{\mathsf{G}}_{\mathcal{M};c}\, T^{\mathsf{G}}_{\mathcal{M};c}$	state variables, initial condition, and transition relation of $[\![c]\!]^{\mathsf{G}}_{\mathcal{M}}$
$[\![pr]\!]^{\mathsf{G}}$	semantics of property pr (generic intruder)
$\Psi_{\mathcal{D}}$	characteristic predicate of computations of discrete system $\mathcal{D}$
$t \trianglelefteq t'$	term t is a subterm of term t'
$[v_1/t_1, \ldots, v_n/t_n]$	simultaneous substitution of variables v_j by terms t_j
ts	application of substitution s to term t
$\mathsf{freeVar}(t)$	set of free variables of term t
$\mathsf{eval}_\beta(t)$	evaluation of term t with respect to variable valuation β

Index