

# CORAL: A Model-Based Approach to Risk-Driven Security Testing

Doctoral Dissertation by  
Gencer Erdogan



Submitted to the Faculty of Mathematics and  
Natural Sciences at the University of Oslo  
in partial fulfillment of the requirements for the degree  
Philosophiae Doctor (Ph.D.) in Computer Science

August 2015

© **Gencer Erdogan, 2016**

*Series of dissertations submitted to the  
Faculty of Mathematics and Natural Sciences, University of Oslo  
No. 1706*

ISSN 1501-7710

All rights reserved. No part of this publication may be  
reproduced or transmitted, in any form or by any means, without permission.

Cover: Hanne Baadsgaard Utigard.  
Print production: John Grieg AS, Bergen.

Produced in co-operation with Akademia Publishing.  
The thesis is produced by Akademia Publishing merely in connection with the  
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright  
holder or the unit which grants the doctorate.

# Abstract

The continuous increase of sophisticated cyber security risks exposed to the public, industry, and government through the web, mobile devices, social media, as well as targeted attacks via state-sponsored cyberespionage, clearly show the need for software security. Security testing is one of the most important practices to assure an acceptable level of security. However, security testers face the problem of determining the tests that are most likely to reveal severe security vulnerabilities. This is important in order to focus security testing on the most risky aspects of a system.

In response to this challenge, the security testing community has proposed an approach to support security testing with security risk assessment (risk-driven security testing). In general, the purpose of risk-driven security testing is to focus the testing on the most severe security risks that the system under test is exposed to. However, current approaches carry out risk assessment at a high-level of abstraction (for example, business level) and then perform the testing accordingly. This is a disadvantage from a testing perspective because it leaves a gap between the risks and the test cases which are defined at a low-level of abstraction (for example, implementation level). This gap makes it difficult to identify exactly where in the system risks occur, and exactly how the risks should be tested. This also indicates that current approaches focus on risk-driven test planning at a high-level of abstraction for test management purposes, and do not necessarily focus on guiding the tester in designing test cases that have the ability to reveal vulnerabilities causing the most severe risks.

This thesis proposes a model-based approach to risk-driven security testing, named CORAL, which is specifically developed to help security testers select and design test cases based on the available risk picture. The CORAL approach consists of seven steps supported by a risk analysis language. The risk analysis language is a modeling language based on UML interactions, and is formalized by an abstract syntax and a schematically defined natural-language semantics.

As part of the development and evaluation process of the CORAL approach we carried out three industrial case studies. In the first two case studies, we investigated how risk assessment may be used to identify security test cases, as well as how security testing may be used to improve security risk analysis results. The experiences we obtained from these two industrial case studies helped us to, among other things, shape the CORAL approach. In the third case study we carried out the CORAL approach in an industrial setting in order to evaluate its applicability. The results indicate that CORAL supports security testers in producing risk models that are valid and directly testable. By directly testable risk models we mean risk models that can be reused and specified as test cases based on the interactions in the risk models. This, in turn, helps testers to select and design test cases according to the most severe security risks posed on the system under test.



# Acknowledgements

First and foremost I would like to express my profound gratitude to my supervisor Ketil Stølen for his guidance and encouragement. His insight, attention to detail, and feedback have been invaluable throughout the course of this work. He has been a constant source of inspiration, motivation and knowledge, and I thank him for the fruitful discussions and everything I have learned from him. I also wish to thank my second supervisor Lillian Røstad for all the valuable recommendations and feedback.

I am grateful to Atle Refsdal and Fredrik Seehusen for their insightful suggestions and ideas for how to improve my work, for the interesting discussions, and for co-authoring most of the papers in this thesis. I have also co-authored papers with Yan Li, Ragnhild Kobro Runde, Jan Øyvind Aagedal, and Jon Hofstad. I wish to thank all of them for their efforts and collaboration.

The work on this thesis has been conducted within the DIAMONDS project (201579/S10) funded by the Research Council of Norway under the VERDIKT program, as well as the RASEN project (316853) funded by the European Commission within the 7th Framework Programme.

Many thanks to the guest scientists in the DIAMONDS project: Alexander Pretschner and Fabio Martinelli. I am also thankful to the members of the DIAMONDS advisory board. Both the guest scientists and the members of the advisory board have on a number of occasions provided useful feedback and advice.

I have during the PhD studies been employed by SINTEF ICT as a research fellow at the Department of Networked Systems and Services. Many thanks to the research director Bjørn Skjellaug and all the colleagues for providing a very pleasant and motivating working environment.

I have also at a number of occasions presented my research to Bjørnar Solhaug, Aida Omerovic, Olav Skjelkvåle Ligaarden, Mass Soldal Lund, Gyrd Brændeland, and Tormod Håvaldsrud. Thanks for the many interesting discussions, for all the useful feedback and advice you have provided, and for being among the many great colleagues who I have been so lucky to collaborate and socialize with. I would also like to thank Aida for reading and commenting on my thesis.

Thanks to the many members of the administrative and academic staff at the University of Oslo for helping me whenever I needed assistance.

I would like to thank my friends for all the support, all the joyful time we spend together, and for reminding me that there is a world outside my office.

Most of all I would like to thank my dear family: I thank my parents Haci Ali Erdogan and Zeynep Köker for all their hard work and sacrifices to give their children better lives and opportunities, and I would like to thank my wonderful sisters Berivan Erdogan and Meral Erdogan, and my brother-in-law Stephen Rodriguez-Elizalde, for all the unconditional love, support, encouragement, patience and at times sacrifice. I would also

like to thank Stephen for reading and commenting on my thesis.

# List of original publications

1. Gencer Erdogan, Yan Li, Ragnhild Kobro Runde, Fredrik Seehusen, Ketil Stølen. **Approaches for the combined use of risk analysis and testing: a systematic literature review.** *International Journal on Software Tools for Technology Transfer*, 16(5):627–642, 2014.
2. Gencer Erdogan, Atle Refsdal, Ketil Stølen. **A systematic method for risk-driven test case design using annotated sequence diagrams.** *In Proceedings of the 1st International Workshop on Risk Assessment and Risk-driven Testing (RISK'13)*, pages 93–108. Springer, 2014.
3. Gencer Erdogan, Atle Refsdal, Ketil Stølen. **Schematic generation of English-prose semantics for a risk analysis language based on UML interactions.** *In Proceedings of the 2nd International Workshop on Risk Assessment and Risk-driven Testing (RISK'14)*, pages 205–310. IEEE Computer Society, 2014.
4. Gencer Erdogan, Ketil Stølen, Jan Øyvind Aagedal. **Evaluation of the CORAL approach for risk-driven security testing based on an industrial case study.** Technical report SINTEF A27097, SINTEF ICT, 2015. Submitted.
5. Gencer Erdogan, Fredrik Seehusen, Ketil Stølen, Jon Hofstad, Jan Øyvind Aagedal. **Assessing the usefulness of testing for validating and correcting security risk models based on two industrial case studies.** *International Journal of Secure Software Engineering*, 6(2):90–112, 2015.

The publications 1 - 5 are available as Chapters 9 - 13 in Part II of this thesis. In the case of publications 2 and 3, we have included the full technical reports which are extended, slightly revised versions of the published papers.





# Contents

<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>v</b>
<b>List of original publications</b>	<b>vii</b>
<b>Contents</b>	<b>ix</b>
<b>List of figures</b>	<b>xiii</b>
<b>I Overview</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Background and motivation . . . . .	3
1.2 Objective . . . . .	5
1.3 Contribution . . . . .	6
1.3.1 The CORAL approach . . . . .	6
1.3.2 Empirical studies . . . . .	6
1.3.3 Systematic literature review . . . . .	8
1.4 Thesis overview . . . . .	8
<b>2 Problem characterization</b>	<b>11</b>
2.1 Conceptual clarification . . . . .	11
2.1.1 Security . . . . .	11
2.1.2 Testing . . . . .	12
2.1.3 Security testing . . . . .	14
2.1.4 Risk assessment . . . . .	14
2.1.5 Security risk assessment . . . . .	16
2.1.6 Modeling . . . . .	17
2.1.7 Model-based testing . . . . .	18
2.1.8 Model-based security testing . . . . .	18
2.1.9 Model-based security risk assessment . . . . .	18
2.1.10 Risk-driven model-based security testing . . . . .	18
2.2 Problem specification . . . . .	19
2.3 Success criteria . . . . .	20

2.3.1	The CORAL risk analysis language . . . . .	20
2.3.2	The CORAL method for risk-driven security testing . . . . .	22
<b>3</b>	<b>Research method</b>	<b>25</b>
3.1	The technology research method . . . . .	26
3.2	Evaluation strategies . . . . .	28
3.3	How we have applied the research method . . . . .	30
<b>4</b>	<b>State of the art</b>	<b>37</b>
4.1	Modeling . . . . .	37
4.2	Model-based testing . . . . .	40
4.3	Model-based security testing . . . . .	42
4.3.1	Approaches with main focus on functional security testing . . . . .	42
4.3.2	Approaches based on fuzzing . . . . .	43
4.3.3	Approaches with main focus on pattern-based security testing . . . . .	44
4.3.4	Approaches with main focus on threat-based security testing . . . . .	45
4.4	Risk-driven testing . . . . .	46
4.4.1	Approaches addressing the combination of risk analysis and testing at a general level . . . . .	47
4.4.2	Approaches with main focus on model-based risk estimation . . . . .	48
4.4.3	Approaches with main focus on test-case generation . . . . .	48
4.4.4	Approaches with main focus on test-case analysis . . . . .	49
4.4.5	Approaches based on automatic source code analysis . . . . .	50
4.4.6	Approaches targeting specific programming paradigms . . . . .	50
4.4.7	Approaches targeting specific applications . . . . .	51
4.4.8	Approaches aiming at measurement in the sense that measurement is the main issue . . . . .	51
4.5	Risk-driven security testing . . . . .	52
<b>5</b>	<b>Summary of contributions</b>	<b>55</b>
5.1	The CORAL approach . . . . .	56
5.2	The CORAL risk analysis language . . . . .	57
5.2.1	Graphical notation . . . . .	57
5.2.2	Abstract syntax . . . . .	63
5.2.3	Natural-language semantics . . . . .	65
5.3	The CORAL method for risk-driven security testing . . . . .	68
5.3.1	Test planning (Step 1) . . . . .	70
5.3.2	Threat scenario risk identification (Step 2) . . . . .	71
5.3.3	Threat scenario risk estimation (Step 3) . . . . .	72
5.3.4	Threat scenario risk evaluation (Step 4) . . . . .	73
5.3.5	Threat scenario test case design (Step 5) . . . . .	74
5.3.6	Test execution (Step 6) . . . . .	74
5.3.7	Test incident reporting (Step 7) . . . . .	75
5.4	Overview of industrial case studies . . . . .	75
5.4.1	Empirical studies on the combinations of security risk analysis and security testing . . . . .	75
5.4.2	Empirical study on the applicability of the CORAL approach . . . . .	77
5.5	Overview of systematic literature review . . . . .	77

<b>6</b>	<b>Overview of research papers</b>	<b>79</b>
6.1	Paper 1: Approaches for the combined use of risk analysis and testing: a systematic literature review . . . . .	79
6.2	Paper 2: A systematic method for risk-driven test case design using annotated sequence diagrams . . . . .	80
6.3	Paper 3: Schematic generation of English-prose semantics for a risk analysis language based on UML interactions . . . . .	80
6.4	Paper 4: Evaluation of the CORAL approach for risk-driven security testing based on an industrial case study . . . . .	81
6.5	Paper 5: Assessing the usefulness of testing for validating and correcting security risk models based on two industrial case studies . . . . .	81
<b>7</b>	<b>Discussion</b>	<b>83</b>
7.1	The CORAL risk analysis language . . . . .	84
7.1.1	Graphical notation . . . . .	84
7.1.2	Natural-language semantics . . . . .	88
7.2	The CORAL method for risk-driven security testing . . . . .	89
7.3	Relating our contributions to the state of the art . . . . .	91
7.3.1	Relating the CORAL approach to other risk-driven testing ap- proaches from a risk assessment perspective . . . . .	91
7.3.2	Relating the CORAL approach to other risk-driven testing ap- proaches from a testing perspective . . . . .	94
7.3.3	Relating our empirical evaluations to evaluations in other risk- driven testing approaches . . . . .	97
7.3.4	Relating our systematic literature review to similar literature re- views . . . . .	98
<b>8</b>	<b>Conclusion</b>	<b>101</b>
8.1	The CORAL approach . . . . .	101
8.2	Empirical studies . . . . .	103
8.3	Systematic literature review . . . . .	104
8.4	Directions for future work . . . . .	104
	<b>Bibliography</b>	<b>107</b>
<b>II</b>	<b>Research Papers</b>	<b>121</b>
<b>9</b>	<b>Paper 1: Approaches for the combined use of risk analysis and testing: a systematic literature review</b>	<b>123</b>
<b>10</b>	<b>Paper 2: A systematic method for risk-driven test case design using annotated sequence diagrams</b>	<b>141</b>
<b>11</b>	<b>Paper 3: Schematic generation of English-prose semantics for a risk analysis language based on UML interactions</b>	<b>179</b>
<b>12</b>	<b>Paper 4: Evaluation of the CORAL approach for risk-driven security testing based on an industrial case study</b>	<b>199</b>

**13 Paper 5: Assessing the usefulness of testing for validating and correcting security risk models based on two industrial case studies 221**

# List of Figures

2.1	Conceptual model for security. . . . .	12
2.2	Conceptual model for testing. . . . .	13
2.3	Conceptual model for security testing. . . . .	14
2.4	The overall risk management process (adapted from ISO 31000 [70] and modified). . . . .	15
2.5	Conceptual model for risk assessment. . . . .	15
2.6	Conceptual model for security risk assessment. . . . .	16
2.7	Conceptual model for model. . . . .	17
3.1	The main steps in the technology research method (adapted from Solheim and Stølen [140]). . . . .	27
3.2	Evaluation strategies (adapted from McGrath [97]). . . . .	29
3.3	Research process for the development of the artifacts. . . . .	33
5.1	The overall relationship between the CORAL risk analysis language and the CORAL method for risk-driven security testing. . . . .	56
5.2	Graphical notation for the diagram frame. . . . .	57
5.3	Graphical notation for lifelines. . . . .	58
5.4	Graphical notation for messages. . . . .	60
5.5	Graphical notation for risk-measure annotations. . . . .	61
5.6	Graphical notation for interaction operators. . . . .	62
5.7	The CORAL method for risk-driven security testing. . . . .	69
5.8	Black-box model of feature Exercise Options. . . . .	70
5.9	Risk evaluation matrix constructed with respect to the frequency scale and consequence scale. . . . .	72
5.10	Threat scenario: Malicious user gains access to another system feature by changing parameter <i>exerciseMethod</i> . . . . .	73
5.11	Threat scenario annotated with risk-measure annotations. . . . .	74
5.12	Risk <i>R1</i> is mapped on the risk matrix with respect to likelihood <i>Likely</i> and consequence <i>Moderate</i> . . . . .	75
5.13	Security test case based on the threat scenario in Figure 5.10. . . . .	76



# Part I

## Overview





# Introduction

In this chapter we present the background and motivation for our work, describe our objective, present the main contributions, and give an overview of the thesis.

## 1.1 Background and motivation

Software security is the ability of software to resist events that threaten its dependability with the preservation of integrity, confidentiality, and availability of information [71, 90, 99]. The continuous increase of sophisticated cyber security risks exposed to the public, industry, and government through the web, mobile devices, social media, as well as targeted attacks via state-sponsored cyberespionage, undoubtedly show the importance and need for software security [30, 119, 148]. Software security is achieved by conducting various software security practices, which are the result of systematic studies for creating secure software [98], within the software development life cycle. Some of the major contributions in this respect are: the Security Touchpoints for a Software Development Lifecycle by McGraw [99], the Security Development Lifecycle by Microsoft [65, 103], and the Secure Software Development Lifecycle by Wysopal et al. [163]. Examples of security practices are security requirements engineering, architectural security risk analysis, and security testing. The state of software security practices in the software industry shows that security testing is one of the most important practices within a development life cycle in order to achieve secure software [100].

Security testing is a type of testing conducted to evaluate the degree to which a test item, and associated data and information, are protected so that unauthorized persons or systems cannot use, read, or modify them, and authorized persons or systems are not denied access to them [73]. A test item is a work product that is an object of testing, for example a system or a design specification. The field of software testing has over the last two decades increasingly adopted a model-based approach to testing. According to Utting et al. [153], model-based testing is a variant of testing that relies on explicit behavior models that encode the intended behaviors of a system under test and/or the behavior of its environment. This is motivated by the fact that, traditionally, the process of deriving tests tends to be unstructured, not reproducible, undocumented, unmotivated in terms of lack of detailed rationale for the test design, and dependent on the creativity of individual engineers [118, 130, 153]. Because of similar reasons, security testing also adopts a model-based approach to testing. Model-based security testing is a relatively new field and is in particular focused on the systematic and efficient

specification and documentation of security test objectives, security test cases and test suites, as well as to their automated or semi-automated generation [49, 131].

Due to the complexity of systems and software it is impossible to exhaustively test every single aspect of any given system under test [73]. We cannot test all possible inputs, combinations of inputs, execution paths, or all potential failures caused by, for example, user interface design errors or incomplete requirements analyses [80]. In addition, when testing security-critical software, we are faced with the problem of determining the tests that have the ability to reveal vulnerabilities causing the most severe security risks. Moreover, security testing is limited by strict budget and time constraints. In order to address these problems, there has been suggested a number of testing approaches in which the testing process is supported by risk assessment [40]. These problems do not only apply to security testing, but also to testing in general [51]. In fact, it seems like the testing community has arrived at a common understanding that testing in general should be supported by risk assessment in order to tackle the aforementioned problems. This is reflected by the recent software testing standard ISO/IEC/IEEE 29119 [73, 74] which provides a general testing process that is supported by risk assessment. In general, the purpose of supporting a testing process with risk assessment is to focus the testing on the most severe risks that the system under test is exposed to. The purpose of supporting security testing with security risk assessment, however, is to focus the testing process on the most severe *security* risks that the system under test is exposed to. We refer to security testing approaches that are supported by security risk assessment as risk-driven security testing approaches. Although the term “risk-based” is commonly used in this context, we use the term “risk-driven” to correctly reflect the fact that risks are used as the main guiding factor to steer all phases of the testing process [48].

Security risk assessment is typically carried out in three consecutive steps that involve identifying, estimating, and evaluating security risks [72]. The purpose of risk identification, in the context of risk-driven security testing, is to identify possible unwanted incidents that the system under test and/or its environment may be exposed to, and that may harm security assets. An example of a security asset is the confidentiality of certain information. The purpose of estimating risks is to estimate how often unwanted incidents may occur, as well as to estimate the consequence of unwanted incidents given that they occur. Finally, the purpose of evaluating risks is to prioritize the unwanted incidents with respect to their estimated likelihood of occurrence and consequence, and select the most severe security risks to test based on their prioritization. Having carried out risk assessment, we may focus the testing on the security risks selected for testing. This involves test planning, test case design, test case implementation, test case execution, and test result evaluation/reporting, with respect to the security risks selected for testing.

Although a number of risk-driven testing approaches have been suggested, in which only a handful specifically address security, the field is still immature. Based on our systematic literature review [40], we discovered that there is clearly a need for further research within the field of risk-driven testing in general. In particular, the field needs more formality and preciseness as well as dedicated tool support, and there is very little empirical evidence regarding the usefulness of the various approaches. In addition, there is a lack of common ground. It seems that many of the approaches have been developed in isolation, and with little impact on the field so far. Moreover, current approaches carry out risk assessment at a high-level of abstraction (for example, busi-

ness level) and then perform the testing accordingly. This is a disadvantage from a testing perspective because it leaves a gap between the risks and the test cases which are defined at a low-level of abstraction (for example, implementation level). This gap makes it difficult to identify exactly where in the system risks occur, and exactly how the risks should be tested. This also indicates that current approaches focus on risk-driven test planning at a high-level of abstraction for test management purposes, and do not necessarily focus on guiding the tester in designing test cases that have the ability to reveal vulnerabilities causing the most severe risks.

Security testing goes beyond simple black-box probing on the application/ presentation layer and functional security testing, such as testing whether input validation mechanisms or cryptographic protocol mechanisms are correctly implemented [117]. Thompson [151] argues that to truly test for security, we must test like detectives following clues to insecure behavior and then zeroing in on vulnerabilities, and highlights the need for proper methods and techniques. This is also pointed out by Potter and McGraw [117] who underline that to properly test for software security, security testers must use a risk-driven approach to security testing, because by identifying risks in the system and creating tests driven by those risks, a software security tester can properly focus on aspects of the system in which an attack is likely to succeed.

This thesis addresses these problems, and focuses specifically on the domain of *risk-driven security testing* from a *model-based* perspective.

## 1.2 Objective

The main objective of this thesis is to provide a model-based approach to risk-driven security testing that is specialized for security testers. The approach should systematically guide security testers in a security testing process in which security risk assessment is used to select and design test cases. This means that the approach should provide a modeling language that uses constructs comprehensible to security testers. That is, the modeling language should be capable of expressing risk-related information for security risk assessment purposes, as well as expressing the behavior of a system and its environment for security testing purposes, in an easily understandable manner. In addition, since the risk assessment is used as a basis for test selection and test design, the approach should help security testers in producing risk models (as a result of risk assessment) that are valid and directly testable. Finally, the modeling language should be formally defined in order to support the development of tools and methods.

In summary, our objective is to develop a model-based approach to risk-driven security testing, that is:

1. comprehensible to security testers,
2. useful for the purpose of selecting and designing test cases with respect to the risk assessment results,
3. effective in the sense that it produces risk models that are valid and directly testable, and
4. sufficiently rigorous to support the development of tools and methods.

## 1.3 Contribution

This thesis provides three kinds of contributions. First, it provides a new artifact in terms of a risk analysis language and a method for risk-driven security testing. The risk analysis language and the method for risk-driven security testing are tightly integrated, and we refer to them collectively as the CORAL approach. Second, it provides empirical studies in terms of industrial case studies. In the case studies, we investigate how security risk assessment may be used to support security testing (and vice versa), and the applicability of the CORAL approach. Third, it provides an overview of state of the art approaches that combine risk analysis and testing, in terms of a systematic literature review.

### 1.3.1 The CORAL approach

The CORAL approach consists of a security risk analysis language, and a stepwise method in which the risk analysis language is applied. The term method, in this context, should be understood as “a means or manner of procedure, especially a regular and systematic way of accomplishing something” [35]. The risk analysis language is based on UML interactions [110] and is formalized by an abstract syntax and a schematically defined natural-language semantics. The method for risk-driven security testing consists of seven steps that are supported by the risk analysis language. The seven steps of the method are test planning, threat scenario risk identification, threat scenario risk estimation, threat scenario risk evaluation, threat scenario test case design, test execution, and test incident reporting. The risk analysis language and the stepwise method are tightly integrated and they are specifically developed to help security testers systematically carry out risk-driven security testing. Moreover, the CORAL approach bridges the gap between high-level risks and low-level test cases by systematically guiding security testers in identifying exactly where in the system risks may occur, and exactly how to test the identified risks.

The graphical notation of the risk analysis language (in other words, the concrete syntax) provides the necessary constructs for modeling the system under test and its environment, as well as identifying, estimating, and evaluating security risks that the system under test is exposed to. The graphical notation is also used for designing security test cases. Moreover, the models representing test cases are used for test execution, as well as for reporting test results.

The abstract syntax provides a set of rules, in terms of a context-free grammar, that defines the correct combinations of the constructs in the CORAL risk analysis language. The syntax is useful for modeling interactions that are syntactically correct in the CORAL language.

The natural-language semantics provides a set of rules for schematically translating threat scenarios modeled using the CORAL language into English prose. Testers may use the natural-language semantics to clearly and consistently document, communicate and analyze security risks.

### 1.3.2 Empirical studies

In the course of the work leading up to this thesis, we carried out three industrial case studies. In the first two industrial case studies, we investigated how security testing

may be used as a means to improve the security risk analysis results, as well as how the risk analysis results may be used as a starting point to identify security test cases. The experiences we obtained from these two industrial case studies helped us to, among other things, shape the CORAL approach. In the third case study we tried out the CORAL approach in an industrial setting in order to evaluate its applicability. In particular, we investigated to what extent the CORAL approach helps security testers in designing valid risk models and directly testable threat scenarios, as well as to what extent the CORAL approach is useful for black-box testing and white-box testing. As mentioned in Section 1.3.1, the CORAL language is based on UML interactions. The threat scenarios modeled using the CORAL language are therefore represented as interactions, and by a directly testable threat scenario we mean a threat scenario that can be reused and specified as a test case based on its interactions. Throughout this thesis, we sometimes use the terms threat scenario and risk model interchangeably when the distinction is not important.

The first case study was carried out between March 2011 and July 2011, the second case study was carried out between June 2012 and January 2013, and the third case study was carried out between October 2014 and December 2014. In the first case study, we analyzed a multilingual web application which is designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans. In the second case study, we analyzed a mobile application designed to provide various online financial services to the users on their mobile devices. In the third case study, we analyzed the same web application which we had previously analyzed in the first case study. However, this time we applied the CORAL approach and analyzed different features of the web application.

From the first two case studies we found out that threat scenarios are a good starting point for identifying security test cases. However, in the approach carried out in the first two case studies we were only able to identify high-level test procedures, which we in turn had to refine into test cases in an ad hoc manner. This indicated a need for formality and preciseness in the process of designing test cases. We also found out that the test results are useful for correcting risk models in terms of adding or deleting vulnerabilities, as well as editing likelihood values. Moreover, the test results also proved to be useful for validating risk models in terms of discovering the presence or absence of presumed vulnerabilities, and thereby increasing the trust in the risk models.

In the third case study we found out that, by making use of the CORAL approach, testers are able to identify both valid risk models, and directly testable threat scenarios. The case study results show that the risk models produced in the CORAL approach were valid and of high quality. This is backed up based on two observations. First, we identified the majority of vulnerabilities (11 out of 13 vulnerabilities) by testing the risks considered as most severe. Only two vulnerabilities were identified by testing the risks considered as low risks. Second, we managed to identify all relevant security risks, compared to previous penetration tests, by using the CORAL approach. In addition, we identified five new security risks not detected by the previous penetration tests. We also made direct use of all threat scenarios identified in the case study as security test cases, and we were able to conduct the complete CORAL approach for black-box testing and white-box testing of the system under test.

### 1.3.3 Systematic literature review

The systematic literature review brings forth a state of the art of the literature on approaches for the combined use of risk analysis and testing, based on publications related to this topic. The systematic literature review was carried out according to the following six steps: (1) define the objective of the study, (2) define research questions, (3) define the search process including inclusion and exclusion criteria, (4) perform the search process, (5) extract data from relevant full texts, and (6) analyze data and provide answers for the research questions. This process is constructed based on the guidelines given by Kitchenham and Charters [83].

For all the approaches identified in the systematic literature review we describe their main goal and the strategies used to achieve that goal, the contexts in which they are considered to be particularly useful, their level of maturity with respect to degree of formalization, empirical evaluation, tool support, and finally the relationships between the approaches with respect to citations between the publications.

The survey highlights the need for more structure and rigor in the definition and presentation of the approaches. Evaluations are missing in most cases. The survey may serve as a basis for examining approaches for the combined use of risk analysis and testing, or as a resource for identifying the adequate approach to use.

## 1.4 Thesis overview

The Faculty of Mathematics and Natural Sciences at the University of Oslo recommends that a dissertation is presented either as a monograph, or as a collection of research papers. We have chosen the latter.

This dissertation is based on a collection of five research papers and structured into two main parts. Part I is the introductory part and provides the context and an overall view of the work. Part II contains the collection of research papers. The purpose of the introductory part is to explain the overall contributions presented in the research papers, and to explain how the contributions fit together. The introductory part is organized into the following eight chapters (including this one).

**Chapter 1 – Introduction** provides background and motivation for this thesis, describes the objective, gives an overview of the main contributions, and provides an overview of the thesis.

**Chapter 2 – Problem characterization** provides a conceptual framework for the central concepts used in this thesis, specifies the problem addressed, and then it refines the overall objective presented in Chapter 1 into a set of success criteria that the artifacts must fulfill.

**Chapter 3 – Research method** presents a method for technology research and explains how we used this method throughout the course of the work leading up to this thesis.

**Chapter 4 – State of the art** presents the state of the art of relevance for this thesis.

**Chapter 5 – Summary of contribution** presents the main contributions of this thesis, that is, the CORAL approach, the empirical studies, and the systematic literature review.

**Chapter 6 – Overview of research papers** provides an overview and a summary of the publications produced in the course of the work leading up to this thesis.

**Chapter 7 – Discussion** provides a discussion with respect to the success criteria defined in Chapter 2 and discusses to what extent our artifacts fulfill the success criteria. Moreover, we also discuss how our contributions are related to, and extend, the state of the art.

**Chapter 8 – Conclusion** concludes the thesis by summarizing the work and what has been achieved, as well as discussing different directions for future work.

The research papers in Part II are self-contained and can be read independently of each other. Because the papers are self-contained, they do overlap to some extent with respect to methodological and terminological explanations. We recommend that the papers are read in the order they appear in the thesis. However, readers that are only interested in the CORAL approach can read Papers 2 and 3. Readers that are only interested in the empirical evaluations, that is, the industrial case studies, can read Papers 4 and 5. Readers that are only interested in the systematic literature review can read Paper 1.





## Problem characterization

In Chapter 1 we presented the overall motivation and objective for our research. In this chapter we refine this objective into a set of success criteria. In Section 2.1 we clarify a set of core concepts that play an essential role throughout the thesis. In Section 2.2 we specify the problem addressed in this thesis. In Section 2.3 we present the success criteria that should be fulfilled for a successful accomplishment of our objective.

### 2.1 Conceptual clarification

This section clarifies the main terminology used throughout the thesis. Concepts related to security, testing, risk assessment, and modeling are presented and their meaning in the context of this work is clarified.

#### 2.1.1 Security

In this thesis, we use the term security in the meaning of information security. According to ISO/IEC 27000 [71], “information is an asset that, like other important business assets, is essential to an organization’s business and consequently needs to be suitably protected. Information can be stored in many forms, including: digital form (for example, data files stored on electronic or optical media), material form (for example, on paper), as well as unrepresented information in the form of knowledge of the employees. Information may be transmitted by various means including: courier, electronic or verbal communication. Whatever form information takes, or the means by which the information is transmitted, it always needs appropriate protection.” As shown in Figure 2.1, information security may be specialized into three main dimensions: integrity, confidentiality, and availability.

- **Security** refers to the preservation of confidentiality, integrity and availability of information [71, p. 3].
- **Integrity** is the property of protecting the accuracy and completeness of information [71, p. 4].
- **Confidentiality** is the property that information is not made available or disclosed to unauthorized individuals, entities, or processes [71, p. 2].

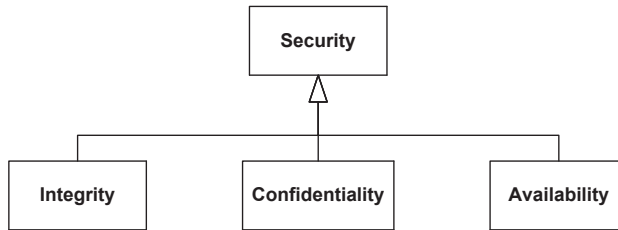


Figure 2.1: Conceptual model for security.

- **Availability** is the property of information being accessible and usable upon demand by an authorized entity [71, p. 2].

## 2.1.2 Testing

In this section, we present the concepts related to software testing that are relevant for this thesis. Our primary source for the notion of software testing and related notions is the international software testing standard ISO/IEC/IEEE 29119 [73, 74]. However, for relevant notions not found in ISO/IEC/IEEE 29119 [73, 74], we use IEEE 829 [67] and BS 7925-1 [149].

Testing is the process of executing a software system with the intent of finding errors [108]. ISO/IEC/IEEE 29119 [74] groups the testing activities that may be performed during the life cycle of a software system into three process groups: organizational test process, test management process, and dynamic test process. The purpose of the organizational test process is to develop, monitor conformance and maintain organizational test specifications, such as the organizational test policy and organizational test strategy [74].

According to ISO/IEC/IEEE 29119 [74], there are three test management processes: test planning, test monitoring and control, and test completion. These generic test management processes may be applied at the project level (project test management), for test management at different test phases (for example, system test management, acceptance test management) and for managing various test types (for example, performance test management, usability test management). The reader is referred to ISO/IEC/IEEE 29119 [74] for further details with respect to these three test management processes.

According to ISO/IEC/IEEE 29119 [74], there are four dynamic test processes: test design & implementation, test environment set-up & maintenance, test execution, and test incident reporting. The dynamic test processes are used to carry out dynamic testing within a particular phase of testing (e.g. unit, integration, system and acceptance) or type of testing (e.g. performance testing, security testing, usability testing). Throughout this thesis, when we refer to a test process in general, we refer to the aforementioned dynamic test process. The conceptual model for testing used in this thesis is defined in Figure 2.2, and includes the following concepts.

- **Test policy** is an executive-level document that describes the purpose, goals, and overall scope of the testing within an organization, and which expresses why testing is performed and what it is expected to achieve [73, p. 4].

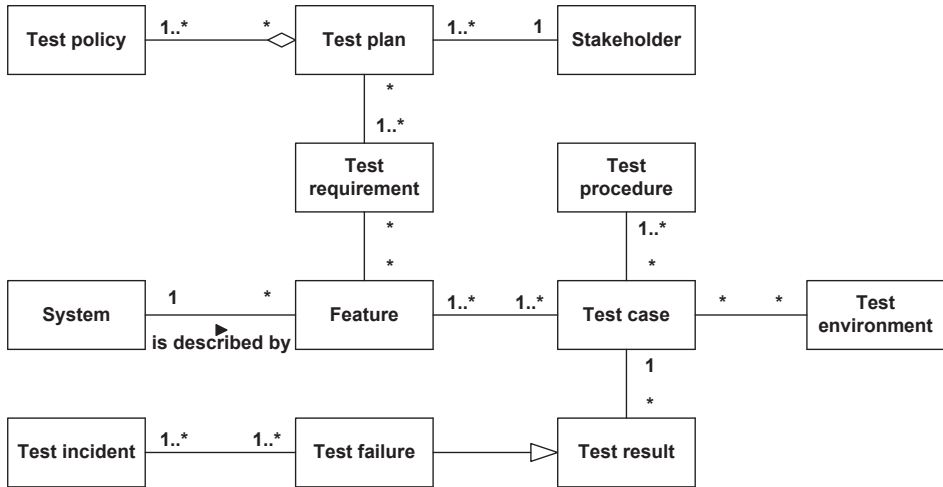


Figure 2.2: Conceptual model for testing.

- **Test plan** is a detailed description of test objectives to be achieved and the means and schedule for achieving them, organized to coordinate testing activities for some test item or set of test items [73, p. 9].
- **Test requirement** is a capability that must be met or possessed by the system (requirements may be functional or non-functional) [149].
- **System** is an interacting combination of elements that aims to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements [16, p. 2-3].
- **Feature** is a distinguishing characteristic of a system (includes both functional and non-functional attributes such as performance and re-usability) [67, p. 9].
- **Test case** is a set of preconditions, inputs (including actions, where applicable), and expected results, developed to drive the execution of a test item to meet test objectives, including correct implementation, error identification, checking quality, and other valued information [73, p. 7]. The terms *test case* and *test* are sometimes used interchangeably.
- **Test procedure** is a sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap up activities post execution [73, p. 10].
- **Test environment** refers to facilities, hardware, software, firmware, procedures, and documentation intended for or used to perform testing of software [73, p. 8].
- **Test incident** is an unplanned event occurring during testing that has a bearing on the success of the test. Most commonly raised when a test result fails to meet expectations [149].

- **Test failure** is the deviation of the software from its expected delivery or service [149].
- **Test result** is an indication of whether or not a specific test case has passed or failed, that is, if the actual result observed as test item output corresponds to the expected result or if deviations were observed [73, p. 10].
- **Stakeholder** is a person or organization that can affect, be affected by, or perceive themselves to be affected by a decision or activity [70, p. 4].

### 2.1.3 Security testing

Security testing is a type of testing conducted to evaluate the degree to which a test item, and associated data and information, are protected so that unauthorized persons or systems cannot use, read, or modify them, and authorized persons or systems are not denied access to them [73]. In other words, the purpose of security testing is to determine whether a system meets its specified security requirements [49]. The basic security concepts that need to be covered by security testing are integrity, confidentiality, and availability. The conceptual model for security testing is shown in Figure 2.3. For the definition of test requirement see Section 2.1.2.

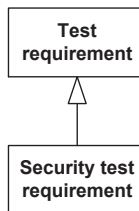


Figure 2.3: Conceptual model for security testing.

- **Security test requirement** is a test requirement specialized towards security.

### 2.1.4 Risk assessment

It is necessary to give an overview of the risk management process before we look closer into risk assessment and other relevant concepts. ISO 31000 Risk management - Principles and guidelines [70] is our main building block in terms of defining and explaining the concept of risk management and the concepts related to risk management. The overall risk management process shown in Figure 2.4 is taken from [70, p. 14]. There is however one deviation: our definition of risk estimation is equivalent to what ISO 31000 refers to as risk analysis. Instead, we use the term risk analysis in line with how the term is used in practice to denote the five step process in the middle of Figure 2.4 starting with establishing the context and ending with risk treatment.

The most relevant part of the risk management process within the context of this thesis is risk assessment. That is, the steps related to risk identification, risk estimation, and risk evaluation. We will therefore explain these steps in the following. The reader is referred to ISO 31000 [70] for a detailed explanation of the complete risk management process depicted in Figure 2.4.

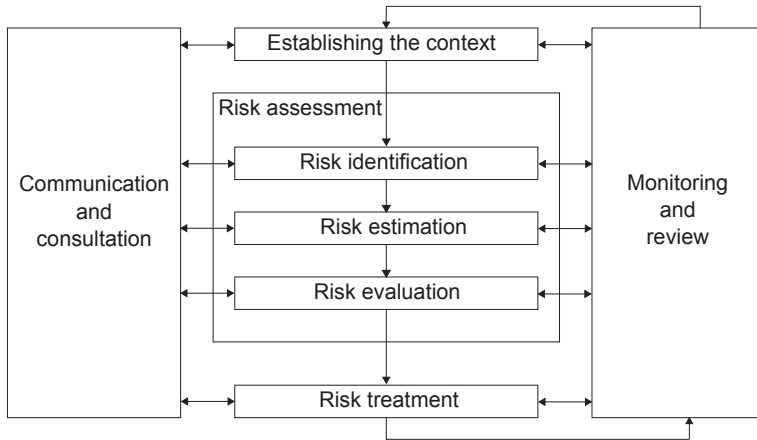


Figure 2.4: The overall risk management process (adapted from ISO 31000 [70] and modified).

Risk identification is the process of finding, recognizing and describing risks. This involves identifying sources of risk, areas of impact, events (including changes in circumstances), their causes and their potential consequences. Risk identification can involve historical data, theoretical analysis, informed and expert opinions, and stakeholder's needs [70, p. 4]. Risk estimation is the process of comprehending the nature of risk and determining the level of risk. Risk estimation provides the basis for risk evaluation and decisions on whether risks need to be treated, and on the most appropriate risk treatment strategies and methods [70, p. 5]. Risk evaluation is the process of comparing the results of risk estimation with risk criteria to determine whether the risk and/or its magnitude is acceptable or tolerable. Risk evaluation assists in the decision about risk treatment [70, p. 6]. The conceptual model for risk assessment is shown in Figure 2.5. For the definition of stakeholder see Section 2.1.2.

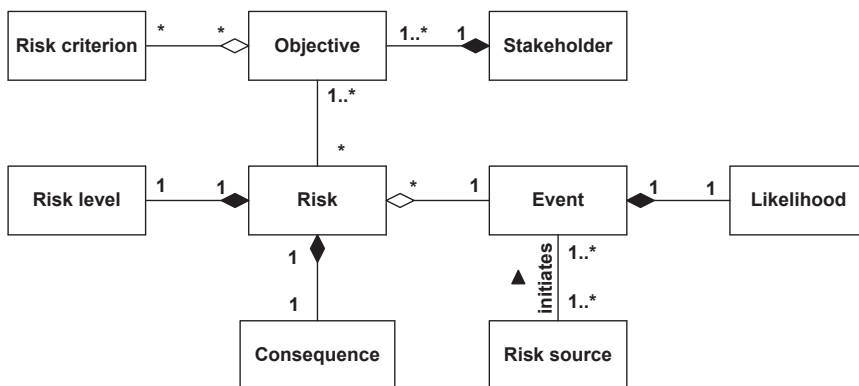


Figure 2.5: Conceptual model for risk assessment.

- **Risk** is the combination of the consequences of an event with respect to an objective and the associated likelihood of occurrence [70, p. 1].

- **Objective** is something the stakeholder is aiming towards or a strategic position the stakeholder is working to attain [36].
- **Risk source** is an element which alone or in combination has the intrinsic potential to give rise to risk [70, p. 4].
- **Event** is the occurrence or change of a particular set of circumstances [70, p. 4].
- **Likelihood** is the chance of something happening [70, p. 5].
- **Consequence** is the outcome of an event affecting objectives [70, p. 5].
- **Risk criterion** is the term of reference against which the significance of a risk is evaluated [70, p. 5].
- **Risk level** is the magnitude of a risk or combination of risks, expressed in terms of the combination of consequences and their likelihood [70, p. 6].

### 2.1.5 Security risk assessment

Security risk assessment is the process of risk assessment specialized towards security. Lund et al. [91] classify risk assessment approaches into two main categories.

- Offensive approaches: risk assessment concerned with balancing potential gain against risk of investment loss. This kind of analysis is more relevant within finance and political strategy making.
- Defensive approaches: risk assessment concerned with protecting what is already there.

In the context of security, the defensive approach is the one that is relevant. The conceptual model for security risk assessment is shown in Figure 2.6. For the definitions of risk, objective, risk source, and event see Section 2.1.4.

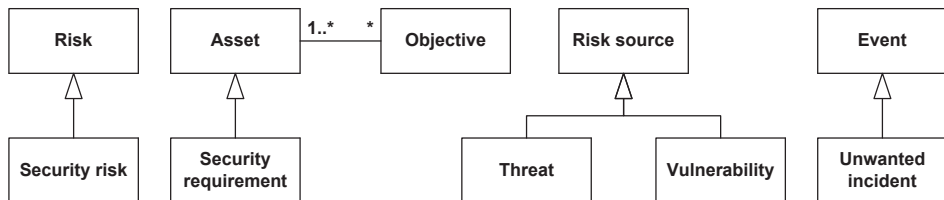


Figure 2.6: Conceptual model for security risk assessment.

- **Asset** is anything that has value to the stakeholders [71].
- **Security requirement** is a specification of the required security for the system [149].
- **Security risk** is a risk caused by a threat exploiting a vulnerability and thereby violating a security requirement.

- **Unwanted incident** is an event representing a security risk.
- **Threat** is potential cause of an unwanted incident [71].
- **Vulnerability** is weakness of an asset or control that can be exploited by a threat [71].

### 2.1.6 Modeling

This section clarifies the notion of model used in this thesis. The conceptual model for *model* is shown in Figure 2.7. The concepts described here are based on information provided by Larkin and Simon [87], Chao [25], TOGAF (The Open Group Architecture Framework) [114], and SWEBOK (Software Engineering Body of Knowledge) [16].

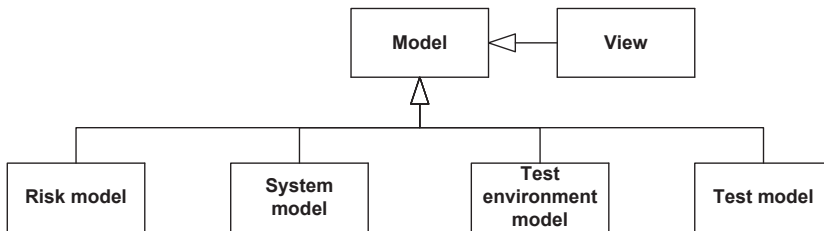


Figure 2.7: Conceptual model for model.

A model may be represented either in a sentential manner, that is, textual representation, or in a graphical/diagrammatic manner.

1. According to Larkin and Simon [87] “in a sentential representation, the expressions form a sequence corresponding, on a one-to-one basis, to the sentences in a natural-language description of the problem. Each expression is a direct translation into a simple formal language of the corresponding natural-language sentence.”
2. According to Larkin and Simon [87] “in a diagrammatic representation, the expressions correspond, on a one-to-one basis, to the components of a diagram describing the problem. Each expression contains the information that is stored at one particular locus in the diagram, including information about relations with the adjacent loci.”

In this thesis, we use the term model in the meaning of graphical/diagrammatic model.

- **Model** is a representation in which information is indexed by twodimensional location [87]. A model provides a smaller scale, simplified, and/or abstract representation of the subject matter.
- **View** represents a related set of concerns.
- **System model** represents a system.

- **Risk model** represents risks.
- **Test model** represents tests/test cases.
- **Test environment model** represents the test environment.

### 2.1.7 Model-based testing

Model-based testing is a software testing approach that relies on behavioral models of a system under test and its environment to derive test cases. Usually, the test model is derived in whole or in part from a model that describes functional or non-functional aspects of the system under test [153].

- **Model-based testing** is testing that involves the construction and analysis of system models, test environment models and test models to derive test cases.

### 2.1.8 Model-based security testing

Model-based security testing is in particular focused on the systematic and efficient specification and documentation of security test objectives, security test cases and test suites, as well as to their automated or semi-automated generation [49, 131].

- **Model-based security testing** is security testing that involves building the behavioral model, defining test selection or generation criteria and transforming them into operational test case specifications [49].

### 2.1.9 Model-based security risk assessment

Based on the notions of model and security risk assessment we define the term model-based security risk assessment as follows.

- **Model-based security risk assessment** is security risk assessment in which each step of the process includes the construction and analysis of models.

### 2.1.10 Risk-driven model-based security testing

Based on the notions of model-based security testing and model-based security risk assessment we define the term risk-driven model-based security testing as follows.

- **Risk-driven model-based security testing** is model-based security testing that makes use of model-based security risk assessment within the security testing process to support risk-driven test planning, risk-driven test design and implementation, and risk-driven test reporting.

In this thesis, we distinguish between two strategies for the combined use of security risk assessment and security testing. In risk-driven model-based security testing we use security risk assessment to support security testing as described above. However, we may also use security testing to support the security risk analysis process. This is referred to as test-driven (model-based) security risk analysis [39, 44].

- **Test-driven model-based security risk analysis** is model-based security risk analysis that makes use of model-based security testing within the security risk analysis to validate and correct risk models.



## 2.2 Problem specification

The problem addressed in this thesis is based on challenges identified within the domain of *testing*, *risk-driven testing*, and *security testing*.

With respect to *testing* in general, we note the following challenges. First, due to the complexity of systems and software it is impossible to exhaustively test every single aspect of any given system under test [73]. We cannot test all possible inputs, combinations of inputs, execution paths, or all potential failures caused by, for example, user interface design errors or incomplete requirements analyses [80]. Second, when testing security/safety/reliability-critical software, testers are faced with the problem of determining the tests that have the ability to reveal faults/errors/failures causing the most severe risks. Third, testing is limited by strict budget and time constraints [51]. In order to address these problems, there has been suggested a number of testing approaches in which the testing is supported by risk assessment [40], that is, risk-driven testing.

With respect to *risk-driven testing* in general, we note the following challenges. First, although a number of risk-driven testing approaches have been suggested, in which only a handful specifically address security, the field is still immature and needs to be improved. Based on our systematic literature review [40], we discovered that there is clearly a need for further research within the field of risk-driven testing in general. The field needs more formality and preciseness as well as dedicated tool support, and there is very little empirical evidence regarding the usefulness of the various approaches. Second, current approaches carry out risk assessment at a high-level of abstraction (for example, business level) and then test accordingly. This is a disadvantage from a testing perspective because it leaves a gap between the risks and the test cases, which are defined at a low-level of abstraction (for example, implementation level). This gap makes it difficult to identify exactly where in the system risks occur, and exactly how the risks should be tested. This also indicates that current approaches focus on risk-driven test planning at a high-level of abstraction for test management purposes, and do not necessarily focus on guiding the tester in designing test cases that have the ability to reveal faults/errors/failures causing the most severe risks.

With respect to *security testing* in general, we note the following challenge. There is a need for proper methods and techniques for security testing because security testing goes beyond simple black-box probing on the application/presentation layer and functional security testing, such as testing whether input validation mechanisms or cryptographic protocol mechanisms are correctly implemented [117]. To properly test for software security, security testers must use a risk-driven approach to security testing, because by identifying risks in the system and creating tests driven by those risks, a software security tester can properly focus on aspects of the system in which an attack is likely to succeed [117]. However, as pointed out in the previous paragraph, only a handful risk-driven testing approaches specifically address security, and the field is still immature and needs more formality and preciseness.

Our objective is to address the aforementioned challenges by developing a *model-based* approach to *risk-driven security testing*, that is:

1. comprehensible to security testers,
2. useful for the purpose of selecting and designing test cases with respect to the risk assessment results,

3. effective in the sense that it produces risk models that are valid and directly testable, and
4. sufficiently rigorous to support the development of tools and methods.

## 2.3 Success criteria

In Chapter 1 and Section 2.2 we outlined the problem areas addressed by this thesis and thereby argued that there is a need for a risk-driven model-based security testing approach that is comprehensible to security testers, useful for test selection and test design, effective in terms of producing valid and directly testable risk models, and sufficiently rigorous for tool and method development. In this section we refine these points into a set of success criteria that our approach, that is the CORAL approach, should fulfill. In the following we motivate and present the success criteria related to the CORAL risk-analysis language, and the CORAL method for risk-driven security testing.

### 2.3.1 The CORAL risk analysis language

The CORAL approach is to be applied within the domain of risk-driven security testing. In order to support model-based security testing, as well as model-based security risk assessment, which are necessary in a model-based approach to risk-driven security testing, the CORAL language must provide graphical constructs that support both security testing, as well as security risk assessment. That is, the graphical notation of the CORAL risk analysis language must be appropriate for the domain of risk-driven security testing. Thus, Success Criterion 1.

**Success Criterion 1.** *The graphical notation must be appropriate for the domain of risk-driven security testing.*

By appropriate for the domain of risk-driven security testing, we mean that the graphical notation should make an integrated use of constructs that are well known within the domain of testing, security, and risk assessment. In addition to be appropriate for the domain of risk-driven security testing, the language has to be appropriate for security testers because they are the main target audience. Thus, Success Criterion 2.

**Success Criterion 2.** *The graphical notation must be appropriate for security testers.*

By appropriate for security testers, we mean that the graphical notation should be comprehensible to and fit for security testers. This includes the use of a modeling notation commonly used by testers for the purpose of test design and execution, as well as for the purpose of risk assessment, which is also an activity commonly carried out by security testers [117, 151].

To fully benefit from model-based testing, which promotes the idea of using explicit abstract models of a system under test and/or its environment to automatically derive tests [153], proper tool support is necessary. However, in order to implement a tool

that correctly supports a given modeling language, the language must be appropriate for tool implementation. By appropriate for tool implementation, we mean that the modeling language must be precisely defined in terms of syntax and semantics. A precise definition of a modeling language is also useful for method developers. Thus, to support tool implementation and method development, the graphical notation of the CORAL language must be precisely defined in terms of syntax and semantics. Thus, Success Criterion 3.

**Success Criterion 3.** *The graphical notation must be appropriate for tool implementation and method development.*

Situations may arise where the information conveyed by CORAL risk models are interpreted differently by different testers. Thus, in order to help software testers to clearly and consistently document, communicate and analyze risks, the CORAL risk analysis language must provide a structured approach to generate the semantics of CORAL risk models in terms of English prose, that is, a natural-language semantics. The natural-language semantics is supposed to be used by testers to document, communicate and analyze risks within the risk-driven testing process. The natural-language semantics must therefore be comprehensible to security testers when conducting risk assessment. Thus, Success Criterion 4.

**Success Criterion 4.** *The English-prose semantics of CORAL risk models must be comprehensible to security testers when conducting risk assessment.*

To support the fulfillment of Success Criteria 1 and 2, we base the graphical notation of the CORAL language on UML interactions [110] (the rationale behind this is explained in detail in Chapters 3 and 5). This means that the CORAL language extends UML interactions with constructs representing risk-related information. Moreover, this also means that the constructs provided by the CORAL language are inherited from UML interactions. We therefore need to make sure that when translating a CORAL risk model into English prose, the resulting English prose of the constructs inherited from UML interactions are consistent with their semantics in the UML standard [110]. Thus, Success Criterion 5.

**Success Criterion 5.** *The English-prose semantics of the constructs inherited from UML interactions must be consistent with their semantics in the UML standard.*

Finally, in order to make sure that the resulting English prose scales with respect to the underlying CORAL risk models, we need to ensure that the complexity of the resulting English prose scales linearly with the complexity of CORAL risk models in terms of size. Thus, Success Criterion 6.

**Success Criterion 6.** *The complexity of the resulting English prose must scale linearly with the complexity of CORAL risk models in terms of size.*

### 2.3.2 The CORAL method for risk-driven security testing

Because the main target audience of the CORAL approach is security testers, the steps in the CORAL method must be in line with the steps commonly carried out in a software testing process (dynamic testing). In particular, the steps related to security risk assessment must be tightly integrated with the testing process in order to assist testers in risk-driven test selection and test design. By following standard testing processes, the CORAL method should be comprehensible to security testers. Thus, Success Criterion 7.

**Success Criterion 7.** *The method must be comprehensible to security testers.*

The overall goal of a risk-driven security testing approach is to focus the testing on the most severe security risks that the system under test is exposed to. Whether or not the testing is focused on the most severe security risks is entirely based on the information on which the risk assessment is based. In a model-based approach to risk-driven security testing, this information is captured in the risk models produced during the risk assessment. In other words, in the CORAL approach, this information is captured in CORAL risk models. It is therefore important that the risk models are valid in order to correctly focus the testing on the most severe security risks. Thus, Success Criterion 8.

**Success Criterion 8.** *The method must produce security risk models that are valid.*

It is also important to keep in mind that the information on which any risk assessment is based will always be limited and will therefore contain an inevitable epistemic uncertainty. However, by conducting a model-based approach to risk-driven security testing, which includes a model-based approach to risk assessment, risk assessment is carried out in a systematic manner, which we believe will help reduce the epistemic uncertainty.

In a model-based approach to risk-driven security testing, the models associated with security risks are used as a basis for designing security tests. However, such models are in general defined at a high level of abstraction, which leaves a gap between the risks represented by the models, and the test cases addressing those risks. In order to bridge this gap, risk models must be defined at a low level of abstraction, which in turn enables security testers to design and execute test cases by making direct use of the low-level risk models. That is, we need to identify directly testable risk models. Recall that the CORAL language extends UML interactions. This means that CORAL risk models are expressed in terms of UML sequence diagrams, and by directly testable risk models we mean risk models that can be reused and specified as test cases based on the interactions in the risk models. Thus, Success Criterion 9.

**Success Criterion 9.** *The method must produce security risk models that are directly testable.*

It is important to identify security risks at the application level, as well as at the source-code level. That is, from a black-box perspective and a white-box perspective, respectively. Although model-based testing is commonly regarded as a pure black-box

testing technique, we believe it may be beneficial to apply model-based testing also for the purpose of white-box testing. The idea of white-box testing, in this context, is to model relevant parts of the system under test, based on information gathered from source code, in order to identify where in the source code a security risk may occur. This, in turn, may help identifying how risks initiated from the application layer affect the system under test at the source code level. Thus, Success Criteria 10.

**Success Criterion 10.** *The method should be appropriate for black-box testing and white-box testing.*

By appropriate for black-box testing and white-box testing, we mean that the complete CORAL method for risk-driven security testing should be conductible both in a white-box testing context, as well as in a black-box testing context.



## Research method

Computer science is a young field that has had an identification problem. There have been disagreements throughout the years on whether computer science is science, engineering, mathematical science or art [33]. For example, Abelson and Sussman [1] claim that computer science is not a science, and back up their claim by contrasting computation with classical mathematics. Brooks [20] argues that science is concerned with the discovery of facts and laws, while computer science is an engineering discipline concerned with building things.

This identification problem is due to the diverse nature in which computer science is applied. Computer science is constantly forming relationships with other fields and thereby settling the foundation for new fields to evolve. Some examples are: autonomic systems, bioinformatics, biometrics, biosensors, cognitive prostheses, cognitive science, DNA computing, immersive computing, neural computing and quantum computing [33]. Although there are some objections to the classification of computer science as a science, there is a widely established agreement that computer science is in fact a science. Hartmanis [61,62], Rosenbloom [124] and Denning [33] point out that that computer science is fundamentally different and is regarded as a fourth great scientific domain that typically forms relationship with the physical sciences (which focus on nonliving matter), life sciences (which focus on living matter), and social sciences (which focus on humans and their societies). Being such an interdisciplinary scientific domain, it is important to utilize the appropriate research method when executing a research project within a given context of computer science.

Depending on the underlying context, a research project within interdisciplinary scientific domains, such as computer science, is typically carried out by utilizing what is commonly referred to as *the scientific method* [37]. It is beyond the scope of this thesis to give a detailed explanation of the scientific method, but the following is a short explanation adapted from Dodig-Crnkovic [37]: “The scientific method is the logical scheme used by scientists searching for answers to the questions posed within science. The scientific method is used to produce scientific theories, including both scientific meta-theories (theories about theories) as well as the theories used to design the tools for producing theories (instruments, algorithms, etc.).” Furthermore, the scientific method is sometimes considered as the classical, standard, or traditional way of performing research [95,140], in which the goal is to seek new knowledge about the world as it is, for example, new knowledge about the cosmos, nature, humans, animals, plants, societies, etc. Following Solheim and Stølen [140], we refer to the scientific

method as the classical research method. Putting it very simply, the starting point of a researcher, when utilizing the classical research method, would be to ask: *What is the real world like?*

However, March and Smith [95] point out that information technology research studies artificial as opposed to natural phenomena. It deals with human creations such as information systems, that is, artifacts. Wieringa [160] defines artifact as something created by people for some practical purpose, for example, algorithms, methods, notations, techniques, and even conceptual frameworks. Based on this, Wieringa [160] defines *design science* as “the design and investigation of artifacts in context”, and explains that the studied artifacts are designed to interact with a problem context in order to improve something in that context. While classical research tries to understand reality, design science is technology oriented and attempts to create artifacts that serve human purposes.

The technology research method [140] is closely related to design science. The technology research method is an iterative method that consists of three consecutive steps: problem analysis, innovation, and evaluation. These steps correspond to the design cycle in design science. The design cycle is also an iterative process and consists of three consecutive steps: problem investigation, treatment design, and treatment validation. However, the design cycle is part of a larger cycle referred to as the engineering cycle, in which a designed and validated treatment is implemented in the problem context, and the implementation is evaluated [160]. In technology research, the artifact is evaluated (also within its intended context) in the evaluation step. The starting point of a researcher, when utilizing the technology research method, would be to ask: *how can one produce new and better artifacts?* Given the scope of this thesis, and the desired results from the work initiated by this thesis (as described in Chapter 2), we have mainly conducted research according to the technology research method.

In the following we first give an overview of the technology research method, and then we give an overview of evaluation strategies to evaluate an artifact, and finally we explain how we applied the research method in the work leading to this thesis.

### 3.1 The technology research method

The technology research method is iterative and motivated by the need for a new artifact, or the need to improve an existing artifact [140]. The researcher starts by identifying a set of requirements to the artifact. Depending on the artifact, the requirements may be identified from the viewpoint of existing users, potential/new users, as well as other stakeholders, such as people who seek to obtain economical gain by maintaining or selling the artifact. Then, having identified the requirements, the researcher aims to invent an artifact which fulfills the requirements. This is the step in which the researcher needs to be innovative and use his/her creativity and technical expertise. Finally, having developed the artifact, the researcher needs to evaluate the artifact to check whether it fulfills the requirements and thereby whether it satisfies the (potential) need. If the evaluation yields successful results, the researcher may argue that the artifact satisfies the need. If the results are negative, the researcher may try to adjust the artifact accordingly and reiterate the evaluation. Thus, the technology research method consists of the following three steps (see Figure 3.1).

1. **Problem analysis:** The researcher captures a potential need for a new or im-



proved artifact by interacting with possible users and other stakeholders.

2. **Innovation:** The researcher tries to construct an artifact that satisfies the potential need. The overall hypothesis is that the artifact satisfies this need.
3. **Evaluation:** Based on the potential need, the researcher formulates predictions about the artifact and checks whether these predictions come true. Predictions are evaluated/tested by making use of evaluation strategies (described in Section 3.2). If the results are positive, the researcher may argue that the artifact satisfies the need.

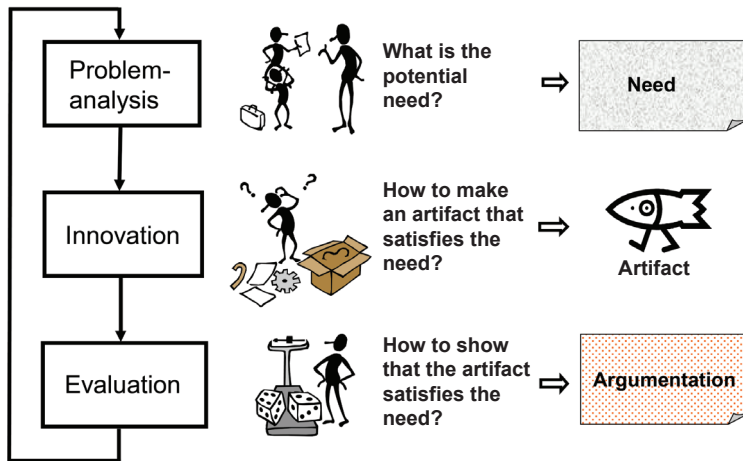


Figure 3.1: The main steps in the technology research method (adapted from Solheim and Stølen [140]).

According to Solheim and Stølen [140], technology research shares a common pattern with classical research, while action research may be regarded as a special case of technology research. However, the objectives in each research approach are different. Table 3.1 shows the main objectives of classical research, technology research, and action research.

- As mentioned above, the starting point of a researcher in classical research would be to ask: *What is the real world like?* That is, there is a need for a new theory to explain the world as it is. In classical research, the resulting new theory is evaluated by comparing it to the relevant part of the real world, and the overall hypothesis is that the new theory/explanation agrees with reality.
- According to Davison et al. [32], the application focus of action research involves solving organizational problems through intervention while at the same time contributing to knowledge. However, the researcher and the researcher's activities are included in the research object. In other words, the researcher and the object under study are not clearly separated [140]. The need is defined in terms of action plans addressing the organizational problems. The action plans are executed in the organization and evaluated by examining their effect in terms of solving the

Table 3.1: The main objectives of classical research, technology research, and action research [140].

	<b>Classical research</b>	<b>Technology research</b>	<b>Action research</b>
<b>Problem</b>	Need for new theory	Need for new artifact	Need for improved organization
<b>Solution</b>	New explanations (new theory)	New artifact	Action plan
<b>Compares solution to:</b>	Relevant part of the real world	Relevant need	Relevant need for improvement
<b>Overall hypothesis</b>	The new explanations agree with reality	The new artifact satisfies the need	The actions result in an improved organization satisfying the need

underlying problem. The overall hypothesis is that an action  $A$  implies that the organization's need for improvement is satisfied.

## 3.2 Evaluation strategies

According to McGrath [97], evaluation strategies are carried out to gather evidence to assess the degree of generality, precision, and realism of the artifact under evaluation. This is illustrated in Figure 3.2. Generality indicates that results are valid across populations. Precision indicates that the measurements are precise. Realism indicates that evaluation is performed in environments similar to reality.

When gathering a batch of research evidence, one is always trying to maximize the scores on generality, precision and realism of the prediction under evaluation. While it is most desirable to maximize all of these three qualities simultaneously, it is, however, an impossible act [97]. Broadly speaking, it is therefore important to consider factors such as the nature of the predictions (that is, whether the predictions address the generality, precision or realism of an artifact), the maturity of the artifacts addressed by the predictions, and the available resources (time, cost and people) when choosing evaluation strategies. In the following, we give an overview of the most common evaluation strategies.

- **Field study** is a direct observation of “natural” systems, with little or no interference from the researcher. Field studies are strong on realism but lack precision and generality because they are difficult to replicate.
- **Field experiment** is similar to field study in the sense that it is an experiment carried out in a natural environment. However, in field experiments, the difference

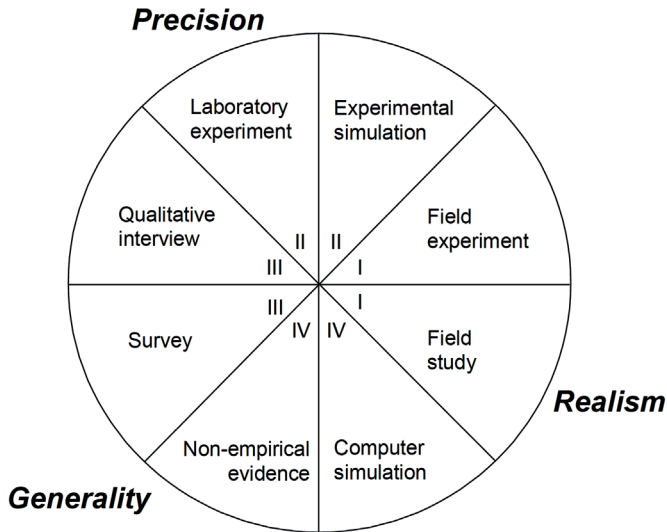


Figure 3.2: Evaluation strategies (adapted from McGrath [97]).

is that the researcher intervenes and manipulates a certain factor.

- **Experimental simulation** is a laboratory test simulating a relevant part of the real world.
- **Laboratory experiment** gives the researcher a large degree of control and the possibility to isolate the variables to be examined. It scores high on precision but lacks realism and generality.
- **Qualitative interview** is a collection of information from a few selected individuals. The answers are more precise than those of a survey, but cannot be generalized to the same degree.
- **Survey** is a collection of information from a broad and carefully selected group of informants. The information is typically collected via questionnaires or interviews. Surveys have a high degree of generality, however, they are less controlled than experiments and therefore lack precision. Moreover, the likelihood of bias on the part of the respondents may weaken the realism of a survey.
- **Non-empirical evidence** is argumentation based on logical reasoning. It scores high on generality, but low on realism and precision because it is not empirical.
- **Computer simulation** is operating on a model of a given system. This means that computer simulations are system-specific and therefore score higher on realism than non-empirical evidence, but lower on generality.

These eight strategies are further divided into the following four groups of pairs as shown in Figure 3.2.

- **I** The evaluation is performed in a natural environment.

- **II** The evaluation is performed in an artificial environment.
- **III** The evaluation is independent of environments.
- **IV** The evaluation is independent of empirical measurements.

In addition to the above evaluation strategies, Wieringa [160] and Zelkowitz et al. [173] point out the following additional strategies.

- **Case study** is an empirical inquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified [128]. According to Yin [167], “a case study allows investigators to focus on a “case” and retain holistic and real-life perspective.” For example, when studying a method for security testing, a software development life cycle, or organizational and managerial processes. The results of a case study can help determine to what extent an artifact is useful, comprehensible and scalable.
- **Literature review** examines existing publications related to a topic and a scope. The method is often used to identify the current state of art and state of practice within a field. Moreover, it may also be useful to confirm an existing hypothesis. However, a weakness with a literature search is that it may be biased in the selection of published works [173].
- **Expert opinion** is an evaluation strategy in which the design of an artifact is submitted to a panel of experts, who imagine how such an artifact will interact with problem contexts imagined by them and then predict what effects they think this would have [160]. However, this kind of validation is limited to the expert’s understanding of the artifact under evaluation.
- **Technical action research** is the use of an artifact prototype in a real-world problem to help a client and to learn from this [160]. This is typically carried out as one of the last stages before an artifact moves from the “laboratory” to the real world.

### 3.3 How we have applied the research method

As mentioned in the introduction of this chapter, we have mainly made use of the technology research method. Following its iterative nature we changed and improved the artifacts, that is, the CORAL approach, and its related success criteria as new insight was gained. The documentation of the three phases, that is, the problem analysis, the innovation, and the evaluation are found in Chapter 2, Chapter 5, and Chapter 7, respectively. Figure 3.3 illustrates our research process for developing the artifacts. Each activity depicted in Figure 3.3 is explained in the following paragraphs.

**Develop conceptual model.** In order to establish a common platform with respect to concepts relevant to the topic of this thesis, we developed a conceptual framework in which we clarify the notions of *security*, *testing*, *security testing*, *risk assessment*, *security risk assessment*, *model*, *model-based testing*, *model-based security testing*, *model-based security risk assessment*, and finally *risk-driven model-based security testing* (RMST). Our conceptual framework is developed with respect to established concepts from state of the art literature, and is documented in Section 2.1. While developing the conceptual framework, we discovered that not only may security risk assessment support security testing, but also that security testing may support security risk analysis. As clarified in Section 2.1.10, we refer to the latter as *test-driven security risk analysis* [39, 44].

**Carry out systematic literature review.** To obtain a holistic picture of the field addressed by this thesis, we carried out a systematic literature review in which we reviewed approaches that combine risk analysis and testing in general. That is, in addition to approaches addressing security, we also reviewed approaches that address other qualities, such as safety and reliability. We identified a total of 25 approaches related to risk-driven testing, and a total of 3 approaches related to test-driven risk analysis. The complete literature review is documented in Sections 4.4 and 4.5, as well as in Paper 1, which also describes the review process. The literature review revealed the following main issues.

1. The field needs more formality and preciseness, as well as dedicated tool support.
2. There is very little empirical evidence regarding the usefulness of the approaches.
3. Risk assessment is carried out at a high-level of abstraction (for example, business level), while test cases are defined at a low-level of abstraction (for example, implementation level). This introduces a gap between identified risks and the test cases exploring the risks.

**Identify success criteria for an approach to RMST.** To address the above issues, we defined an overall objective to develop a method for risk-driven security testing supported by a graphical modeling language that is:

1. comprehensible to security testers,
2. useful for the purpose of selecting and designing test cases with respect to the risk assessment results,
3. effective in the sense that it produces risk models that are valid and directly testable, and
4. sufficiently rigorous to support the development of tools and methods.

We refined these goals into the following ten success criteria. These success criteria are explained in detail in Chapter 2. Success criteria 7 through 10 were quite persistent throughout the work, while success criteria 1 through 6 were adjusted as new knowledge was obtained from the industrial case studies.

- **Success Criterion 1.** The graphical notation must be appropriate for the domain of risk-driven security testing.
- **Success Criterion 2.** The graphical notation must be appropriate for security testers.
- **Success Criterion 3.** The graphical notation must be appropriate for tool implementation and method development.
- **Success Criterion 4.** The English-prose semantics of CORAL risk models must be comprehensible to security testers when conducting risk assessment.
- **Success Criterion 5.** The English-prose semantics of the constructs inherited from UML interactions must be consistent with their semantics in the UML standard.
- **Success Criterion 6.** The complexity of the resulting English prose must scale linearly with the complexity of CORAL risk models in terms of size.
- **Success Criterion 7.** The method must be comprehensible to security testers.
- **Success Criterion 8.** The method must produce security risk models that are valid.
- **Success Criterion 9.** The method must produce security risk models that are directly testable.
- **Success Criterion 10.** The method should be appropriate for black-box testing and white-box testing.

**Develop approach combining security risk assessment and security testing.**

In an initial attempt to fulfill the success criteria, we developed an approach consisting of three phases. In this approach, we were interested in exploring the two strategies for the combined use of security risk assessment and security testing. That is, not only were we interested in supporting security testing with security risk assessment (risk-driven security testing), but also in supporting security risk analysis with security testing (test-driven security risk analysis). The three-phased approach is carried out as follows. Phase 1 expects a description of the target of evaluation. Then, based on this description, the security risk assessment is planned and carried out. The output of Phase 1 is security risk models, which is used as input to Phase 2. In Phase 2, security tests are identified based on the risk models and executed. The output of Phase 2 is security test results, which is used as input to the third and final phase. In the third phase, the risk models are validated and corrected with respect to the security test results. In this approach, we made use of the CORAS risk analysis language [91] to model risks and to carry out security risk assessment. This approach is documented in detail in Paper 5.

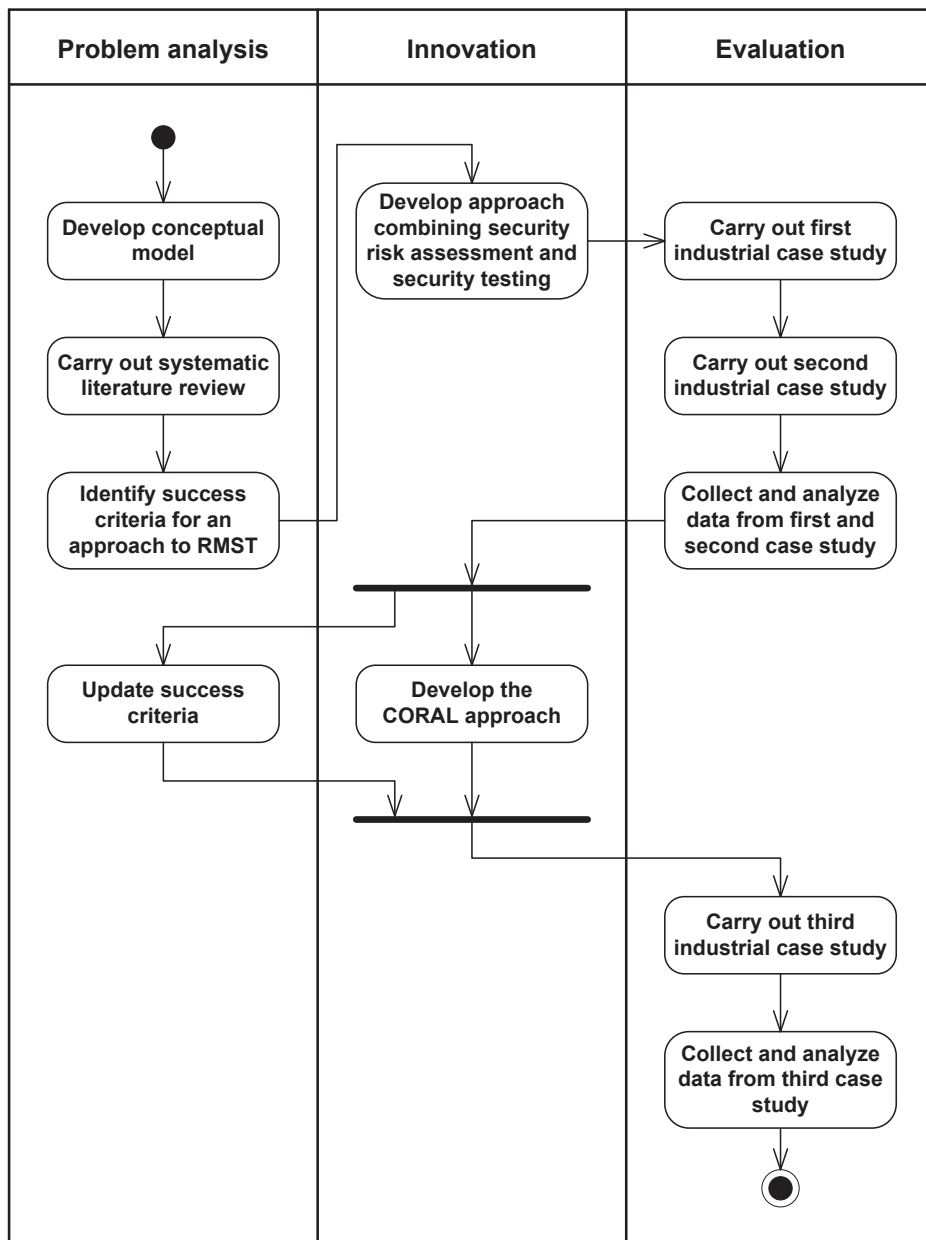


Figure 3.3: Research process for the development of the artifacts.

**Carry out first industrial case study.** Using the approach described in the previous paragraph, we carried out an industrial case study in which we analyzed a multilingual web application, which is designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans. The web application was

deployed on the servers of a third party service provider, as well as maintained by the same service provider with respect to infrastructure. However, the web application was completely administrated by the client commissioning the case study for business purposes, such as customizing the web application for each customer, as well as patching and updating features of the web application. The focus of this case study was to analyze the system to identify security risks that may be introduced internally by the client when administrating the application, as well as security risks that may be introduced externally via features available to customers. In this case study, it was decided not to consider security risks related to infrastructure because this was a contractual responsibility of the service provider. This case study is referred to as Case Study 1 in Paper 5.

The objective of this case study was to assess how useful risk assessment is for identifying and designing tests (risk-driven security testing), as well as to assess how useful testing is for validating and correcting security risk models (test-driven security risk analysis).

**Carry out second industrial case study.** In the second case study, we analyzed a mobile application designed to provide various online financial services to the customers on their mobile devices. In contrast to the first case study, this application was deployed on the local servers of the client commissioning the case study. The online financial services were accessible only via a dedicated mobile application installed on a mobile device. Moreover, all aspects related to maintenance and administration was the responsibility of the client. However, some few aspects related to content displayed to the customers were directly handled by a third party. The main focus of this case study was to analyze the mobile application to identify security risks that may be introduced from an external point of view. That is, we identified security risks that may be introduced by the third party when administrating the few aspects of the mobile application, as well as security risks that may be introduced via features available to customers. This case study is referred to as Case Study 2 in Paper 5.

The objective of this case study was the same as the objective of the first case study.

**Collect and analyze data from first and second case study.** In short, the results from both case studies may be summarized as follows.

With respect to risk-driven security testing, we discovered that threat scenarios are a good starting point for identifying security test cases. Having CORAS risk models [91] as the starting point, we may identify tests with respect to single threat scenarios, or with respect to a chain of threat scenarios (CORAS risk models are represented as directed acyclic graphs). The problem, however, was that we were only able to identify high-level test procedures due to the high-level nature of CORAS risk models. This, in turn, caused us to refine each high-level test procedure into an executable test case in a manual and ad hoc manner.

With respect to test-driven security risk analysis, we discovered that the test results are useful for correcting risk models in terms of adding or deleting vulnerabilities, as well as editing likelihood values. Moreover, the test results also proved to be useful for validating risk models in terms of discovering the presence or absence of presumed vulnerabilities.

The problem related to risk-driven security testing indicated the need for a rigorous process to select and design test cases. In order to bridge the gap between high-level



risks (from which high-level test procedures are defined) and the low-level test cases actually testing the risks, we need to express risks at the level of test cases. This also means that there is a need for a risk analysis language capable of expressing risks as well as test cases.

**Update success criteria and develop the CORAL approach.** In light of the discoveries made through the case studies described above, we decided to develop a method for risk-driven security testing supported by a domain-specific language to help security testers select and design test cases based on the available risk picture. In search of a suitable notation for our language, we reviewed the literature and found that UML interactions, along with statechart diagrams and class diagrams, are the most commonly used notations for testing purposes [10, 34, 111]. We selected UML interactions as the basis for our notation, and thus developed an interaction based risk analysis language (CORAL).

Based on these design decisions, we updated success criteria 1 through 6 and developed the CORAL approach. The CORAL approach is presented and described in Papers 2, 3 and 4.

**Carry out third industrial case study.** In the third case study, we applied the CORAL approach and tested the same system as described in the first case study. However, in this case study, we tested two features not considered in the first case study: a feature for selling shares, and a feature for exercising options for the purpose of buying shares in a company. The objective of the case study was to evaluate to what extent the CORAL approach helps security testers in selecting and designing test cases. The test report delivered to the client that commissioned the case study describes, in addition to the test results, risk models and security tests designed with respect to the risk models. Our hypothesis was that the report is good in the sense that (1) the risk models are valid, and (2) the threat scenarios represented by the risk models are directly testable. By a directly testable threat scenario, we mean a threat scenario that can be reused and specified as a test case based on its interactions. Thus, the units of analysis in this case study were the risk models. The case study is documented in Paper 4.

**Collect and analyze data from third case study.** In short, the case study results may be summarized as follows.

With respect to our first hypothesis, that is, *the CORAL approach is effective in terms of producing valid risk models*, we noted two supporting observations. First, we identified in total 21 risks, and 11 of these risks were considered as most severe, while the remaining 10 risks were considered as low risks. By testing the 11 risks we identified 11 vulnerabilities, while by testing the remaining 10 risks we identified only 2 vulnerabilities. Second, we identified all relevant security risks compared to previous penetration tests. In addition, we identified five new security risks and did not leave out any risks of relevance for the features considered.

With respect to our second hypothesis, that is, *the CORAL approach is effective in terms of producing directly testable threat scenarios*, we noted the following supporting observation. The results obtained in the case study point out that all threat scenarios were directly testable. We believe this result is generalizable because, in the CORAL

approach, risks are identified at the level of abstraction testers commonly work when designing test cases [10,34,111]. This is also backed up by the fact that we made direct use of all threat scenarios as security test cases.

To complement the case study results, we have also carried out an analytical evaluation of the CORAL language using the SEQUAL framework [85]. SEQUAL is a general framework used for discussing and evaluating the quality of models, as well as the quality of modeling languages. This is documented in Chapter 7.

## State of the art

In this chapter, we first give an overview of state of the art related to modeling (Section 4.1) and model-based testing (Section 4.2) in general. Then, in Sections 4.3, 4.4, and 4.5 we present state of the art of relevance for the artifacts developed in this thesis. For a more detailed discussion on the relationship of our artifacts to the literature, the reader is referred to Section 7.3. The reason why this is not discussed in this chapter is because we first need to present our artifacts in detail, which we do in Chapter 5, before we relate them to the literature presented in this chapter.

### 4.1 Modeling

The usage of graphical models to describe computer systems may be traced back to the 1950's [169], and the graphical notations commonly applied today may be traced back to the 1970's. Chen [26] proposes an entity-relationship model, which builds on the earlier principles of a network model [7], a relational model [31], and an entity set model [136]. Chen [26] points out that instead of using the network model, the relational model, and the entity set model separately, systems should be modeled in a unified manner to reflect a more natural view. According to Chen [26], the entity-relationship model adopts a more natural view in the sense that the real world consists of *entities* and *relationships*. The advantages of describing systems in terms of graphical models (versus sentential models) is also pointed out from a cognitive science perspective. Larkin and Simon [87] point out that sentential representations are one-dimensional because information is indexed by position in a list (like the propositions in a text), while graphical representations are two-dimensional because information is indexed by location in a plane. This two-dimensional property makes graphical models superior to a verbal description for solving problems. Research in diagrammatic reasoning shows that the form of representations has an equal, if not greater, influence on cognitive effectiveness as their content [104]. The following points list the main advantages of graphical models as given by Larkin and Simon [87].

- Graphical models can group together all information that is used together, thus avoiding large amounts of search for the elements needed to make a problem-solving inference.
- Graphical models typically use location to group information about a single element, avoiding the need to match symbolic labels.

- Graphical models automatically support a large number of perceptual inferences, which are extremely easy for humans.

Sentential representations do not provide the abovementioned advantages, but does, for example, preserve relations such as temporal or logical sequence [87]. Thus, we may divide modeling notations in two main categories: graphical and sentential. Moreover, we may group modeling notations in various modeling paradigms. According to Lam-sweerde [154] and Utting et al. [153], modeling notations can be grouped into seven modeling paradigms: state-based (or pre/post) notations, transition-based notations, history-based notations, functional notations, operational notations, stochastic notations, and data-flow notations. In the following, we give a brief overview of these paradigms.

**State-based (or pre/post) notations.** State-based notations are used to model a system as a collection of variables [153]. These models represent admissible system states at some arbitrary snapshot of the internal state of the system, as well as operations that may modify the states [154]. The states are described in terms of sets, sequences, relations, and functions, and the operations are described in terms of pre/post conditions [63]. Examples of state-based (or pre/post) notations are Z [143, 144], Vienna Development Method (VDM) [76], and B [2].

**Transition-based notations.** Transition-based notations are used to model the required transitions from one system state to another (including the transition from a state  $S$  to itself) [154]. These models are typically represented as nodes with edges connecting the nodes. The nodes represent states in the system, while the edges represent transitions between the states. Transitions between states are triggered by the dispatching of series of events. Examples of transition-based notations are finite state machines (FSM) [89], and Statecharts such as UML State Machines [110].

**History-based notations.** History-based notations are used to model a system by describing the allowable traces of its behavior over time [153]. These kinds of models may represent time in various ways, for example, linear or branching, discrete or continuous, and time points or time intervals. A well known example of a history-based notation is UML interactions [110], which is inspired by Message Sequence Charts [126].

**Functional notations.** Functional notations specify a system as a structured collection of mathematical functions [154]. Functional notations may be divided into two groups: algebraic specifications and high-order functions. In the context of model-based testing, algebraic specifications focus mainly on data type abstraction and may be used for testing data type implementations [52]. High-order functions are grouped into logical theories in terms of type definitions, variable declarations, and axioms defining the various functions in the theory [154].

**Operational notations.** Operational notations are used to model a system in terms of a structured collection of executable processes. These notations are particularly useful for describing distributed systems and communication protocols [153]. An example of an operational notation is the Petri net notation [53].

**Stochastic notations.** According to Utting et al. [153]: “Stochastic notations describe a system by a probabilistic model of the events and input values and tend to be used to model environments rather than the system under test. For example, Markov chains are used to model expected usage profiles, so that the generated tests exercise that usage profile.”

**Data-flow notations.** According to Utting et al. [153]: “Data-flow notations concentrate on the data rather than the control flow. Prominent examples are Lustre, and the block diagrams of Matlab Simulink, which are often used to model continuous systems.”

The above paradigms address both graphical and sentential notations, and several paradigms can often be represented in one single notation [153]. Similar paradigms are provided by Krogstie [85] with a particular focus on graphical notations. Based on a survey of state of the art approaches for conceptual modeling, Krogstie [85] presents the following graphical modeling perspectives.

- **Behavioral perspective.** Systems are modeled in terms of states and transitions between the states, similar to transition-based notations mentioned above. Example notation: finite state machines (FSM) [89].
- **Functional perspective.** Systems are modeled in terms of transformations. A transformation is defined as an activity, which transforms a set of phenomena to another set of phenomena. Other terms used for the main concept are function, process, activity and task. Example notation: data flow diagrams (DFD) [50].
- **Structural perspective.** Systems are modeled in terms of entities describing the static structure of the system. Example notation: entity-relationship [26].
- **Goal and rule perspective.** Systems are modeled in terms of goals and rules, where a rule is something that influences the actions of a set of actors [85]. Example notation: entity-relationship with time [96].
- **Object perspective.** Systems are modeled in terms of objects, processes, and classes, and other concepts commonly found in object-oriented programming languages [85]. Example notation: the Unified Modeling Language [110].
- **Communication perspective.** The communication perspective is based on language/action theory [6,134]. According to Krogstie [85], “the basic assumption of language/action theory is that persons cooperate within work processes through their conversations and through mutual commitments taken within them.”
- **Actor and role perspective.** Systems are modeled in terms of actors (or agents) and roles. Typically, the nodes represent a social actor/role, and the edges connecting the nodes represent a relationship between the actors. Example notation:  $i^*$  (i star) [170].
- **Topological perspective.** The topological perspective is related to the modeling of the topological ordering between different concepts, for example, the modeling of where different tasks shall be performed. Example notation: place-oriented modeling [56].

Modeling techniques within the above perspectives are mostly used for human sense-making, supporting human communication, computer-assisted analysis, quality assurance (model-based testing), model deployment and activation, and to give the context for a traditional system development project [85].

## 4.2 Model-based testing

The basic ideas of model-based testing can be traced back to the 1970s [29,153]. Since then, numerous approaches for model-based testing have been suggested. Dias-Neto et al. [34] provide a systematic literature review in which they analyze 202 model-based testing approaches. A similar comprehensive review is also provided by Hierons et al. [63]. It is beyond the scope of this thesis to describe the various approaches, hence the reader is referred to the aforementioned surveys for a detailed explanation and classification of exiting model-based testing approaches. Based on the existing approaches, Utting et al. [153] provide a generic model-based testing process which is divided into three phases: model specification, test generation, and test execution. The phase related to model specification focuses on building a model of the system under test, choosing test selection criteria, and specifying tests. The phases related to test generation and test execution focus on the usage of tools to generate and execute test cases. Utting and Legeard [152], Utting et al. [153], Schieferdecker [130], and Hierons et al. [63] provide an overview of current model-based testing tools.

According to Utting and Legeard [152], the benefits of model-based testing may be grouped into six areas: system under test fault detection, reduced testing cost and time, improved test quality, requirements defect detection, traceability, and requirements evolution. The following points give a brief explanation of these benefits. The reader is referred to Utting and Legeard [152] for a detailed explanation.

- **System under test fault detection.** Model-based testing is effective in terms of identifying faults in systems that are under development, as well as systems that have been deployed and used for some time. Moreover, numerous industrial case studies show that the number of faults detected by applying a model-based testing process is in most cases greater than the number of faults detected by a manual process [152].
- **Reduced testing cost and time.** Model-based testing reduces testing cost and time because the time needed to write and maintain the model, plus the time spent on directing the test generation, is less than the time needed to manually design and maintain a test suite.
- **Improved test quality.** Model-based testing improves test quality because the test design and generation process is systematic and repeatable, and because generated tests are associated with system requirements in a clear and transparent manner.
- **Requirements defect detection.** Model-based testing helps detecting errors in the system requirements and system design because the model is precise enough to be analyzed by computer. Of course, this requires that the analyzed model is free of errors potentially introduced by the tester.

- **Traceability.** Model-based testing facilitates traceability in terms of relating each test case to the model, the test selection criteria, and to the informal system requirements.
- **Requirements evolution.** Model-based testing requires less effort in maintaining test suites with respect to changing and evolving requirements, compared to manual testing.

As pointed out in Section 4.1, there is a variety of modeling paradigms that may be applied in model-based testing, and in practice, several paradigms can be represented in one single notation [153]. However, based on the systematic literature review provided by Dias-Neto et al. [34], the three most popular notations used in model-based testing are statechart diagrams, class diagrams, and sequence diagrams. These are followed by other notations commonly used within model-based testing, such as the Object Constraint Language (OCL) [112], finite state machines [89], activity diagrams [110], object diagrams [110], and the Z notation [143, 144]. This does not necessarily mean that the three most popular notations are better suited for model-based testing, compared to the other notations, because the various notations focus on different aspects of the system under test, that is, states, transitions, traces, etc. However, this does indicate that most approaches focus on testing a system with respect to transitions between various states in the system, or with respect to traces of events (occurring over time) in the system.

Model-based testing is mostly applied at the level of system testing, followed by integration testing, unit testing, and regression testing. This ranking is based on the systematic literature review reported by Dias-Neto et al. [34], which was conducted in 2007. In a recent survey, conducted in 2014, Binder et al. [11, 12] investigates the state of model-based testing in practice by surveying 100 practitioners. The survey indicates that this trend has in fact not changed over the years, and that model-based testing is still mostly used at the level of system testing, followed by integration testing, unit testing, and regression testing [12]. Moreover, the survey shows that model-based testing is mostly carried out to test web applications, enterprise IT applications (including packaged applications), and embedded controller applications (real-time). In addition, the survey shows that model-based testing is not only used for automated test generation and execution, but also for designing test cases for manual executions, and for other purposes such as identifying test data and test suites, as well as for documenting test cases. From a human factor perspective, the survey shows that practitioners usually have very high expectations to model-based testing, and that model-based testing does not completely fulfill the high expectations. On the positive side, however, nearly all participants in the survey rate model-based testing as an effective testing approach.

As pointed out by Binder et al. [12], model-based testing clearly has the potential of increasing the rigorousness, efficiency, and effectiveness of software testing in the industry. However, a recent survey addressing the practice on software testing in Canada shows that model-based testing has not fully arrived in the industry [51]. This is also in line with the findings of a similar survey, conducted in 2011, which addressed the practice on software testing in Germany, Switzerland, and Austria [130]. On the positive side, compared to older surveys, Garousi et al [51] report on a more positive trend with respect to the usage of model-based testing in the industry.

## 4.3 Model-based security testing

Model-based security testing is a relatively new field. Most of the approaches within model-based security testing have been published after 2010. Model-based security testing is in particular focused on the systematic and efficient specification and documentation of security test objectives, security test cases and test suites, as well as to their automated or semi-automated generation [131]. Current approaches may be grouped into four categories [131].

- Approaches with main focus on functional security testing.
- Approaches based on fuzzing.
- Approaches with main focus on pattern-based security testing.
- Approaches with main focus on threat-based security testing.

Approaches addressing functional security testing mainly focus on testing the implementation of security features, in order to evaluate whether the system exhibits the expected security behavior. Fuzzing is a black-box testing technique in which the system under test is stressed with invalid, unexpected, or random inputs at its interfaces. The purpose is to identify vulnerabilities causing failures in the system [129]. Approaches addressing pattern-based testing focus on testing a system with respect to known attack patterns, in order to evaluate whether the system is vulnerable to certain malicious attacks. Threat-based security testing approaches focus on identifying potential threat scenarios the system under test may be exposed to. The threat scenarios are then used as a basis to identify security tests, which in turn are executed on the system under test. The purpose is to evaluate whether the system under test is vulnerable to certain security risks caused by the threat scenarios. The terms threat-based testing and risk-based testing are sometimes used interchangeably. However, there are some fundamental differences: In addition to identifying threat scenarios, risk-driven testing approaches estimate and evaluate security risks in order to focus the testing on the most severe security risks. Risk-driven testing approaches are discussed in detail in Sections 4.4 and 4.5.

In the following, we present relevant approaches for the aforementioned categories. However, some of the approaches may belong to more than one category. The categorization is an attempt to group papers with respect to their main focus, so that those that are related are discussed close to each other.

### 4.3.1 Approaches with main focus on functional security testing

Jürjens and Wimmel [79,161] propose a method for specification-based testing for the purpose of detecting vulnerabilities in security-critical systems. In particular, they focus on security testing firewalls [79], and transaction systems [161]. The proposed approach specifies a system by making use of Focus, which is a mathematical and logical framework for the specification, refinement, and verification of distributed, reactive systems [21]. Test cases are generated from the system specifications. Jürjens also suggests an approach to specification-based security testing by making use of the



language UMLsec [78]. UMLsec is a UML profile that provides security concepts in order to support the development of secure systems [77].

Blackburn et al. [13] summarize the results of applying a model-based approach to automate functional security testing. The process for the automated security test approach begins with the development of a model, by making use of a commercial tool named SCRtool, representing functional security requirements. Then, they make use of a commercial tool set named T-VEC in order to translate the functional security requirements to a test specification, from which test vectors are generated. Then, the T-VEC tool is used to generate test drivers for the purpose of executing tests. Finally, test results are compared with the expected results from the test vectors to determine whether the system under test complies with the functional security requirements. The approach is targeted towards Java applications and database servers.

Mouelhi et al. [106] propose a model-driven approach for specifying, deploying and testing access control policies in Java applications. The approach consists of five steps. The purpose of the first step is to build a security model of the application. The security model is a platform independent model which captures the access control policies defined in the requirements of the system. In the second step, the security model is automatically transformed into Policy Decision Points (PDP), which are the points where policy decisions are made. In the third and fourth step, the PDPs are integrated into the functional code of the application, and to reduce the risk of introducing mistakes during the integration, the approach makes use of aspect oriented programming techniques to support systematic integration. Finally, in step five, the resulting integrated application is tested in terms of mutation testing to ensure that the final running code conforms to the security model. Xu et al. [165] also present a model-based approach for testing access control implementations. Similar to the aforementioned approach, Xu et al. also carry out tests in terms of mutation testing. However, their approach generates tests automatically, and it generates test code in a variety of languages, such as Java, C, C++, C#, and HTML/Selenium IDE.

Katkalov et al. [81] introduce a method to test the functionality and security of a distributed application based on cryptographic protocols. The approach is model driven and represents test cases in terms of UML activity diagrams. The test cases are defined at the implementation level by automatically transforming the UML activity diagrams into Java test code. The approach also provides a custom domain-specific language tailored to design and test security protocols, as well as guidelines to support the modeler in designing test cases for complex systems.

### 4.3.2 Approaches based on fuzzing

Schieferdecker et al. [131] provide a model-based fuzzing approach in which the valid sequences of a system, specified in UML sequence diagrams, are altered in terms of changing the order and appearance of messages. This is done by either rearranging/altering/removing messages in a valid sequence in order to obtain an invalid sequence, or by rearranging/altering/removing a group of messages encapsulated by control structures, that is, interaction operators, of UML 2.x sequence diagrams. The former enables straight-lined sequences to be fuzzed in terms of removing, repeating, replacing, or relocating messages in the sequence diagram. The latter allows more sophisticated fuzzing by applying similar fuzzing techniques at the level of interaction operators that avoids less efficient random fuzzing. The latter also includes fuzzing

the interaction operators, for example by negating a guard of an interaction operand, changing the bounds of a loop operand, or disintegrate combined fragments. The approach was developed in a European ITEA2 project named DIAMONDS, and the detailed description of the approach can be found in the public project publications [92]. Schneider et al. [132] provide a model-based fuzzing approach based on the aforementioned approach. However, in this approach they focus particularly on generating new test cases during test execution, as well as on taking previous test results into consideration when generating new test cases, in order to reduce the test execution time.

Wang et al. [156, 157] suggest a model-based fuzzing approach to improve the efficiency of security testing of database management systems, as well as network services. The approach consists of a general framework for model-based behavioral fuzzing that is guided by final state machine models. The final state machine models support state-aware fuzzing, automate a so-called multi-phase fuzzing process, and detect test-script errors.

Johansson et al. [75] present a model-based fuzzing method which aims to improve conformance testing. The method is named T-Fuzz and is a fuzzing framework for telecommunication networks. As mentioned by the authors, the T-Fuzz framework shares similarities with the approach provided by Schieferdecker et al. [131], and Schneider et al. [132]. However, compared to these approaches, T-Fuzz “can automatically generate values for all models in TTCN-3 without relying on third party generators or mutators [75]”.

### 4.3.3 Approaches with main focus on pattern-based security testing

Lebeau et al. [88] aim at improving the accuracy and precision of vulnerability testing of web applications, by means of models and test patterns, in order to avoid both false positives and false negatives. They suggest an approach to automate model-based vulnerability testing of web applications. The approach is supported by a commercial tool named CertifyIt [138]. The approach consists of four activities. In the first activity, the security test engineer formalizes test purposes with respect to a set of vulnerability test patterns. The rationale is that the generated test cases have to cover the test purposes. In the second activity, the tester defines a model that captures the behavioral aspects of the system under test in order to generate consistent sequences of stimuli. In the third activity, abstract test cases are automatically generated with respect to the artifacts defined in the two previous activities. In the fourth activity, the tester prepares executable test cases by specifying the abstract test cases as executable scripts. Finally, the test cases are automatically executed, and the test results are recorded.

Bozic et al. [17–19] also suggest a model-based security testing approach specialized for web applications. The approach makes use of UML state machines in order to model certain web-vulnerability attack patterns. In particular, they focus on SQL injection attack patterns, as well as cross site scripting attack patterns. Other attack patterns may also be modeled. Based on the modeled attack patterns, they automatically execute the attacks, that is, the security test cases.

### 4.3.4 Approaches with main focus on threat-based security testing

Wang et al. [158] provide a model-driven approach to test possible violations of security policies. Threats to security policies are modeled using UML sequence diagrams. Based on these models, a set of threat traces are extracted, where each threat trace represents a sequence of events that should not occur during system execution. Each threat trace is matched with actual execution traces in the system, and if an execution trace is an instance of a threat trace, security violations are reported.

Armando et al. [5] present an approach to security testing of web-based applications in which test cases are automatically derived from counterexamples found through model checking. The approach consists of four steps and requires an informal description of the protocol used by the web-application, as well as a description of the system under test and its environment. In the first step, an abstract model, amenable to formal analysis of the protocol, is formulated and message mapping information is specified. In the second step, the abstract model is automatically analyzed via model checking. If one of the expected security properties is violated, a counterexample is discovered. If no counterexample is found, the procedure terminates. In the third step, an abstract test sequence, in terms of UML sequence diagrams, is generated from the counterexamples. Finally, in the fourth step, a concrete test sequence, that is, a test case, is generated from the abstract test sequence and then executed. The result is recorded and a verdict is assigned accordingly.

Zulkernine et al. [175] propose an attacker-centric approach for automatic model-based security testing. Attack scenarios are modeled using formalisms based on extended abstract state machines. The models represent system attack behavior in terms of states, conditions, and transitions. The attack scenarios are made executable by developing a so-called suitable attack signature generator, based on the attack scenarios. The resulting attack signatures are then passed to an attack test engine, which executes the attack, that is, the security test, on the system under test.

Botella et al. [14] present an approach similar to the one presented by Lebeau et al. [88]. However, the focus of the approach presented by Botella et al. [14] is not on patterns, but rather on the specification of security test objectives and how the test objectives may be used to extract test cases from a predefined model. In order to formalize security test objectives, Botella et al. [14] propose a test purpose language and show how this language can be used to formalize security test objectives using the tool CertifyIt. Security test cases are generated from a predefined model with respect to the formalized security test objectives. The approach is evaluated by applying it on two cryptographic components, one software component and one hardware component.

Marback et al. [93, 94] propose a threat-based security testing approach that automatically generates security test sequences from attack trees, and transforms the test sequences into executable tests. The approach consists of three main steps. The first step consists of modeling threat models in terms of threat trees. The second step consists of generating security test sequences from the threat trees. The third step consists of creating executable test cases from the test sequences by considering valid and invalid inputs. Threat trees are modeled by making use of the Microsoft Threat Modeling Tool [101, 102]. Security test sequences are generated by making use of a software implemented by the authors. The tool takes threat models as input and generates test sequences, adds input parameters to test sequences, generates test inputs

including valid and invalid inputs, and finally generates test scripts that execute all test sequences with assigned input values.

Hagerman et al. [60] propose an approach for security testing of aerospace launch systems. The approach is based on building a security test suite from behavioral models, attack types, and mitigation models. The goal of the approach is to generate a security test suite to provide a scalable model-based testing approach to test proper mitigation of security attacks. Behavioral and mitigation models may be modeled using UML sequence diagrams, UML activity diagrams, final state machines, and petri nets. Mitigation models describe mitigation patterns associated with an attack. Based on the mitigation patterns, mitigation test paths (in the model) are generated and then woven into the behavioral model. In other words, the mitigation models are used to identify certain attack paths, and these attack paths are then integrated with the model representing the system under test.

Thomas et al. [150] do not provide an approach to model-based security testing. However, they point out the lack of benchmarks that may be used to test and evaluate existing model-based testing approaches. To this end, Thomas et al. [150] present an approach to security mutation analysis which they apply on Magento, a fully-fledged open source e-commerce web application. In the approach, the authors create security mutants by injecting vulnerabilities in a systematic way. In particular, they consider the causes of vulnerabilities according to OWASP top 10 web application security risks [115], the application's business logic, as well as other consequences of vulnerabilities, such as STRIDE attacks [103]. In addition, using their benchmark, Thomas et al. [150] evaluate the approach provided by Marback et al. [93,94], and Xu [164].

## 4.4 Risk-driven testing

In Paper 1 we present the results from a systematic literature review addressing the combined use of risk analysis and testing. There are basically two main strategies for the combined use of risk analysis and testing: the use of risk analysis to support the testing process, and the use of testing to support the risk analysis process. We refer to the first strategy as risk-driven testing and to the second strategy as test-driven risk analysis. Since both strategies are strongly related, we present and discuss relevant approaches addressing both strategies.

In the systematic literature review, we identified a total of 32 papers. Some of these papers were written by the same authors and describe different aspects of the same approaches. By grouping the 32 papers based on first author, we obtain 24 approaches. In order to include relevant approaches published after the completion of our systematic literature review, we conducted a similar search process and found four additional approaches. This gives a total of 28 approaches. 25 out of the 28 approaches address risk-driven testing, while 3 out of the 28 approaches address test-driven risk analysis. Similar to Section 4.3, we categorize the approaches with respect to their main focus so that the approaches that are related are discussed close to each other. The approaches may overlap the various categories. The approaches are categorized into the following nine different categories.

- Approaches addressing the combination of risk analysis and testing at a general level.

- Approaches with main focus on model-based risk estimation.
- Approaches with main focus on test-case generation.
- Approaches with main focus on test-case analysis.
- Approaches based on automatic source code analysis.
- Approaches targeting specific programming paradigms.
- Approaches targeting specific applications.
- Approaches aiming at measurement in the sense that measurement is the main issue.
- Approaches with main focus on security, that is risk-driven security testing.

In the following, we present relevant approaches within the abovementioned categories. However, we discuss the approaches addressing risk-driven *security* testing in Section 4.5. The reader is referred to Paper 1 for detailed description of the review process, and discussion related to the research questions addressed in the literature review.

#### 4.4.1 Approaches addressing the combination of risk analysis and testing at a general level

The approaches provided by Amland [4], Felderer et al. [45, 47], Redmill [121, 122], and Yoon et al. [168] address risk-driven testing at a general level. The focus of both Amland [4] and Redmill [121, 122] is on the risk analysis part. Amland [4] achieves test case prioritization by expert meetings where probability indicators and failure costs are determined for each relevant function, and then combined into a risk exposure value. However, other elements, such as frequency of use, may also be taken into account. For Redmill [121, 122], test case prioritization may also be performed on the basis of either probability or consequence analysis alone.

In contrast, Yoon et al. [168] assume that the risk exposure value has been predetermined by a domain expert, and use mutation analysis to assess whether a test case covers a given fault or not. Each test case is then given a total weight based on the correlation with all the risks weighted with their individual risk exposure values.

Felderer et al. [45, 47] show a model-based approach to risk-driven testing, with the focus on product risks affecting the quality of the product itself. The approach includes a static risk assessment model using the factors of probability, impact, and time, each determined by several criteria. Each criterion is determined by a metric to be evaluated automatically (e.g., code complexity), semi-automatically (e.g., functional complexity), or manually (e.g., frequency of use and importance to the user).

Amland [4] includes a real case study on the application of risk-driven testing in a project within the financial domain. The case study indicates a positive effect in reduced time and resources, which also may increase quality as more time may be spent on the critical functions. Yoon et al. [168] present an experimental evaluation based on a real aircraft collision avoidance system, and find that their approach is more effective than using risk exposure values alone.

Both Amland [4] and Redmill [121, 122] emphasize the human and organizational factors. For risk-driven testing to reach its potential, there must exist a structured testing process, where everyone involved (e.g., programmers and test managers) understand how the risk analysis results should be used to direct the testing process. The risk-driven testing approach provided by Felderer et al. [47] is aligned with standard test processes, and the paper describes four stages for risk-driven test integration in a project. Challenges for integration at the initial stage in an anonymized industrial application are discussed, with a number of practical lessons learned. The approach itself is not compared to other approaches.

#### **4.4.2 Approaches with main focus on model-based risk estimation**

There are two approaches with main focus on model-based risk estimation, namely the approaches proposed by Gleirscher [54, 55] and Ray et al. [120]. Both approaches use state machine diagrams of the system behavior as a starting point. Gleirscher [54, 55] analyzes the state machine diagrams to identify hazards threatening the safety of the system. Also, a test model describing both the system and the environment is transformed into a Golog script, from which test cases can be derived.

Ray et al. [120] estimate reliability-based risks for individual scenarios as well as for the overall system by estimating the complexity for each state in each component and then using well-known hazard techniques for determining the severity of each scenario. Testing is not presented as part of the approach itself, but in the experimental validation using a library management case study. The approach is found to improve test efficiency by finding bugs responsible for severe failures and also finding more faults than two risk analysis approaches used for comparison.

#### **4.4.3 Approaches with main focus on test-case generation**

In the approach provided by Nazier et al. [109] and Wendland et al. [159], test cases are generated automatically from models with risk annotations. Test-case generation is also the focus of Kloos et al. [84] and Zimmermann et al. [174]. Kloos et al. [84], Nazier et al. [109], and Zimmermann et al. [174] focus on safety-critical systems, while Wendland et al. [159] emphasize the generality of their approach to cover critical situations in any kind of system.

For Wendland et al. [159], the starting point is a requirements specification formalized as an integrated behavior tree. The tree is then augmented with risk information combining likelihood and severity into risk levels. For each risk exposure, an appropriate test directive should be identified by experienced personnel, describing precisely the test derivation technique and strategy to be used for each risk level. The test themselves can then be generated automatically by the use of a tool.

In the approach provided by Nazier et al. [109], fault trees are used as system models, and state charts as system behavior models. Risk analysis information in the form of expected causes of failures are extracted from fault tree analysis and associated with system state chart elements. From this, a risk-driven test model is generated automatically, and verified to be correct, complete and consistent using a model checker. The model checking is also used to generate the risk-driven test cases.

Fault tree analysis is also used by Kloos et al. [84] to identify event sets in order of descending risk. A detailed algorithm is provided for using the event sets to transform the base model, a finite state automaton describing possible system situations, into a test model from which test cases are generated.

In the approach provided by Zimmermann et al. [174], the main idea is to use systematic construction or refinement of risk-driven test models so that only critical test cases can be generated. The approach uses state charts and model-based statistical testing with Markov chains test models to describe the simulation and system under test.

Tool support is important for both Nazier et al. [109] and Wendland et al. [159]. While Nazier et al. [109] has implemented a prototype tool, Wendland et al. [159] assert that all features discussed in the paper will be integrated in their existing modeling environment tool Fokus!MBT using UML profiles.

The approach by Kloos et al. [84] is not supported by a dedicated tool, but they make use of some existing tools for part of their evaluation. The approach is demonstrated on a modular production system, where it is found that the approach provides a significant increase in the coverage of safety functions.

No tool support is mentioned in the approach provided by Zimmermann et al. [174]. The approach is tested on a railway control system, and found to be successful in generating only critical test cases which represent high risk.

#### 4.4.4 Approaches with main focus on test-case analysis

The approaches proposed by Chen et al. [27, 28], Entin et al. [38], and Stallbaum et al. [145] focus on analyzing the generated test cases to select and prioritize the most critical ones based on risk. For Chen et al. [27, 28] and Stallbaum et al. [145], UML activity diagrams are used as the basic model from which test cases are generated, whereas Entin et al. [38] use state charts.

For each test case, Chen et al. [27, 28] start by estimating the cost based on the consequence of a fault for the customer or the vendor, and then deriving what the authors refer to as the severity probability for each test case based on the number and severity of defects. Finally, risk exposure (cost multiplied with probability of occurrence) is calculated for each test case, before those with the highest risk exposure are selected for safety tests.

Stallbaum et al. [145] use the test model first for generating unordered test case scenarios, and then for ordering these based on the risk information provided in the model. Risk prioritization may be based on the total risk score, that is, the sum of the risks of all actions covered by the test case scenario, or by only considering risks not already covered by previously chosen test case scenarios.

Markov chains are used for the test case generation in the approach provided by Entin et al. [38], which also uses the prioritization algorithm of Stallbaum et al. [145] to prioritize previously generated test suites.

Of the approaches outlined in this section, only the one by Stallbaum et al. [145] is supported by a dedicated prototype tool, but all have been tested on examples taken from real systems. Using a subset of IBM WebSphere Commerce 5.4, the approach provided by Chen et al. [27, 28] is found to be more effective and cover more critical test cases with a slightly better case coverage. A significant increase in coverage is reported by Entin et al. [38], where it is also mentioned that much effort was needed



to convince various interest groups (e.g., software developers and management) of the benefits of model-based testing. The approach provided by Stallbaum et al. [145] is evaluated on a real income tax calculation example, and the approach is found to enable early detection of critical faults.

#### 4.4.5 Approaches based on automatic source code analysis

A different kind of approach is provided by Hosseingholizadeh [64] and Wong et al. [162], which are based on automatic source code analysis. The approach by Wong et al. [162] is one of the few ones which focus on test-driven risk analysis. Risk of code is described as the likelihood that a given function or block within source code contains a fault. A fault within source code may lead to execution failures, such as abnormal behavior or incorrect output. The risk model is updated based on metrics related to both the static structure of code as well as dynamic test coverage. A more complex static structure leads to higher risk, while more thoroughly tested code has less risk.

The approach suggested by Hosseingholizadeh [64] is based on the approach provided by Wong et al. [162], but focuses on risk-driven testing aiming at test prioritization and optimization. More structural observations are added to the analysis process in the approach suggested by Wong et al. [162], such that, for example, errors in loop conditions result in a higher risk for the affected code block.

The approaches by both Hosseingholizadeh [64] and Wong et al. [162] are supported by prototype tools for automating the source code analysis. Even though the approach suggested by Wong et al. [162] has been tested on a real program, where the program faults were found to be located in the blocks and functions identified as high risk, the authors are careful to conclude with respect to the generalizability of the results.

#### 4.4.6 Approaches targeting specific programming paradigms

The risk-driven testing approach given by Kumar et al. [86] is targeted towards aspect-oriented programming (AOP), arguing that AOP solutions typically have an enterprise or platform level scope and, therefore, AOP errors have a much larger impact than errors in traditional localized code. The approach consists of a model for risk assessment, an associated fault model, and AOP testing patterns. The risk model is intended to be used for high-level business decisions to determine the role of AOP in the enterprise or in a particular project. Using the fault model and the test framework, test plans and test cases can be created to mitigate risk identified using the model. The framework supports both white-box and black-box testing, using unit and regression testing. The existing AOP testing patterns are targeted towards service-oriented architectures and enterprise architecture solutions.

A more low-level approach is provided by Rosenberg et al. [123], identifying the most important classes to test in an object-oriented program. The probability of failure of a portion of code (that is, a class) is determined by its complexity. For this, Rosenberg et al. [123] use six metrics identified by the Software Assurance Technology Center at NASA Goddard Space Flight Center. These metrics include both internal complexity (for example, number of methods in the class and minimum number of test cases needed for each method), the structural position of the class (for example, depth in inheritance tree and number of immediate subclasses), and relationship to other classes (for example, coupling and the number of methods that can be invoked from the



outside). For all these metrics, threshold values are defined, and a class is said to have high risk if at least two of the metrics exceed the recommended limits. If further prioritization is needed, it is up to the project to determine the criticality of each of the high-risk classes.

The approach provided by Kumar et al. [86] is said to have been used with reasonably large customers, but this is not described in the paper. Empirical assessment is not documented by Rosenberg et al. [123] either. Neither one of the two approaches seem to be supported by a dedicated tool.

#### 4.4.7 Approaches targeting specific applications

Bai et al. [8,9] show an approach for risk-driven group testing of semantic web services. Test cases are grouped according to their risk level, and the groups with highest risk are tested first. Services that do not meet a defined threshold are ruled out. In this way, a large number of unreliable web services are removed early in the testing process, reducing the total number of executed tests. In the risk analysis, failure probability and importance are analyzed from three aspects: ontology data, service, and composite service. In addition, a run-time monitoring mechanism is used to detect dynamic changes in the web services and adjust the risk accordingly. The approach is evaluated in an experiment using the BookFinder OWL ontology to compare risk-driven testing with random testing. The experiment concluded that the risk-driven testing approach greatly reduced the test cost and improved test efficiency.

Web services are also the focus in the framework of Casado et al. [23,24], where the aim is to test the advanced, long-lived transactions used in web services. The conceptual framework is hierarchically organized into four levels, consisting of (1) a method to define functional transactional requirements, (2) division into subsystems of transaction system properties, (3) applying risk analysis for predefined properties of web services transactions, and (4) applying testing techniques to generate test scenarios and mitigate risks. The most developed part of the framework is the use of fault tree analysis to perform risk analysis for the recovery property. This analysis is exemplified using a travel agency example, but no empirical evaluation is reported.

#### 4.4.8 Approaches aiming at measurement in the sense that measurement is the main issue

The approach suggested by Schneidewind [133] explicitly supports both risk-driven testing and test-driven risk analysis. With a focus on reliability, the paper describes a risk-driven reliability model and a testing process where the risk of software failure is used to drive test scenarios and reliability predictions. Both consumer and producer risks are considered. In addition to comparing empirical values of risk and reliability to specified threshold values, emphasis is placed on evaluating also the model that predicts risk and reliability.

The approach provided by Schneidewind [133] does not seem to be supported by a dedicated tool, but the approach is applied to a real application involving the NASA space shuttle flight software. For this application, all tests are passed with the conclusion that this safety critical software has been accepted. The predicted consumer reliability is compared to the actual reliability, and also to predicted reliability using

another approach (Yamada S Shaped Model), with favorable results for the approach suggested by Schneidewind [133].

The approach suggested by Souza et al. [141, 142] defines a set of risk-driven testing metrics to control and measure (i) the risk-driven test cases, (ii) the risk-driven testing activities, and (iii) the impact and advantages of using risk-driven testing in an organization. The risk-driven testing process, together with the proposed metrics, is tested in a real case study involving developers of an Eclipse-based tool environment for automatic development of simulators. The results are positive, concluding that for this application, the proposed approach was able to find the most important defects earlier than a more traditional functional approach to testing, thus saving costs. The case study also demonstrated the need for better tools, and the authors report to be working on such a tool, supporting in particular risk management which test engineers find more difficult to perform than testing.

## 4.5 Risk-driven security testing

Based on the literature review, we identified six approaches that focus on risk-driven security testing. These are the approaches suggested by Xu et al. [166], Murthy et al. [107], Zech et al. [171, 172], Botella et al. [15], Großmann et al. [57, 58], and Seehusen [135].

In the approach suggested by Xu et al. [166], formal threat models in the form of predicate/transition nets are used to generate the security tests. These are then converted into executable test code using a model-implementation mapping specification. The approach is not supported by a dedicated tool. However, the approach is applied in two case studies based on real-world systems: a web-based shopping system and a file server implementation. For both systems, multiple security risks were found, and the approach was also able to kill the majority of the security mutants injected deliberately.

The approach provided by Murthy et al. [107] focuses on how to introduce risk-driven testing principles into application security testing. The approach combines best practices from various areas like NIST and OWASP to model threat scenarios and test cases. The paper reports on a case study comparing traditional testing with risk-driven testing on a gaming application. Risk-driven testing was found to provide savings in time, cost, and resource usage.

Security testing is also the focus of Zech et al. [171, 172], targeting especially cloud computing environments. This is a model-based approach to risk-driven testing, emphasizing the negative perspective where the main goal is not to assure system validity but rather to show its deficiency. Model-to-model transformations are used from a system model to a risk model and then to a misuse-case model. In addition, testing is performed by run-time model execution, instead of the more traditional approach of generating executable test code from the models. No empirical evaluation is reported. The approach is supported by a dedicated tool, developed as an Eclipse plug-in.

The approaches suggested by Botella et al. [15], Großmann et al. [57, 58], and Seehusen [135] identify security risks by making use of the CORAS risk analysis language [91]. The risk models contain threat scenarios which are used in these approaches to identify high-level test procedures. The test procedures are in turn used to design concrete test cases. Seehusen [135] provides guidelines explaining how threat scenarios

in CORAS risk models may be used as a basis to identify high-level test procedures.

Botella et al. [15] and Großmann et al. [57,58] employ a similar strategy as suggested by Seehusen [135] in order to identify high level test procedures based on the threat scenarios represented in a CORAS risk model. Then, for each test procedure, they identify an associated test pattern. While Botella et al. [15] make use of UML class diagrams, object diagrams, and state machines to express the test model (instantiation of the test pattern), Großmann et al. [57,58] incorporate guidelines for how certain test cases should be modeled, as part of the test pattern instantiation (referred to as test design strategy).

The approach provided by Seehusen [135] is supported by a tool for the purpose of creating CORAS risk models. Botella et al. [15] make use of the tool provided by Seehusen [135] for the purpose of risk modeling, while they use the tool CertifyIt for test-case design and execution. The approach provided by Großmann et al. [57, 58] is supported by a tool framework which they use to create risk models (using the CORAS notation), as well as models representing test cases which are in turn used for test execution.



## Summary of contributions

This thesis provides three kinds of contributions. First, it provides a new artifact in terms of a risk analysis language and a method for risk-driven security testing. The risk analysis language and the method for risk-driven security testing are tightly integrated, and we refer to them collectively as the CORAL approach. Second, it provides empirical studies related to risk-driven security testing and test-driven security risk analysis, in terms of industrial case studies. Third, it provides an overview of state of the art approaches that combine risk analysis and testing, in terms of a systematic literature review.

The approaches included in the systematic literature review are approaches that combine risk analysis and testing in general. That is, in addition to approaches addressing security, we also reviewed approaches that address other qualities, such as safety and reliability. We did this in order to get a holistic picture of the domain. The insight obtained from the literature review was mainly used to support the development of the CORAL approach. However, the systematic literature review is also a contribution on its own because it may serve as a basis for examining various approaches that combine risk analysis and testing, and it may serve as a resource for identifying the appropriate approach to use.

The industrial case studies were also carried out to support the development of the CORAL approach. Based on problems identified in the systematic literature review, we developed possible solutions which we tried out in industrial case studies. In total, we carried out three case studies, and from each case study we gathered new information which eventually led to the creation and further improvement of the CORAL approach. However, the case studies are also contributions on their own because they provide insight, that are general in nature, about the combination of risk analysis and testing in practice.

The remainder of this chapter is organized as follows. Section 5.1 presents an overview of the CORAL approach, while Sections 5.2 and 5.3 present in more detail the risk analysis language and the method for risk-driven security testing, respectively. Section 5.4 presents an overview of the industrial case studies, and finally, Section 5.5 presents an overview of the systematic literature review.

## 5.1 The CORAL approach

The CORAL approach consists of a risk analysis language and a method for risk-driven security testing. The approach is specialized for security testers, and its purpose is to help security testers to systematically carry out risk-driven security testing. As illustrated in Figure 5.1, the CORAL risk analysis language lies at the core of the approach, and it is applied within the CORAL method for risk-driven security testing. The risk analysis language is a modeling language based on UML interactions [110]. There are two main reasons for this. First, UML interactions are among the top three modeling languages within the model-based testing community [34], and often used for testing purposes [10, 111, 153]. Second, in the risk analysis language, we represent risk-related information by annotating the constructs inherited from UML interactions with appropriate graphical icons. By annotating the constructs inherited from UML interactions with risk-related information, we bring risk analysis to the work bench of testers without the burden of a separate risk analysis language, thus reducing the effort needed to adopt the approach. The risk analysis language consists of the following three components: a graphical notation, an abstract syntax, and a natural-language semantics.

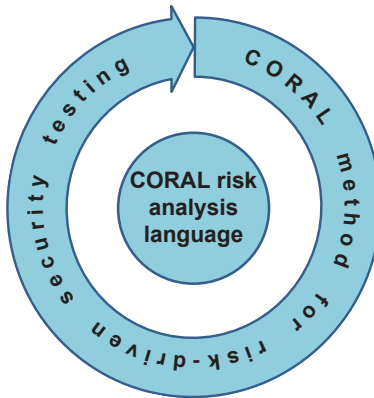


Figure 5.1: The overall relationship between the CORAL risk analysis language and the CORAL method for risk-driven security testing.

The method for risk-driven security testing consists of seven steps that are supported by the risk analysis language. The seven steps of the method are: test planning, threat scenario risk identification, threat scenario risk estimation, threat scenario risk evaluation, threat scenario test case design, test execution, and test incident reporting.

The graphical notation of the risk analysis language provides the necessary constructs for identifying, estimating, and evaluating security risks. The graphical notation is also used for designing security test cases. Moreover, the models representing the test cases are used for test execution, as well as for reporting test results.

The abstract syntax provides a set of rules, in terms of a context-free grammar, that defines the correct combinations of the constructs in the CORAL risk analysis language. The syntax is useful for modeling interactions that are syntactically correct in the CORAL language.

The natural-language semantics provides a set of rules for schematically translating threat scenarios modeled using the CORAL language into English prose. Testers may use the natural-language semantics to clearly and consistently document, communicate and analyze security risks.

## 5.2 The CORAL risk analysis language

In this section we give a more detailed explanation of the CORAL risk analysis language with respect to its graphical notation, abstract syntax, and natural-language semantics.

### 5.2.1 Graphical notation

The graphical notation of the CORAL language is mainly based on the graphical notation of UML interactions [110]. However, the graphical icons that are used to represent risk-related information in the CORAL language are based on corresponding graphical icons in the CORAS risk analysis language [91]. This is a deliberate design decision because the graphical icons in CORAS are empirically shown to be cognitively effective [139].

The various constructs in the CORAL language can be grouped into five categories: diagram frame, lifelines, messages, risk-measure annotations, and interaction operators.

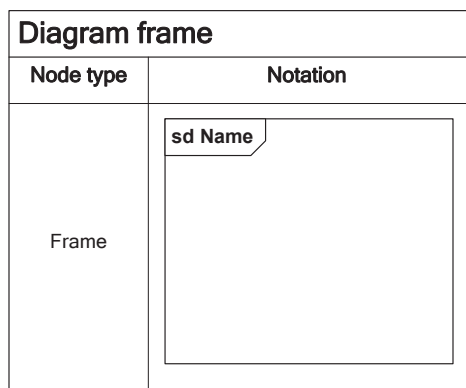


Figure 5.2: Graphical notation for the diagram frame.

**Diagram frame.** The diagram frame is the frame in which an interaction is modeled. An interaction may represent the system under test, its environment, as well as threat scenarios that the system under test and its environment is exposed to. A threat scenario includes the threat causing the threat scenario, the aspect of the system under test exposed to the threat scenario, the unwanted incident caused by the threat scenario, and the security asset harmed by the unwanted incident. Figure 5.2 illustrates the graphical notation for the diagram frame. The diagram frame is graphically equivalent to the diagram frame in UML interactions [110, p. 516]. However, there is one main difference between the diagram frame used in the CORAL language and the diagram frame used in UML interactions. In UML interactions, different kinds of interaction

diagrams may be represented within the diagram frame, such as sequence diagrams, communication diagrams, interaction overview diagrams, and timing diagrams. In the CORAL approach, only interaction sequence diagrams constructed using the CORAL language may be represented. No other constructs may be used.

Similar to UML interactions, the keyword **sd** is used to denote that the diagram is a sequence diagram. The keyword **sd** is followed by the name of the diagram, that is, an abstract description of the interaction modeled within the diagram frame. The keyword and the name of the diagram is placed in a pentagon in the upper left corner of the diagram frame, as illustrated in Figure 5.2.

**Lifelines.** According to UML, a lifeline represents an individual participant in an interaction [110, p. 504]. As illustrated in Figure 5.3, we distinguish between five different lifelines: general lifeline, deliberate threat lifeline, accidental threat lifeline, non-human threat lifeline, and asset lifeline.

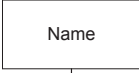




Lifelines	
Node type	Notation
General lifeline	
Deliberate threat lifeline	
Accidental threat lifeline	
Non-human threat lifeline	
Asset lifeline	

Figure 5.3: Graphical notation for lifelines.

The general lifeline is graphically equivalent to a lifeline in UML interactions. In the CORAL language, a general lifeline is used to model the system under test, as well as the environment interacting with the system under test. The name of the lifeline is placed inside the rectangle of the lifeline as illustrated in Figure 5.3. The naming convention of a general lifeline is equivalent to the naming convention of lifelines in UML interactions [110, p. 505].



The lifelines representing threats are used to model threats that may initiate threat scenarios, which in turn may cause security risks in the system under test. Inspired by CORAS [91], we distinguish between three kinds of threats: deliberate threat, accidental threat, and non-human threat. A deliberate threat is a human threat that has malicious intents. For example, a hacker, or a malicious employee in a company or an organization (insider). An accidental threat is also a human threat, but this threat is different in the sense that it does not have malicious intents. For example, a system administrator with low security knowledge may carry out an action, with good intentions, but still cause a security risk without realizing it. The non-human threat is a threat that may be anything else except a human. For example, a power failure in a server hall may cause problems with respect to the availability of a system. Another example is a security bug in a source code that compromises the integrity of certain data when executed. In practice, the distinction between a human threat and a non-human threat is sometimes not straight forward. For example, if a hacker exploits a security bug in a source code in order to attack a system, then the threat is the hacker. On the other hand, if the security bug lies dormant in the source code and is triggered at some point during system execution, then the threat is the bug in the source code, that is, a non-human threat. In other words, the distinction between a human threat and a non-human threat depends on the viewpoint from which a threat is regarded. The name of a threat is placed below the icon representing the threat as illustrated in Figure 5.3. The name of a threat typically represents a threat profile which is described by the tester. A threat may, for example, be named “hacker” (deliberate threat), “database administrator” (accidental threat), or “computer virus” (non-human threat).

During the steps for threat scenario risk identification, estimation, and evaluation, we assess security risks that may harm certain security assets we want to protect. That is, the security risk assessment is carried out with respect to certain security assets that are regarded important and that need protection. In the CORAL language, we use the asset lifeline to model and represent a security asset. The name of a security asset is placed below the moneybag icon representing the asset as illustrated in Figure 5.3. Examples of security assets are “availability of customer data” and “integrity of bank transactions”. What is meant by “customer data” and “bank transactions” has to be described by the tester.

**Messages.** According to UML, a message defines a particular communication between lifelines of an interaction [110, p. 505]. UML interactions distinguish between complete, lost and found messages. Complete messages have both a sender and a receiver lifeline. A lost message has a sender lifeline, but not a receiver lifeline. A found message has a receiver lifeline, but not a sender lifeline. The graphical notation for these messages are different. However, lost and found messages are unnecessary in most interaction models and are used in rare situations [127].

Furthermore, UML interactions categorize complete messages into synchronous and asynchronous messages. The synchronous and asynchronous messages have different graphical notations. A synchronous message is used to call an operation, and the lifeline transmitting a synchronous message always expects a responding message. An asynchronous message, on the other hand, is used to send a signal which may or may not be responded. Synchronous messages are therefore syntactically more strict than asynchronous messages because they require a corresponding response message for each operation call. However, at a logical level, sending a signal and calling an

operation are similar. Both types of messages involve a communication from a sender to a receiver [127].

In the CORAL language, we are interested in expressing complete interactions between two lifelines, in terms of a communication from a sender to a receiver. We therefore make use of complete messages in the CORAL language. Moreover, because synchronous and asynchronous messages are similar at a logical level, it is not necessary to express both in the CORAL language. For this reason, we choose to treat all messages in the CORAL language as asynchronous messages. The graphical notation for messages in the CORAL language are therefore based on the graphical notation for the asynchronous message in UML interactions [110, p. 518]. As illustrated in Figure 5.4, we distinguish between five messages in the CORAL language: general message, new message, altered message, deleted message, and unwanted incident message.






Messages	
Node type	Notation
General message	
New message	
Altered message	
Deleted message	
Unwanted incident message	

Figure 5.4: Graphical notation for messages.

The general message is graphically equivalent to the asynchronous message in UML interactions [110, p. 518], and it is used to model the expected behavior between lifelines representing the system under test and the environment interacting with the system under test, that is, the interaction between general lifelines. Recall that general lifelines are used to model the system under test and its environment. The signature of a message in the CORAL language, that is, the content of a message, is placed above the arrow representing the message, as illustrated in Figure 5.4. Signatures are written using the same convention as given for messages in UML interactions [110, p. 507]. In addition, we represent the risk related information, in the signatures, using a red-colored, bold, and italic font to distinguish between the expected behavior and the risk-related information.

The messages representing a new, an altered, a deleted and an unwanted incident message are used in combination to represent threat scenarios. A new message is a message that is initiated by a threat. This may be a deliberate human threat, an accidental human threat, or a non-human threat. A new message is represented by a red triangle which is placed at the transmitting end of the message. An altered message is a message in the system under test that has been altered by a threat to deviate from its expected behavior. Altered messages are represented by a triangle with red borders and white fill. A deleted message is a message in the system under test that has been deleted by a threat. Deleted messages are represented by a triangle with red

borders and a red cross in the middle of the triangle. Finally, an unwanted incident is a message modeling that an asset is harmed or its value is reduced. Unwanted incidents are represented by a yellow explosion sign.

**Risk-measure annotations.** The risk-measure annotations are used to annotate messages for the purpose of estimating and evaluating security risks. As illustrated in Figure 5.5, we distinguish between three kinds of risk-measure annotations: frequency, conditional ratio, and consequence.

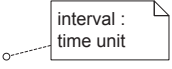

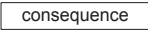
Risk-measure annotations	
Node type	Notation
Frequency	
Conditional ratio	
Consequence	

Figure 5.5: Graphical notation for risk-measure annotations.

The frequency annotation represents either the frequency of the transmission or the frequency of the reception of a message. The graphical notation of a frequency annotation is equivalent to the graphical notation of a comment generally used in UML [110, p. 56]. The connector on the frequency annotation is attached on either the transmission end or the reception end of a general message, new message, or an altered message. It may also be attached on the transmission end of an unwanted incident message. In Section 5.2.2 we define the syntax of the CORAL language, and we explain why the frequency annotation cannot be attached on a deleted message, or on the reception end of an unwanted incident message. The frequency is written inside the comment frame, in terms of an interval followed by a time unit, as illustrated in Figure 5.5.

The conditional ratio annotation represents the ratio by which a message is received, given that it is transmitted. The conditional ratio annotation may be attached on a general message, a new message, or an altered message. The conditional ratio may not be attached on a deleted message and an unwanted incident message. The reason for this is explained in Section 5.2.2. The conditional ratio annotation is attached below the mid-center of the message.

The consequence annotation represents the consequence an unwanted incident has on an asset. Recall that an unwanted incident is represented by an unwanted incident message. The consequence annotation may only be attached on unwanted incident messages. The reason for this is explained in Section 5.2.2. Similar to the conditional ratio annotation, the consequence annotation is also attached below the mid-center of the message.

**Interaction operators.** In UML interactions, messages may be combined in rectangles containing special keywords in order to convey a particular relationship between the combined messages. The rectangle encapsulating the messages is referred to as a combined fragment, while the keyword is referred to as an interaction operator. An interaction operator specifies the operation that defines the semantics of the combination of messages [110, p. 482]. As illustrated in Figure 5.6, the CORAL language makes use of four interaction operators inherited from UML interactions: potential alternatives (keyword **alt**), referred interaction (keyword **ref**), parallel execution (keyword **par**), and loop (keyword **loop**). All interaction diagrams are by default encapsulated within an implicit combined fragment that makes use of an interaction operator named weak sequencing (keyword **seq**) [110, p. 482]. The **seq** operator is the implicit composition mechanism of interactions. However, because the **seq** operator is always implicitly included in all interaction models, it is generally not modeled explicitly. We therefore describe the graphical notation of the aforementioned interaction operators.

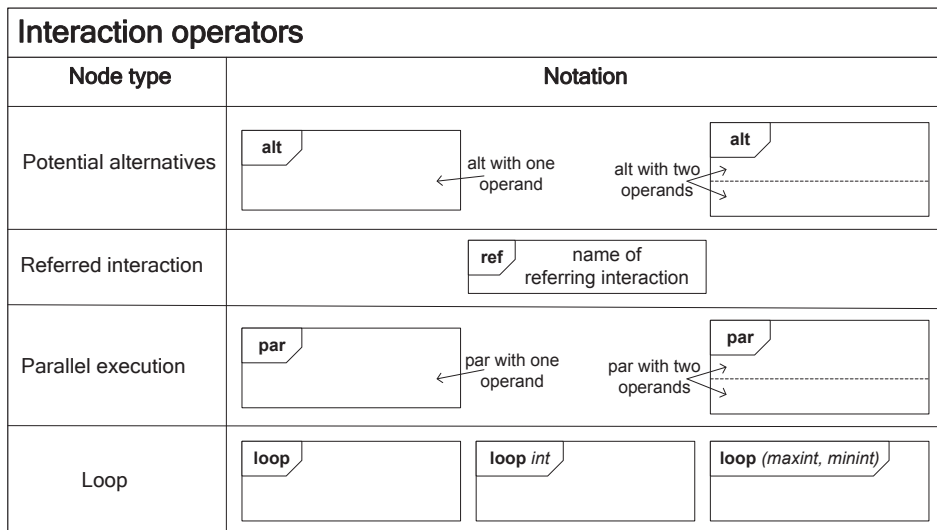


Figure 5.6: Graphical notation for interaction operators.

Depending on the keyword, the combined fragment may have one or more compartments in which messages may be modeled. The compartments are referred to as interaction operands. If the keywords **alt** or **par** are used, then the combined fragment may consist of one or more operands. If the keywords **ref** or **loop** are used, then the combined fragment may only consist of one operand. However, the operand in **ref** does not contain messages, but rather text describing the name of the interaction that is referred to, as illustrated in Figure 5.6.

According to UML, the interaction operator potential alternatives (**alt**) designates that the operands represent a choice of behavior [110, p. 482]. The UML standard requires that the chosen operand must have an explicit or implicit guard expression that evaluates to true. An implicit true guard is implied if the operand has no explicit guard. In the CORAL language, we currently allow only the usage of implicit true guards. We therefore do not model guards when using the interaction operator

potential alternatives.

According to UML, the interaction operator parallel execution (*par*) designates a parallel merge between the behaviors of the operands. A parallel merge defines a set of traces that describe all the ways that events of the operands may be interleaved without obstructing the order of the events within the operands [110, p. 483].

According to UML, the interaction operator loop designates that the operand represents a loop. The keyword *loop*, which is placed in a pentagon in the upper left corner of the diagram frame (see Figure 5.6), may be followed by no integers, one integer, or a pair of a maximum and a minimum integer. If the pentagon only consists of the keyword *loop*, then the operand represents a loop with zero as lower bound and infinity as upper bound. If *loop* is accompanied by an integer *int*, the operand represents a loop that loops exactly *int* times. Finally, if *loop* is accompanied by two integers, *minint* and *maxint*, the operand represents a loop that loops minimum *minint* times and maximum *maxint* times [110, pp. 485-486].

According to UML, an interaction use (*ref*) refers to an interaction. The interaction use is shorthand for copying the contents of the referred interaction where the interaction use is. To be accurate, the copying must take into account substituting parameters with arguments and connect the formal gates with the actual ones [110, p. 501].

## 5.2.2 Abstract syntax

The syntax of the CORAL risk analysis language is defined in Extended Backus-Naur Form [69]. The syntax defines a set of rules which explain how we may combine the constructs in the CORAL language in order to model syntactically correct interactions.

Throughout the definition of the syntax, we use different fonts to distinguish between the non-terminals and the terminals. Non-terminals are written in font *math mode*, while terminals are written in font **Sans Serif**. The terminals written in font **Bold Sans Serif** represent the type of a syntactical element. Furthermore, we make use of eight undefined terms in the grammar; *identifier*, *asset lifeline*, *int*, *minint*, *maxint*, *exact*, *interval*, and *time unit*. The reader is referred to Appendix A in Paper 3 for the explanation of these terms. We present the syntax by grouping the syntactical elements that are closely related; risk interaction, messages, lifelines, risk-measure annotations, and interaction operators.

**Risk interaction.** The term *risk interaction* is a collective term for the various constructs in the CORAL language.

$$\begin{aligned} \textit{risk interaction} = & \textit{message} \mid \textit{weak sequencing} \mid \textit{potential alternatives} \\ & \mid \textit{referred interaction} \mid \textit{parallel execution} \mid \textit{loop}; \end{aligned}$$

**Messages.** In the following, we define the syntax of the five different kinds of messages in the CORAL language: general, new, altered, deleted, and unwanted incident messages. The collective term for general, new and altered messages is *risky message*, the term for a deleted message is *deleted message*, and the term for an unwanted incident message is *unwanted incident message*.

$message = risky\ message \mid unwanted\ incident\ message$   
 $\mid deleted\ message;$   
 $risky\ message = \mathbf{rm}(identifier, transmitter\ lifeline, receiver\ lifeline,$   
 $risky\ message\ category, transmission\ frequency,$   
 $conditional\ ratio, reception\ frequency);$   
 $unwanted\ incident\ message = \mathbf{um}(identifier, transmitter\ lifeline, asset\ lifeline,$   
 $transmission\ frequency, consequence);$   
 $deleted\ message = \mathbf{dm}(identifier, transmitter\ lifeline,$   
 $receiver\ lifeline);$   
 $risky\ message\ category = \mathbf{general} \mid \mathbf{new} \mid \mathbf{alter};$

In Section 5.2.1, we highlighted three constraints with respect to risk-measure annotations on messages. As reflected in the syntax, the constraints are the following.

1. Frequency annotations may not be attached on a deleted message, nor may it be attached on the reception end of an unwanted incident message.
2. The conditional ratio may not be attached on a deleted message or an unwanted incident message.
3. The consequence annotation may only be attached on unwanted incident messages.

The reason to the first constraint is that a deleted message in the CORAL language represents the complete deletion of a message. That is, if a message is deleted, then it is not transmitted and therefore not received. It therefore does not make sense to estimate how often a message is *not* received, given that it is *not* transmitted. A message is either deleted, or it is not. Also, in the context of testing, we are interested in testing the messages that may *cause* the deletion of other messages.

We see from the syntax that the lifeline receiving an unwanted incident message is an *asset lifeline*. The asset lifeline is modeled in an interaction to represent the asset that is harmed by an unwanted incident. The frequency attached on the transmission end of an unwanted incident conveys how often an unwanted incident harms a certain security asset. This information needs to be represented only once. It is therefore not necessary to represent the same information by attaching a frequency on the reception end of an unwanted incident.

With respect to the second constraint, a conditional ratio may not be attached on a deleted message because the deleted message represents a complete deletion, as described above. A conditional ratio is the ratio by which a message is received, given that it is transmitted. Thus, it may only be attached on general, new, and altered messages.

With respect to the third constraint, as mentioned above, an unwanted incident message represents an unwanted incident that harms a security asset. In order to represent the impact of an unwanted incident, that is, to what degree the unwanted incident harms a security asset, we annotate the unwanted incident message with a consequence value. Thus, consequence values may not be attached on other messages than unwanted incident messages.

**Lifelines.** In the following, we define the syntax of the lifelines in the CORAL language. The lifeline transmitting a message and the lifeline receiving a message may be the same kind of lifeline. It may also be identical, that is, a lifeline may send a message to itself.

$$\begin{aligned}
 \text{transmitter lifeline} &= \text{general lifeline} \mid \text{deliberate threat lifeline} \\
 &\quad \mid \text{accidental threat lifeline} \\
 &\quad \mid \text{non-human threat lifeline}; \\
 \text{receiver lifeline} &= \text{general lifeline} \mid \text{deliberate threat lifeline} \\
 &\quad \mid \text{accidental threat lifeline} \\
 &\quad \mid \text{non-human threat lifeline}; \\
 \text{general lifeline} &= \mathbf{gl}(\text{identifier}); \\
 \text{deliberate threat lifeline} &= \mathbf{dtl}(\text{identifier}); \\
 \text{accidental threat lifeline} &= \mathbf{atl}(\text{identifier}); \\
 \text{non-human threat lifeline} &= \mathbf{ntl}(\text{identifier});
 \end{aligned}$$

**Risk-measure annotations.** In the following, we define the syntax of the risk-measure annotations in the CORAL language. The term *exact* refers to an exact value, while the term *interval* refers to an interval. The reader is referred to Appendix A in Paper 3 for examples and further explanation.

$$\begin{aligned}
 \text{transmission frequency} &= \text{frequency}; \\
 \text{reception frequency} &= \text{frequency}; \\
 \text{frequency} &= \mathbf{f}(\text{exact}, \text{time unit}) \mid \mathbf{f}(\text{interval}, \text{time unit}); \\
 \text{conditional ratio} &= \mathbf{cr}(\text{exact}) \mid \mathbf{cr}(\text{interval}); \\
 \text{consequence} &= \mathbf{c}(\text{identifier});
 \end{aligned}$$

**Interaction operators.** In the following, we define the syntax of the interaction operators in the CORAL language. In Extended Backus-Naur Form, “{ }<sup>-</sup>” means an ordered sequence of one or more repetitions of the enclosed element [69]. This means that the interaction operators **seq**, **alt** and **par** may consist of an ordered sequence of one or more risk interactions. The term *risk interaction* is defined above.

$$\begin{aligned}
 \text{weak sequencing} &= \mathbf{seq}(\{\text{risk interaction}\}^-); \\
 \text{potential alternatives} &= \mathbf{alt}(\{\text{risk interaction}\}^-); \\
 \text{referred interaction} &= \mathbf{ref}(\text{identifier}); \\
 \text{parallel execution} &= \mathbf{par}(\{\text{risk interaction}\}^-); \\
 \text{loop} &= \mathbf{loop}(\text{risk interaction}) \mid \mathbf{loop}(\text{int}, \text{risk interaction}) \\
 &\quad \mid \mathbf{loop}((\text{minint}, \text{maxint}), \text{risk interaction});
 \end{aligned}$$

### 5.2.3 Natural-language semantics

Situations may arise where the information conveyed by interactions modeled using the CORAL language are interpreted differently by different testers. Thus, in order to help software testers to clearly and consistently document, communicate and analyze risks,

we define a structured approach to generate the semantics of interactions constructed by the CORAL language in terms of English prose.

For each terminal representing the **type** of a syntactical element defined in Section 5.2.2, there is an associated English-prose semantics defined in this section. The English-prose semantics is defined by a function  $\llbracket \cdot \rrbracket$  that takes a syntactical element as input, and provides English prose of the syntactical element. Similar to Section 5.2.2, we group the syntactical elements that are closely related.

**Messages.** In the following, we define the English-prose semantics for the messages in the CORAL language. Let the syntactical variables

- $id$  range over *identifier*
- $t$  range over *transmitter lifeline*
- $r$  range over *receiver lifeline*
- $al$  range over *asset lifeline*
- $f$  range over *frequency*
- $cr$  range over *conditional ratio*
- $c$  range over *consequence*

$\llbracket \mathbf{rm}(id, t, r, \mathbf{general}, f_1, cr, f_2) \rrbracket$  = The message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  $\llbracket r \rrbracket$   $\llbracket f_1 \rrbracket$ , the transmission leads to its reception  $\llbracket cr \rrbracket$ , and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \mathbf{rm}(id, t, r, \mathbf{new}, f_1, cr, f_2) \rrbracket$  = The new message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  $\llbracket r \rrbracket$   $\llbracket f_1 \rrbracket$ , the transmission leads to its reception  $\llbracket cr \rrbracket$ , and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \mathbf{rm}(id, t, r, \mathbf{alter}, f_1, cr, f_2) \rrbracket$  = The altered message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  $\llbracket r \rrbracket$   $\llbracket f_1 \rrbracket$ , the transmission leads to its reception  $\llbracket cr \rrbracket$ , and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \mathbf{uim}(id, t, al, f, c) \rrbracket$  = The unwanted incident  $id$  occurs on  $\llbracket t \rrbracket$   $\llbracket f \rrbracket$ , and impacts asset  $al$   $\llbracket c \rrbracket$ .

$\llbracket \mathbf{dm}(id, t, r) \rrbracket$  = The message  $id$  transmitted from  $\llbracket t \rrbracket$  to  $\llbracket r \rrbracket$  is deleted.

**Lifelines.** In the following, we define the English-prose semantics for the lifelines in the CORAL language. Let the syntactical variable

- $id$  range over *identifier*



$$\llbracket \mathbf{gl}(id) \rrbracket = id$$

$$\llbracket \mathbf{dtl}(id) \rrbracket = \text{the deliberate threat } id$$

$$\llbracket \mathbf{atl}(id) \rrbracket = \text{the accidental threat } id$$

$$\llbracket \mathbf{ntl}(id) \rrbracket = \text{the non-human threat } id$$

**Risk-measure annotations.** In the following, we define the English-prose semantics for the risk-measure annotations in the CORAL language. Let the syntactical variables

- $id$  range over *identifier*
- $e$  range over *exact*
- $i$  range over *interval*
- $tu$  range over *time unit*

Undefined values are represented by  $\perp$ .

$$\llbracket \mathbf{f}(e, tu) \rrbracket = \text{with frequency } e \text{ per } tu$$

$$\llbracket \mathbf{f}(i, tu) \rrbracket = \text{with frequency interval } i \text{ per } tu$$

$$\llbracket \mathbf{f}(\perp, \perp) \rrbracket = \text{with undefined frequency}$$

$$\llbracket \mathbf{cr}(e) \rrbracket = \text{with conditional ratio } e$$

$$\llbracket \mathbf{cr}(i) \rrbracket = \text{with conditional ratio interval } i$$

$$\llbracket \mathbf{cr}(\perp) \rrbracket = \text{with undefined conditional ratio}$$

$$\llbracket \mathbf{c}(id) \rrbracket = \text{with consequence } id$$

$$\llbracket \mathbf{c}(\perp) \rrbracket = \text{with undefined consequence}$$

**Interaction operators.** In the following, we define the English-prose semantics for the interaction operators in the CORAL language. Let the syntactical variables

- $d$  range over *risk interaction*
- $id$  range over *identifier*
- $x$  range over *int*
- $a$  range over *minint*
- $b$  range over *maxint*

The pair of square brackets, ‘[’ and ‘]’, is part of the resulting English-prose semantics and it is used to enclose an operand.

$\llbracket \text{seq}(d_1, d_2, \dots, d_m) \rrbracket = [ \llbracket d_1 \rrbracket ]$  weakly sequenced by  $[ \llbracket d_2 \rrbracket ]$  weakly sequenced by ...  
weakly sequenced by  $[ \llbracket d_m \rrbracket ]$

$\llbracket \text{alt}(d_1, d_2, \dots, d_m) \rrbracket =$  either  $[ \llbracket d_1 \rrbracket ]$  or  $[ \llbracket d_2 \rrbracket ]$  or ... or  $[ \llbracket d_m \rrbracket ]$

$\llbracket \text{ref}(id) \rrbracket =$  Refer to interaction:  $id$ .

$\llbracket \text{par}(d_1, d_2, \dots, d_m) \rrbracket = [ \llbracket d_1 \rrbracket ]$  parallelly merged with  $[ \llbracket d_2 \rrbracket ]$  parallelly merged with ...  
parallelly merged with  $[ \llbracket d_m \rrbracket ]$

$\llbracket \text{loop}(d) \rrbracket =$  loop minimum zero times and maximum infinitely  $[ \llbracket d \rrbracket ]$

$\llbracket \text{loop}(x, d) \rrbracket =$  loop exactly  $x$  times  $[ \llbracket d \rrbracket ]$

$\llbracket \text{loop}(a, b, d) \rrbracket =$  loop minimum  $a$  times and maximum  $b$  times  $[ \llbracket d \rrbracket ]$

### 5.3 The CORAL method for risk-driven security testing

As illustrated in Figure 5.7, the CORAL method for risk driven security testing consists of seven steps. We also see from the figure that the CORAL risk analysis language lies at the core of the method. In order to initiate the process for risk-driven security testing, the tester needs to gather the necessary description of the system under test. This is used as input to the first step. The description of the system under test may be in form of system diagrams, use case documentation, system manuals, source code, executable versions of the system, and so on.

Based on the system description given as input to **Step 1**, we plan the risk-driven security testing process, prepare the model of the system under test, identify security assets we need to protect, define frequency scales, define consequence scales, and construct the risk evaluation matrix based on the frequency and consequence scales. The system under test is modeled as UML interactions by making use of the CORAL language.

In **Step 2**, we take as input the model of the system under test, as well as the security assets identified in Step 1. Based on these inputs, we first identify security risks by analyzing the models with respect to the security assets. Security risks are represented by unwanted incident messages. Then we identify threat scenarios that may cause the security risks. The output of this step is a set of threat scenarios that the system under test is exposed to.

In **Step 3**, we take the threat scenarios identified in Step 2 as input. In addition, we make use of the frequency and consequence scales defined in Step 1. Based on these inputs, we estimate the frequency of the messages causing security risks. As part of the frequency estimation, we also estimate the conditional ratio between the transmission of a message and the reception of a message. Based on these estimates, we calculate the frequency of security risks. Moreover, we estimate the consequence of security risks in terms of impact on security assets. The estimates are modeled by using the constructs related to risk-measure annotations in the CORAL language. The output of this step is the set of threat scenarios given as input to the step, updated with risk-measure annotations.

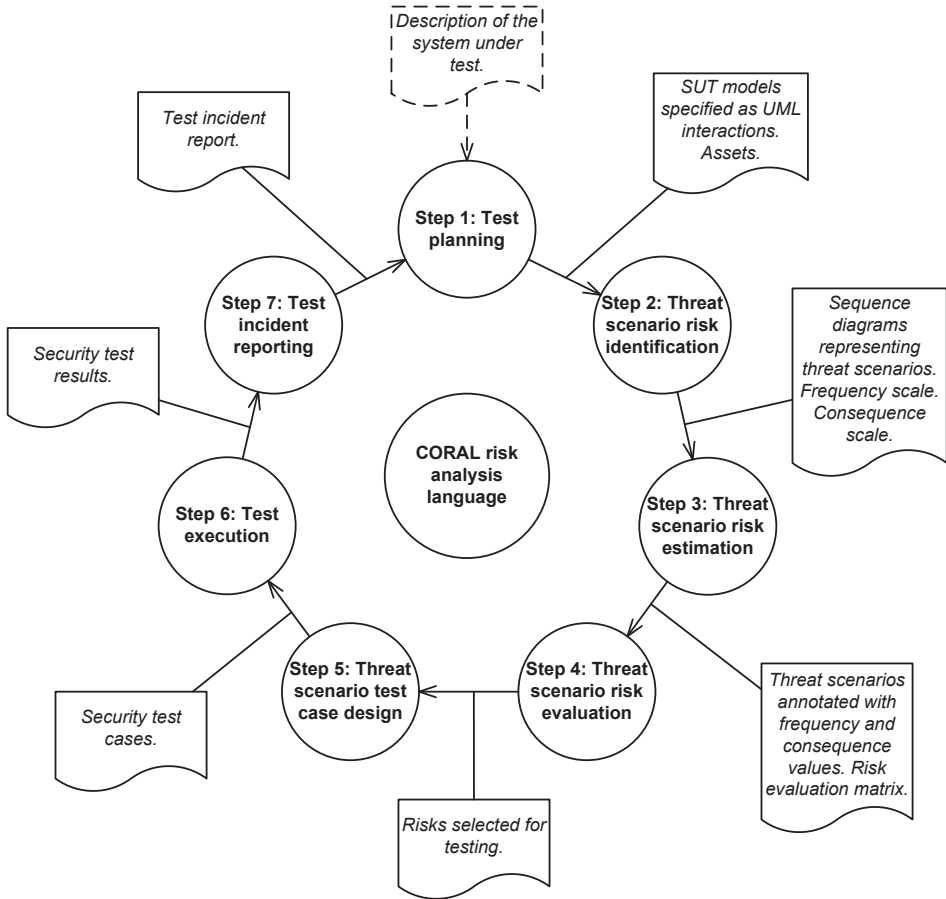


Figure 5.7: The CORAL method for risk-driven security testing.

In **Step 4**, we take the estimated threat scenarios as input. Based on this input, we evaluate the risks caused by the threat scenarios. First, we plot the risks into the risk evaluation matrix with respect to their estimated frequency and consequence values. Second, we specify suspension criteria, which are used as a basis for selecting the most severe security risks that need to be tested. According to the software testing standard ISO/IEC/IEEE 29119 [73], suspension criteria are used to stop all or a portion of the testing activities. Third, we aggregate risks, that are similar in nature, in order to evaluate whether their risk level should be increased. If the risk level is increased, we assess whether they should be included in the testing. Finally, based on the suspension criteria, we select the risks to test, and exclude the risks that are not regarded as severe. The output of this step is a set of risks selected for testing.

In **Step 5**, we design security test cases with respect to the set of risks selected for testing. First, for each risk selected for testing, we refer to the threat scenario in which the risk occurs. Second, for each threat scenario, we specify a test objective. Third, for each test objective, we select the interaction fulfilling the test objective by annotating

the threat scenario with stereotypes from the UML Testing Profile [111]. The output of this step is a set of security test cases.

In **Step 6**, we carry out security testing with respect to the test cases designed in Step 5. The test cases may be executed manually, semi automatically, or automatically. This depends on whether the test case is implementable in a tool, for example as an executable model, or whether the interaction conveyed by the test case must be carried out manually.

In **Step 7**, we write a test incident report. First, we analyze the test results and confirm test incidents. A test incident occurs if a security test case fails [74, p. 38]. Second, we document each test incident. For each test case that fails, we use its model as a basis for documenting the test incident. The result is a test incident report. This report is then communicated to the relevant stakeholder. The test incident report may be used as input for a new run of the CORAL method. It may also be used as a basis for re-testing after the test incidents have been treated.

In the following, we present an example-driven explanation of the CORAL method. The example is a small fragment from an industrial case study which is reported in Paper 4. The system under test, in the example, is a feature in a web-based e-business application designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans. The feature is named Exercise Options and it is used for buying shares in a company.

### 5.3.1 Test planning (Step 1)

In the case study, we modeled the system under test by analyzing the source code (white-box model), and observing the behavior of the system under test by executing it on a test environment (black-box model). In the following, we will only focus on the black-box model of Exercise Options (see Figure 5.8). A user may exercise options by typing in the number of options to exercise and then submitting the request (*exercise(options)*). Then the web application responds with a request for selecting the method, that is, the attorney, by which the options are to be exercised (*selectExerciseMethod*). The user selects the method for exercising options by checking the appropriate attorney and then pressing continue (*continue(exerciseMethod)*). Upon successful method selection, the web application responds with a confirmation stating that the options have been exercised (*exerciseRequestConfirmation*).

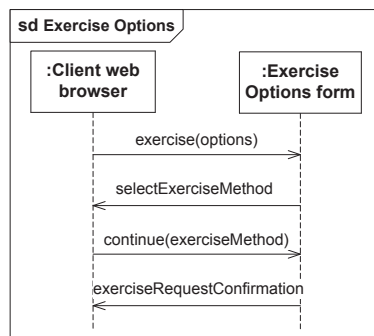


Figure 5.8: Black-box model of feature Exercise Options.

Having modeled the system under test, we identified a number of security assets. The asset identification was conducted together with the customer. One of the security assets identified in the case study was *integrity of data* for the feature Exercise Options. As part of Step 1, we also defined a likelihood scale in terms of frequencies, and a consequence scale. Table 5.1 shows an example of a frequency scale, while Table 5.2 shows an example of a consequence scale for security asset *integrity of data*. These scales are used in Step 3 for estimating security risks. Figure 5.9 shows the risk evaluation matrix constructed with respect to the frequency scale and consequence scale. The risk matrix is used in Step 4 to evaluate security risks. Risks are grouped in nine levels horizontally on the matrix where Risk Level 1 is the lowest risk level and Risk Level 9 is the highest risk level. The risk level of a risk is identified by mapping the underlying color to the column on the left-hand side of the matrix.

Table 5.1: An example frequency scale.

Likelihood	Definition	Frequency interval
Certain	100 times or more per year	$[100, \infty):1y$
Likely	From and including 25 to less than 100 times per year	$[25, 100):1y$
Possible	From and including 10 to less than 25 times per year	$[10, 25):1y$
Unlikely	From and including 1 to less than 10 times per year	$[1, 10):1y$
Rare	From and including 0 to less than 1 times per year	$[0, 1):1y$

Table 5.2: An example consequence scale for security asset *integrity of data*.

Consequence	Definition
Catastrophic	The integrity of the number of options being exercised is compromised
Major	The integrity of user data is compromised
Moderate	The integrity of terms and conditions is compromised
Minor	The integrity of log data is compromised
Insignificant	The integrity of text in the graphical user interface is compromised

### 5.3.2 Threat scenario risk identification (Step 2)

Let us assume that a malicious user attempts to access another system feature, say an administrative functionality, by altering certain parameters in the HTTP request sent to Exercise Options. The malicious user could achieve this, for example, by first intercepting the request containing the message `continue(exerciseMethod)` using a network proxy tool such as OWASP ZAP [116], and then altering the parameter `exerciseMethod` in the message. This alteration, could in turn give the malicious user access to another system feature. This unwanted incident occurs if the alteration is

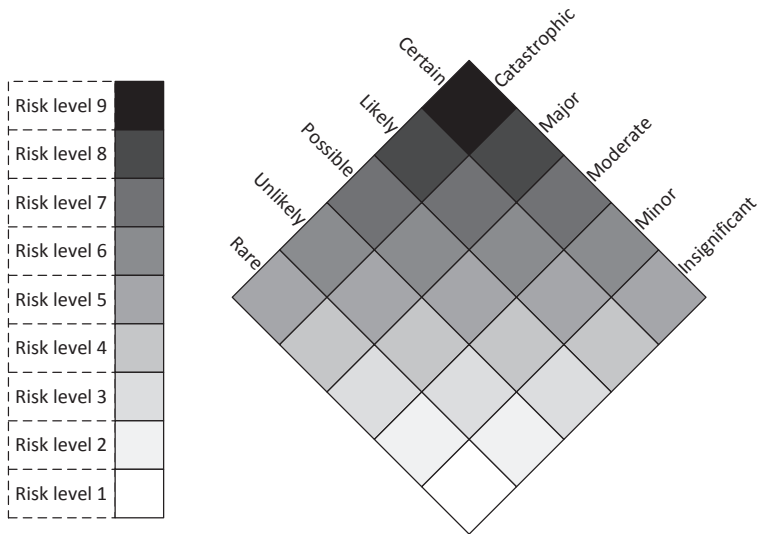


Figure 5.9: Risk evaluation matrix constructed with respect to the frequency scale and consequence scale.

successfully carried out, and Exercise Options responds with another system feature instead of the expected message *exerciseRequestConfirmation*. Thus, the unwanted incident may occur after the reception of the last message in Figure 5.8. The resulting threat scenario is shown in Figure 5.10.

The threat scenario in Figure 5.10 shows that the new message *interceptHTTPRequest* causes the message *continue(exerciseMethod)* to be received by the network tool, rather than the web application. This is an alteration, and the message *continue(exerciseMethod)* is therefore annotated with a triangle that has red borders and white fill, indicating that the message has been altered. Furthermore, we see that if the malicious user successfully tampers the parameter *exerciseMethod*, which is shown by the new messages *setExerciseMethod(otherSysFeat)* and *continue(otherSysFeat)*, then the web application responds with another system feature instead of the expected message *exerciseRequestConfirmation*. If the web application responds with another system feature, then that is an alteration of the content of message *exerciseRequestConfirmation*. The new content of the message, due to the alteration, is shown by the altered message *respOtherSysFeat*. We represent risk-related content in the messages using a font that is red colored, bold, and italic. The reception of the altered message *respOtherSysFeat* causes access to another system feature, and thus, causes the occurrence of the security risk which is illustrated by the unwanted incident message.

### 5.3.3 Threat scenario risk estimation (Step 3)

To come up with estimates, we based ourselves on knowledge data bases such as OWASP [115], Symantec threat reports [148], Cisco security reports [30], reports and papers published within the software security community, as well as expert knowledge within security testing. The estimates in Figure 5.11 illustrate risk estimation.

We see from Figure 5.11 that the malicious user successfully alters the parame-

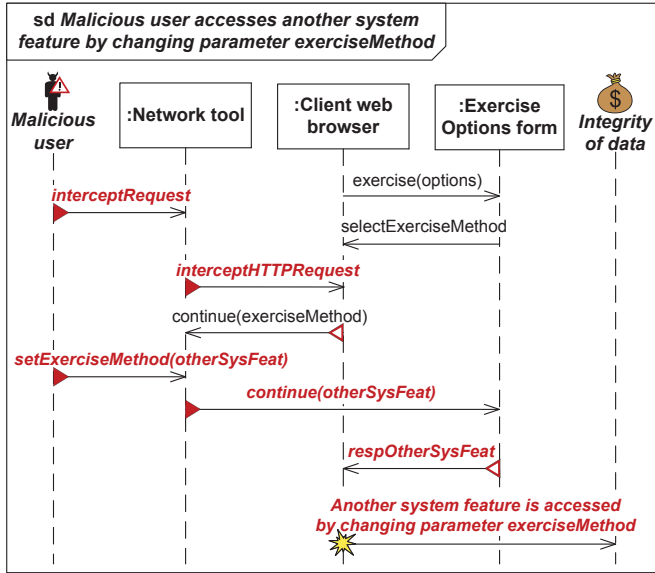


Figure 5.10: Threat scenario: Malicious user gains access to another system feature by changing parameter *exerciseMethod*.

ter *exerciseMethod* with frequency  $[20,50]:1y$ . Given that parameter *exerciseMethod* is successfully altered and transmitted, it will be received by Exercise Options with conditional ratio  $0.8$ . The conditional ratio causes the new frequency  $[16,40]:1y$  for the reception of message `continue(otherSysFeat)`. This is calculated by multiplying  $[20,50]:1y$  with  $0.8$ . Given that message `continue(otherSysFeat)` is processed by Exercise Options, it will respond with another system feature. This, in turn, causes the unwanted incident (security risk) to occur with frequency  $[16,40]:1y$ . The unwanted incident has an impact on security asset *integrity of data* with consequence *Moderate*.

### 5.3.4 Threat scenario risk evaluation (Step 4)

Let us rename the risk (*another system feature is accessed by changing parameter exerciseMethod*) as *R1*. In order to identify the likelihood value of risk *R1*, we need to map its frequency to the frequency scale in Table 5.1. We see from Figure 5.11 that risk *R1* has frequency  $[16,40]:1y$ . By mapping this frequency interval to the frequency scale in Table 5.1, we see that it overlaps the likelihood values Possible and Likely. However, we also see that the frequency interval is skewed more towards Likely than Possible. For this reason, we choose to assign likelihood Likely on risk *R1*. We map risk *R1* to the risk evaluation matrix based on likelihood *Likely* and consequence *Moderate*. The result is shown in Figure 5.12. Based on its position in the risk matrix (and the underlying color of the cell), we see that it has Risk Level 6.

Let us, for the sake of the example, say that the suspension criterion was defined as “test all risks of Risk Level 6 or more”. Based on this criterion, we select all risks in the matrix that are of Risk Level 6 or more. This means that we select risk *R1* for testing. If there had been risks that were similar in nature, then we would have to

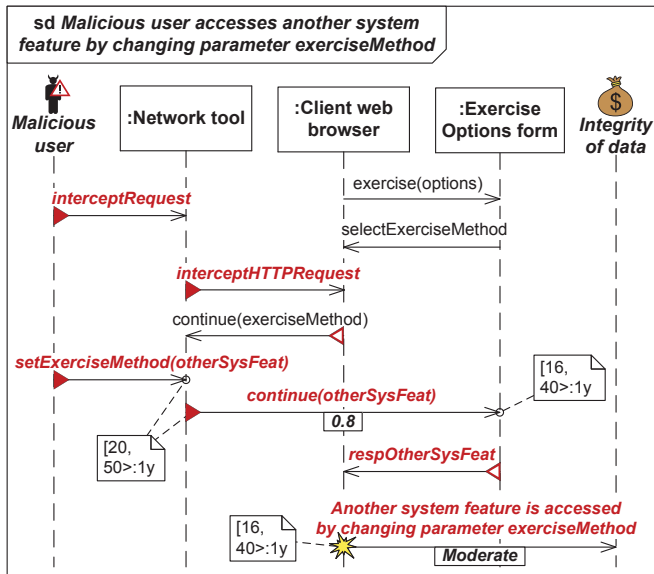


Figure 5.11: Threat scenario annotated with risk-measure annotations.

aggregate them in order to evaluate whether their risk level should be increased. If the risk level is increased as a result of risk aggregation, we assess whether they should be included in the testing. The reader is referred to Paper 2 and Paper 4 for examples of risk aggregation.

### 5.3.5 Threat scenario test case design (Step 5)

Risk *R1* is selected for testing. This means that we need to design a security test case with respect to the threat scenario in which risk *R1* occurs, that is, Figure 5.10. One possible test objective is “verify whether the malicious user is able to access another system feature by changing parameter *exerciseMethod* into a valid system parameter”. Based on this test objective, we annotate the threat scenario in Figure 5.10 with stereotypes from the UML Testing Profile [111]. The resulting security test case is shown in Figure 5.13. Needless to say, the security tester takes the role as “malicious user” in the test case.

### 5.3.6 Test execution (Step 6)

The test case in Figure 5.13 may be executed in different ways. In the case study, we executed the test case semi automatically from a black-box perspective. We intercepted the HTTP requests and responses using OWASP Zed Attack Proxy tool [116], and thereby tampered with the parameter *exerciseMethod*. The test case was executed on a test environment prepared by the client.



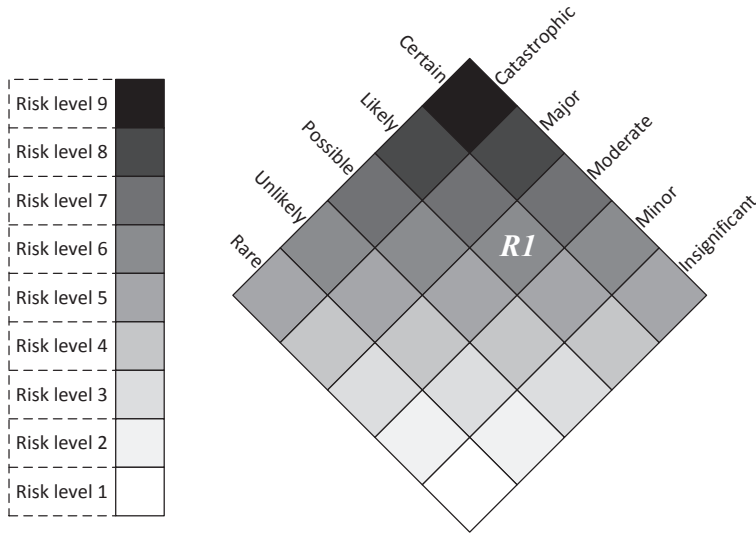


Figure 5.12: Risk  $RI$  is mapped on the risk matrix with respect to likelihood *Likely* and consequence *Moderate*.

### 5.3.7 Test incident reporting (Step 7)

In the test report we include, in addition to the test results, risk models and security tests designed with respect to the risk models. In the case study, the test case in Figure 5.13 did not fail. That is, it was not possible to access another system feature by changing parameter *exerciseMethod* into a valid system parameter. This was documented in the test report using Figure 5.13, explaining its test objective (as described above), and writing its test result. The test report may be used as input for a new run of the CORAL approach, and each test case documented in the test report may be fully repeated with respect to their models.

## 5.4 Overview of industrial case studies

In the course of the work leading up to this thesis, we carried out three industrial case studies. In the first two industrial case studies, we investigated how risk assessment may be used to identify, select, and design security test cases, as well as how security testing may be used as a means to validate and correct the security risk analysis results. The experiences we obtained from these two industrial case studies helped us to, among other things, shape the CORAL approach. In the third case study we carried out the CORAL approach in an industrial setting, in order to evaluate its applicability.

### 5.4.1 Empirical studies on the combinations of security risk analysis and security testing

The first case study was carried out between March 2011 and July 2011, while the second case study was carried out between June 2012 and January 2013. In the first

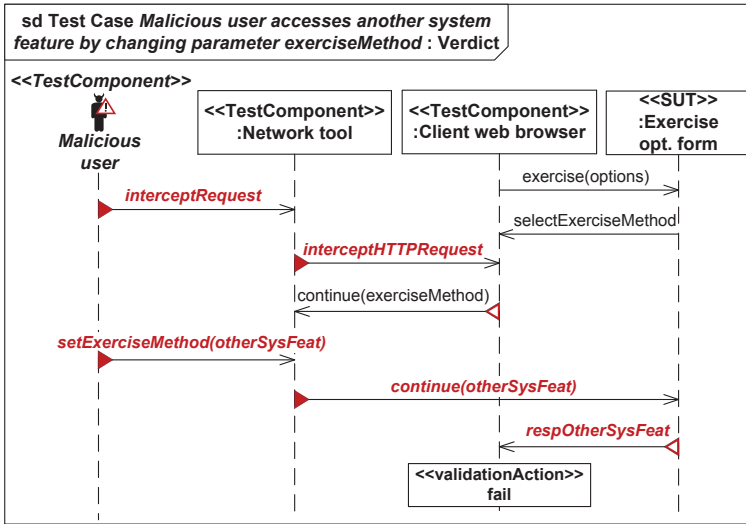


Figure 5.13: Security test case based on the threat scenario in Figure 5.10.

case study, we analyzed a multilingual web application which is designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans. In the second case study, we analyzed a mobile application designed to provide various online financial services to the users on their mobile devices. Both case studies were carried out in an industrial setting, and the analyzed systems were already deployed in production and used by a large number of customers every day.

In the case studies, we carried out security risk analysis using the CORAS approach, which is a model-driven approach to risk analysis [91]. However, we combined security risk analysis and security testing in a three-phased approach. Phase 1 expects a description of the target of evaluation. Then, based on this description, the security risk assessment is planned and carried out. The output of Phase 1 is security risk models expressed as CORAS risk models, which is used as input to Phase 2. In Phase 2, security tests are identified based on the risk models and executed (this is the risk-driven security testing part of our three-phased approach). The output of Phase 2 is security test results, which are used as input to the third and final phase. In the third phase, the CORAS risk models are validated and corrected with respect to the security test results (this is the test-driven security risk analysis part of our three-phased approach). By conducting this approach, we investigated how the risk assessment results may be used as a starting point to identify security test cases, as well as how security testing may be used as a means to improve the security risk analysis results.

With respect to risk-driven security testing, we found out that threat scenarios are a good starting point for identifying security test cases. However, we were in this approach only able to identify high-level test procedures, which were directly based on the threat scenarios in a CORAS model, before manually describing the test procedures into detailed test cases. This indicated the need for formality and preciseness in the process of designing test cases. As pointed out by our systematic literature review in Paper 1, this is also a problem that reoccurs in most of the risk-driven testing approaches in the literature. These findings guided us to improve risk-driven security

testing in terms of actively using the risk assessment for the purpose of designing test cases, instead of designing test cases after the risk assessment. This motivation shaped the development of the CORAL approach.

With respect to test-driven security risk analysis, we found out that the test results are useful for correcting the risk models in terms of adding or deleting vulnerabilities, as well as editing likelihood values. Furthermore, the test results also proved to be useful for validating the risk models in terms of discovering the presence or absence of presumed vulnerabilities, and thereby increasing the trust in the risk models. The case studies are described in detail in Paper 5.

### **5.4.2 Empirical study on the applicability of the CORAL approach**

The third case study was carried out between October 2014 and December 2014. In the third case study, we again analyzed the multilingual web application which is designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans. Similar to the first two case studies, the third case study was also carried out in an industrial setting.

In the third case study, we carried out the CORAL approach as described in Section 5.3. The purpose of the case study was to evaluate to what extent the CORAL approach helps security testers in test selection and test design. The case study results indicate that the CORAL approach is effective in terms of producing valid risk models. This is backed up by two observations. First, we identified in total 21 risks, and 11 of these risks were considered as most severe, while the remaining 10 risks were considered as low risks. By testing these 11 risks we identified 11 vulnerabilities, while by testing the remaining 10 risks we identified only 2 vulnerabilities. Second, we identified all relevant security risks compared to previous penetration tests. In addition, we identified five new security risks and did not leave out any risks of relevance for the features considered. However, the fact that we identified 2 vulnerabilities by testing low risks reflects the epistemic uncertainty that is associated with frequency/consequence estimates.

The CORAL approach seems to work equally well for black-box and white-box testing. One point worth noting for white-box testing is that the threat scenarios help locating risks at the source code level although they are initiated at the application level.

Finally, one of the most important findings we did in the case study is that the CORAL approach is very useful for identifying security test cases. We made direct use of all threat scenarios identified in the case study for the purpose of security test case design and execution. The case study is described in detail in Paper 4.

## **5.5 Overview of systematic literature review**

The objective of the systematic literature review was to review and bring forth state of the art approaches for the combined use of risk analysis and testing, based on publications related to this topic.

The systematic literature review process consisted of the following six steps: (1) define the objective of the study, (2) define research questions, (3) define the search

process including inclusion and exclusion criteria, (4) perform the search process, (5) extract data from relevant full texts, (6) analyze data and provide answers for the research questions. This process was constructed based on the guidelines given by Kitchenham and Charters [83].

As already mentioned, there are basically two main strategies for the combined use of risk analysis and testing.

- The use of testing to support the risk analysis process
- The use of risk analysis to support the testing process

We refer to the first strategy as test-driven risk analysis and to the second strategy as risk-driven testing. In the literature review, we address the following research questions.

- **RQ1** What approaches exist for performing risk-driven testing (RT) and test-driven risk analysis (TR)?
- **RQ2** For each of the identified approaches, what is the main goal, and what strategies are used to achieve that goal?
- **RQ3** Are there contexts in which TR and RT are considered to be particularly useful?
- **RQ4** How mature are the approaches, considering degree of formalization, empirical evaluation, and tool support?
- **RQ5** What are the relationships between the approaches, considering citations between the publications?

The answers to the above research questions, including the complete details for each step in the process, are documented in Paper 1. However, in general, we discovered that, within the field addressed by this survey, there is clearly need for further research. The field needs more formality and preciseness as well as dedicated tool-support. In particular, there is very little empirical evidence regarding the usefulness of the various approaches. In addition, there is little common ground for the different approaches. It seems that many of the approaches have been developed in isolation, and with little impact on the field so far.

## Overview of research papers

The main results of the work presented in this thesis are documented in the papers in Part II. In the following we give an overview of these research papers, by describing the topics of each paper and indicating how much of the results are credited the author of this thesis.

### **6.1 Paper 1: Approaches for the combined use of risk analysis and testing: a systematic literature review**

**Authors:** Gencer Erdogan, Yan Li, Ragnhild Kobro Runde, Fredrik Seehusen, Ketil Stølen.

**Publication status:** Published in the International Journal on Software Tools for Technology Transfer (vol. 16, no. 5, 2014) [40].

**My contribution:** Gencer Erdogan was one of two main authors, responsible for about 45% of the work.

**Main topics:** This paper presents a systematic literature review, and brings forth state of the art approaches for the combined use of risk analysis and testing, based on publications related to this topic. The existing approaches are first identified through a systematic literature review. The identified approaches are then classified and discussed with respect to main goal, context of use, and maturity level. The paper highlights the need for more structure and rigor in the definition and presentation of approaches. Evaluations are missing in most cases. The paper may serve as a basis for examining approaches for the combined use of risk analysis and testing, or as a resource for identifying the adequate approach to use.

## 6.2 Paper 2: A systematic method for risk-driven test case design using annotated sequence diagrams

**Authors:** Gencer Erdogan, Atle Refsdal, Ketil Stølen.

**Publication status:** Technical report SINTEF A26036, SINTEF ICT, 2014. The report presented in this thesis is a revised and extended version of the paper published in proceedings of the 1st International Workshop on Risk Assessment and Risk-driven Testing (RISK'13) [41].

**My contribution:** Gencer Erdogan was the main author, responsible for about 90% of the work.

**Main topics:** This paper presents the CORAL approach for risk-driven security testing in an example-driven manner. The paper mainly focuses on the steps related to test planning, threat scenario risk identification, threat scenario risk estimation, threat scenario risk evaluation, and threat scenario test case design. The paper also presents the CORAL risk analysis language and how it is applied in the approach. Thus, the paper may also be used as a guideline for conducting risk-driven security testing using the CORAL approach.

## 6.3 Paper 3: Schematic generation of English-prose semantics for a risk analysis language based on UML interactions

**Authors:** Gencer Erdogan, Atle Refsdal, Ketil Stølen.

**Publication status:** Technical report SINTEF A26407, SINTEF ICT, 2014. The report presented in this thesis is a revised and extended version of the paper published in proceedings of the 2nd International Workshop on Risk Assessment and Risk-driven Testing (RISK'14) [42].

**My contribution:** Gencer Erdogan was the main author, responsible for about 90% of the work.

**Main topics:** This paper presents the abstract syntax and the natural-language semantics of the CORAL language. The abstract syntax is presented in the Extended Backus-Naur Form [69]. The natural-language semantics is presented in terms of a systematic translation of CORAL diagrams into English prose. The translation is carried out in three steps. A CORAL diagram is first translated into a corresponding textual representation with respect to the abstract syntax. The textual representation of a diagram is then translated into English prose by making use of a translation algorithm, which is also presented in the paper. The paper highlights the necessity of

a natural-language semantics, provides an analytical comprehensibility evaluation of the produced English prose, and presents translation examples.

## 6.4 Paper 4: Evaluation of the CORAL approach for risk-driven security testing based on an industrial case study

**Authors:** Gencer Erdogan, Ketil Stølen, Jan Øyvind Aagedal.

**Publication status:** Technical report SINTEF A27097, SINTEF ICT, 2015. Submitted.

**My contribution:** Gencer Erdogan was the main author, responsible for about 90% of the work.

**Main topics:** This paper presents experiences from applying the CORAL approach in an industrial case. The objective of the industrial case study was to evaluate to what extent the CORAL approach helps security testers in selecting and designing test cases, as well as to what extent the CORAL approach is appropriate in the context of black-box testing and white-box testing. The system under test was a comprehensive web-based e-business application designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans, and is used by a large number of customers across Europe. The case study indicates that the CORAL approach is effective in terms of producing valid risk models and directly testable threat scenarios. Moreover, the CORAL approach seems to work equally well for black-box and white-box testing.

## 6.5 Paper 5: Assessing the usefulness of testing for validating and correcting security risk models based on two industrial case studies

**Authors:** Gencer Erdogan, Fredrik Seehusen, Ketil Stølen, Jon Hofstad, Jan Øyvind Aagedal.

**Publication status:** Published in the International Journal of Secure Software Engineering (vol. 6, no. 2, 2015) [43]. This thesis presents the camera-ready version of the paper because the published version was not available at the time of writing.

**My contribution:** Gencer Erdogan was the main author, responsible for about 90% of the work.

**Main topics:** This paper presents an evaluation of an approach in which we combine security risk analysis and security testing. The evaluation is based on two industrial case studies. The objective was to assess how useful testing is for validating and correcting security risk models, as well as how useful risk assessment is for identifying and designing security test cases. The approach is divided into three phases. Phase 1 expects a description of the target of evaluation. Then, based on this description, the security risk assessment is planned and carried out. The output of Phase 1 is security risk models, which are used as input to Phase 2. In Phase 2, security tests are identified based on the risk models and executed. The output of Phase 2 is security test results, which are used as input to the third and final phase. In the third phase, the risk models are validated and corrected with respect to the security test results. The evaluation is carried out by comparing the risk models produced before and after testing. The case studies indicate that the test results are useful for correcting the risk models in terms of adding or deleting vulnerabilities, as well as editing likelihood values. The test results are also useful for validating the risk models in terms of discovering the presence or absence of presumed vulnerabilities. Moreover, the risk models are useful for identifying security test procedures.



## Discussion

This chapter is divided into three sections. In Section 7.1 we discuss and evaluate the CORAL risk analysis language, while in Section 7.2 we discuss and evaluate the CORAL method for risk-driven security testing, in both cases with respect to the success criteria defined in Section 2.3. In Section 7.3, we relate our contributions, that is, the CORAL approach, our industrial case studies, and our systematic literature review, to the state of the art and discuss how they extend and improve the state of the art.

Before we discuss and evaluate the CORAL approach it is necessary to revisit the technology research method and have a closer look at the role of success criteria. First of all, technology research is research for the purpose of producing new and improved artifacts [140]. By artifacts, we mean objects manufactured by human beings. The success criteria are used to express the overall expectations of the artifacts. For example, as explained in Section 2.3, we have defined a success criterion which states that the CORAL method for risk-driven security testing must be comprehensible to security testers. We may in turn use this success criterion as a test to check whether we have succeeded in producing such a method. Thus, a success criterion characterizes a test that may be carried out to check if the technology research undertaken was successful [146].

The tests characterized by the success criteria may be specified and carried out using evaluation techniques such as case studies, analytical evaluations, computer simulations, surveys, and so on. As pointed out in Chapter 3, the purpose is to gather a batch of evidence in order to maximize scores related to generality, precision, and realism of the artifact under evaluation [97]. However, it is necessary to consider factors such as the nature of the success criteria (whether the success criteria consider the generality, precision or realism of an artifact), the maturity of the artifact addressed by the success criteria, and available resources such as time, cost and people when choosing evaluation techniques. While it is most desirable to maximize the generality, precision and realism of an artifact simultaneously, it is, however, an impossible act [97].

As explained in Chapter 5, the artifact in this thesis is the CORAL approach. We identified in total 10 success criteria for the CORAL approach. Because the CORAL approach is to be used by security testers in an industrial setting, we have mainly evaluated the CORAL approach in an industrial case study with respect to the success criteria (documented in Paper 4). We also evaluated our initial risk-driven testing approach, which shaped the CORAL approach, in industrial case studies (documented

in Paper 5). To complement the results obtained in the case study reported in Paper 4, we have also evaluated the CORAL approach by conducting analytical evaluations. In particular, analytical evaluations of the CORAL risk analysis language. The results obtained from these evaluations are promising, but this does not mean that the success criteria are completely fulfilled. As for any research dependent on empirical evidence, epistemic uncertainty related to the gathered batch of evidence is always present. It is therefore, in principle, not possible to state that a success criterion is *completely* fulfilled, even if the epistemic uncertainty is insignificant. However, in our case the situation is even worse, not because we have done bad work, but because it is extremely difficult to conduct experiments and studies of technology for industrial use and arrive at fully convincing conclusions. The results obtained from industrial case studies are valuable for maximizing scores related to realism, but the results are difficult to generalize. This is because the replication of industrial case studies is extremely hard to design and carry out in practice [128], and many industrial case studies in software engineering are carried out based on availability due to strict budget and time constraints [128]. We therefore discuss the extent to which the success criteria have been fulfilled, in Sections 7.1 and 7.2, given these considerations.

## 7.1 The CORAL risk analysis language

In this section, we evaluate and discuss the extent to which the success criteria related to the graphical notation, and the natural-language semantics, have been fulfilled.

### 7.1.1 Graphical notation

In order to discuss and evaluate the extent to which the graphical notation fulfills the success criteria, we make use of the SEQUAL framework [85]. SEQUAL is a general framework used for discussing and evaluating the quality of models, as well as the quality of modeling languages. The framework has been empirically shown to be valid and useful in practice [105]. Different parts of SEQUAL can be used in a number of different ways, but we use it primarily for evaluating the graphical notation of the CORAL risk analysis language. SEQUAL suggests addressing the following six areas for language quality when evaluating the quality of a modeling language [85].

1. Domain appropriateness (relates the modeling language and the domain).
2. Comprehensibility appropriateness (relates the language to the social actor interpretation).
3. Participant appropriateness (relates the participant knowledge to the language).
4. Modeler appropriateness (relates the language to the knowledge of the one doing the modeling).
5. Tool appropriateness (relates the language to the interpretation from the technical audience (tools)).
6. Organizational appropriateness (focuses on to what extent the language used is appropriate to reach organizational goals of the organization using it).

In the following, we discuss the success criteria for the graphical notation, that is Success Criteria 1, 2, and 3 described in Section 2.3.1, with respect to domain, comprehensibility, participant, modeler, and tool appropriateness. Organizational appropriateness is outside the scope of the evaluation because the CORAL language is not developed for a specific organization.

**Success Criterion 1.** *The graphical notation must be appropriate for the domain of risk-driven security testing.*

By appropriate for the domain of risk-driven security testing, we mean that the graphical notation should make an integrated use of constructs that are well known within the domain of testing, security, as well as the domain of risk assessment. As already pointed out in Chapter 5, the CORAL risk analysis language is appropriate for the domain of risk-driven security testing for several reasons. First, the language is based on UML interactions, which are among the top three modeling languages within the model-based testing community [34], and often used for testing purposes [10, 111, 153]. Second, the CORAL language is specifically developed to support security testers to conduct security risk assessment of a system under test. By annotating the constructs inherited from UML interactions with risk-related information such as threat, unwanted incident, and security asset, we bring security risk assessment to the work bench of testers without the burden of a separate risk analysis language. The risk-related concepts used in the graphical notation are in line with concepts defined in the international standard for information security risk management [72], as well as the international standard for information security management systems [71]. Third, the constructs representing risk-related information are visually based on corresponding symbols used in the CORAS language [91], which have been empirically shown to be cognitively effective [139]. Fourth, the process of risk assessment, in the CORAL approach, involves security risk modeling. The resulting risk models are used as a basis for designing and subsequently executing security test cases.

Thus, the graphical notation of the CORAL language supports security testing, as well as security risk assessment, and is therefore appropriate for the domain of risk-driven security testing.

**Success Criterion 2.** *The graphical notation must be appropriate for security testers.*

By appropriate for security testers, we mean that the graphical notation should be comprehensible to and fit for security testers. This success criterion is related to comprehensibility appropriateness, participant appropriateness, and modeler appropriateness. The SEQUAL framework refers to the principles provided by Moody [104] for discussing these areas of appropriateness. Moody [104] suggests the following nine principles: Semiotic clarity, perceptual discriminability, semantic transparency, complexity management, cognitive integration, visual expressiveness, dual coding, graphic economy, and cognitive fit.

The principle of semiotic clarity states that there should be a 1:1 correspondence between semantic constructs and graphical symbols. In the CORAL language, there is a 1:1 correspondence because none of the anomalies related to symbol redundancy, symbol overload, symbol excess, and symbol deficit are possible. Symbol redundancy

occurs when multiple graphical symbols are used to represent the same semantic construct. Symbol overload occurs when multiple semantic constructs are represented by the same graphical symbol. Symbol excess occurs when graphical symbols do not correspond to any semantic construct. Symbol deficit occurs when there are semantic constructs that are not represented by any graphical symbols.

The principle of perceptual discriminability states that different symbols should be clearly distinguishable from each other. The symbols representing a threat, an unwanted incident, a security asset, and a frequency estimate are easily distinguishable from each other because of their distinct shapes and colors. Figure 5.11 shows an example of a CORAL risk model, and we see that the aforementioned symbols are easily distinguishable. Although the conditional ratio symbol and the consequence symbol are rectangular, they are easily distinguishable because conditional ratios may be assigned on general, new, and altered messages, while consequences are assigned only on unwanted incident messages. In addition, conditional ratios are always represented as nonnegative real numbers, while consequences are always represented textually. However, the new, altered, and deleted messages are similar in the sense that they all have a triangular shape at the transmission end, but they are distinguishable with respect to the coloring inside the triangles (see Figure 5.4). In our experience, CORAL risk models typically contain a greater number of new messages compared to the number of altered and deleted messages. In some cases, particularly in large risk models, this makes it somewhat difficult to spot the altered/deleted messages. We may mitigate this by using different shapes at the transmission end on new, altered and deleted messages. However, the reason why we use triangles (in combination with the color red) is to support semantic transparency, which is discussed next.

The principle of semantic transparency states that symbols should use visual representations whose appearance suggests their meaning. The risk-related symbols used in the CORAL language are based on the corresponding symbols used in the CORAS risk analysis language [91]. The symbols in CORAS are designed to convey their meaning. For example, the symbol depicting a human with “devil horns” conveys a deliberate threat, and the triangle with red borders conveys a threat scenario. This has been empirically shown to be cognitively effective [139], and these concepts are also commonly used in security testing [117], which is why we use similar symbols in the CORAL language.

The principle of complexity management states that the language should include explicit mechanisms for dealing with complexity. Dealing with complexity, in this context, refers to the ability of a visual notation to represent information without overloading the human mind [104]. In order to reduce the complexity of risk models, the CORAL language makes use of the UML construct `ref` (interaction use) to divide complex risk models into simpler risk models. The `ref` construct refers to an interaction, and it is shorthand for copying the contents of the referred interaction [110, p. 501]. Because of its modular property, the `ref` construct may also be used to support cognitive integration. The principle of cognitive integration states that the language should include explicit mechanisms to support integration of information from different diagrams. The cognitive integration supported by the `ref` construct is limited because it only gives an abstract description, and not any detailed information, about the referred interaction. However, this is sufficient for constructing high-level risk models, which are useful for getting an overview of the various risk models and their relationships. Thus, a security tester may divide complex risk models into simpler risk models, as

well as compose high-level risk models, by making use of the `ref` construct.

The principles of visual expressiveness and dual coding refer to the usage of the full range and capacities of visual variables, and the usage of text to complement graphics, respectively. In addition to using symbols that are distinguishable with respect to shape and color, the CORAL language uses a red colored, bold, and italic font to highlight the risk-related information (text) on messages. Based on our experience, this convention is useful only for new and altered messages, as well as unwanted incidents. The text on new messages is always formatted as risk-related information because the messages are initiated by threats. The text on altered messages is formatted as risk-related information when highlighting the alteration in the message. This could be a part of the text or the complete text on the altered message. The text on unwanted incidents are always formatted as risk-related information because they represent that assets are harmed or reduced in value. This formatting strengthens the visual expressiveness and helps security testers to keep track of and distinguish between risk-related and non risk-related information on the messages.

The principle of graphic economy states that the number of different graphical symbols should be cognitively manageable. The number of symbols should ideally be limited to 6 symbols. However, this only applies if a single visual variable is used, for example, if a language only uses shapes [104]. If a language consists of more than 6 symbols, which is the case in the CORAL language, then one can deal with graphic complexity by increasing visual expressiveness. We have explained above how this is achieved and also how text is used to complement the graphics, which in turn strengthens the visual expressiveness. In particular, we position the symbols representing new, altered, deleted, and unwanted incident messages so that they are horizontally aligned with the message, as well as correctly oriented with respect to the message direction. These two visual variables give an additional increase to the visual expressiveness [104].

The principle of cognitive fit states that the language should use different visual dialects for different tasks and audiences. The CORAL language is mainly to be used by security testers, for the purpose of risk-driven security testing. This implies that the CORAL language must provide concepts and a corresponding graphical notation necessary to carry out security risk assessment, as well as security testing. As discussed above, and as pointed out in the discussion of the first success criterion, the CORAL language does provide this. However, this also means that the CORAL language requires testers to be familiar with security risk assessment. Security testers usually have this required background, because they often have to carry out activities related to security risk assessment, such as creating security abuse/misuse cases, performing architectural risk analysis, and building risk-driven security test plans [117].

Based on the above discussion, we find it reasonable to assume that the graphical notation of the CORAL language is appropriate for security testers.

**Success Criterion 3.** *The graphical notation must be appropriate for tool implementation and method development.*

The third success criterion is related to tool appropriateness. A prerequisite for tool interpretation is that the language must have a syntax and semantics that are formally defined [85]. As discussed in Section 5.2, the CORAL language is accompanied by an abstract syntax as well as a natural-language semantics. Testers may use the abstract syntax in order to create risk models that are syntactically correct, and the natural-

language semantics in order to clearly and consistently document, communicate and analyze risks (documented in Paper 3). The abstract syntax and the natural-language semantics may also be used by tool and method developers. The graphical notation is supported by the necessary foundation for tool implementation and is therefore appropriate for tool implementation.

### 7.1.2 Natural-language semantics

The fulfillment of the success criteria related to the natural-language semantics is thoroughly discussed in Paper 3. In the following, we highlight the main arguments that support the fulfillment of the success criteria.

**Success Criterion 4.** *The English-prose semantics of CORAL risk models must be comprehensible to security testers when conducting risk assessment.*

The comprehensibility of the resulting English prose is supported both from a general viewpoint and from a security testing viewpoint.

From a general viewpoint, we observe the following two points. First, the structure of the translations is similar to the structure of their corresponding CORAL risk models. In particular, the ordering of the translated constructs is maintained. Second, the user-defined text is unchanged in the translations. By user-defined text, we mean the text typed in CORAL risk models, such as the text on messages, lifelines, frequency assignments, consequence assignments, and so on. The reader is referred to Paper 3 for translation examples.

From a security testing viewpoint, we observe that risk-related concepts from the CORAL language are integrated with concepts from UML interactions in the resulting English prose. As already explained in Section 5.1, UML interactions are often used by testers for testing purposes. It is therefore reasonable to assume that security testers understand the concepts from UML interactions. Moreover, we find it reasonable to assume that security testers also comprehend the risk-related concepts we introduce in the CORAL language, such as altered messages and messages representing unwanted incidents, because these are concepts that are also known within the security testing community. For example, in fuzz testing, the expected behavior of a system is attempted altered by providing invalid, unexpected, or random data, which may lead to unwanted incidents [113].

**Success Criterion 5.** *The English-prose semantics of the constructs inherited from UML interactions must be consistent with their semantics in the UML standard.*

In Paper 3 we present the English-prose semantics of the constructs in the CORAL language. Moreover, we show that the English-prose semantics of the constructs inherited from UML interactions are consistent with their semantics in the UML standard. We do this by relating the English-prose semantics of the constructs in the CORAL language with the semantics of the corresponding constructs given in the UML standard [110].

**Success Criterion 6.** *The complexity of the resulting English prose must scale linearly with the complexity of CORAL risk models in terms of size.*

As shown in Paper 3, the definition of the translation algorithm ensures that the structure of the resulting English prose mirrors the risk model that is translated, and that there is a linear relationship between the size of the risk model and its resulting English prose. A formal argument that this would hold for any risk model  $d$  could be given based on induction over the syntactical structure of  $d$ .

## 7.2 The CORAL method for risk-driven security testing

**Success Criterion 7.** *The method must be comprehensible to security testers.*

The steps in the CORAL method related to test planning, threat scenario test case design, test execution, and test incident reporting are in line with corresponding steps in the dynamic test process defined in the software testing standard ISO/IEC/IEEE 29119 [74]. The dynamic test process is used to carry out testing within a particular phase of testing (for example unit, integration, system and acceptance) or type of testing (for example security testing, performance testing, usability testing) [74]. The steps provided by the dynamic test process are steps typically carried out in testing projects and are therefore familiar to software testers. This is true whether the testing is focused on security or other software qualities. Moreover, the steps in the CORAL method related to threat scenario risk identification, threat scenario risk estimation, and threat scenario risk evaluation, are also familiar to security testers because security testers often have to carry out activities related to security risk assessment [117, 151].

This means that the CORAL method for risk-driven security testing is in line with standard testing processes, as well as activities typically carried out by security testers. Thus, it is reasonable to assume that the CORAL method for risk-driven security testing is comprehensible to security testers.

**Success Criterion 8.** *The method must produce security risk models that are valid.*

In order to evaluate to what extent the CORAL method produces valid security risk models, we addressed the following research questions in the industrial case study reported in Paper 4.

- **Research Question A:** To what extent is the risk level of identified risks correct?
- **Research Question B:** To what extent are relevant risks identified compared to previous penetration tests?

The two variables that determine the risk level of a risk, that is, the frequency value and the consequence value, are estimates based on data gathered during the security risk assessment. In other words, these estimates tell us to what degree the identified risks exist. Thus, in principle, the higher the risk level, the more likely it is to reveal vulnerabilities causing the risk. The same applies the other way around. That is, the lower the risk level, the less likely it is to reveal vulnerabilities causing the risk.

The results obtained for Research Question A show that 11 vulnerabilities were revealed by testing the risks considered as most severe, while only 2 vulnerabilities were



revealed by testing the risks considered as low risks. Additionally, the 2 vulnerabilities identified by testing the low risks were assigned low severity. These findings indicate that the risk levels of identified risks were highly accurate. In contrast, if we had found 2 vulnerabilities by testing the most severe risks, and 11 vulnerabilities by testing the low risks, then that would have indicated inaccurate risk levels, and thus a risk assessment of low quality. However, the fact that we found two vulnerabilities by testing the low risks reflects the inevitable epistemic uncertainty that is associated with frequency/consequence estimates. This basically means that the more empirical data we gather for the estimates, the more certain we may be about the risk levels. The results obtained for Research Question B show that we identified all relevant security risks compared to previous penetration tests. In addition, we identified five new security risks and did not leave out any risks of relevance for the features considered.

This initial evaluation of the CORAL method indicates that the method is capable of producing valid risk models. To this end, the success criterion is fulfilled. However, as future work, the CORAL method may be supported by techniques for mitigating epistemic uncertainty. Further empirical studies are then needed to evaluate the benefits and the feasibility of such techniques in the context of the CORAL method.

**Success Criterion 9.** *The method must produce security risk models that are directly testable.*

CORAL risk models represent threat scenarios that the system under test is exposed to. In order to evaluate to what extent the CORAL method produces security risk models that are directly testable, we addressed the following research question in the industrial case study documented in Paper 4.

- To what extent are the threat scenarios that causes the identified risks directly testable?

Recall that the CORAL language extends UML interactions. This means that threat scenarios represented by CORAL risk models are expressed in terms of UML interaction sequence diagrams. By a directly testable threat scenario, we mean a threat scenario that can be reused and specified as a security test case based on its interactions. The results obtained for the above research question point out that all threat scenarios were directly testable. We believe this result is generalizable because, in the CORAL approach, risks are identified at the level of abstraction testers commonly work when designing test cases [34]. This is also backed up by the fact that we made direct use of all threat scenarios as security test cases. Thus, based on this, it is reasonable to assume that the CORAL method is capable of producing threat scenarios that are directly testable. Moreover, this supports one of the main goals of the CORAL approach, which is, as pointed out in Chapter 5, to help security testers systematically design test cases with respect to the risk assessment results.

However, it is important to note that the CORAL approach is designed to be used by individual security testers, or by a group of security testers collaborating within the same testing project. The risk models produced by a tester, or a group of testers working together, will most likely be used by the same tester(s) to design test cases, and consequently execute the test cases.



**Success Criterion 10.** *The method should be appropriate for black-box testing and white-box testing.*

In order to evaluate the CORAL method with respect to Success Criterion 10, we addressed the following research question in the industrial case study documented in Paper 4.

- To what extent is the CORAL approach useful for black-box testing and white-box testing, respectively?

By appropriate for black-box testing and white-box testing, we mean that the complete CORAL method for risk-driven security testing should be conductible both in a white-box testing context, as well as in a black-box testing context. As documented in Paper 4, all the steps in the CORAL method were fully applicable both in a black-box context, and in a white-box context. Thus, it is reasonable to assume that the CORAL method is appropriate for black-box testing and white-box testing.

## 7.3 Relating our contributions to the state of the art

In the following, we relate our contributions to the state of the art and discuss how they extend and improve the state of the art. In order to get a holistic picture, we relate the CORAL approach to risk-driven testing approaches at a general level, and not only to risk-driven testing approaches specialized for security. The purpose of risk-driven testing is to focus the testing process on the most severe risks that the system under test is exposed to. This is achieved by first conducting risk assessment of a system under test, and then carrying out the testing process with respect to the risk assessment results. We will therefore relate the CORAL approach to other risk-driven testing approaches from a risk assessment perspective (Section 7.3.1), as well as from a testing perspective (Section 7.3.2). Moreover, in Section 7.3.3 we relate the empirical evaluations of the CORAL approach, that is, our industrial case studies, to the evaluations of the state of the art approaches. Finally, in Section 7.3.4 we relate our systematic literature review to similar literature reviews.

### 7.3.1 Relating the CORAL approach to other risk-driven testing approaches from a risk assessment perspective

Risk assessment is the overall process of identifying, estimating, and evaluating risks [70]. In the context of risk-driven testing approaches, risk assessment is carried out to identify, estimate, and evaluate risks posed on the system under test. Current risk-driven testing approaches carry out risk assessment using different techniques. However, we may divide the approaches in two overall groups based on how they carry our risk assessment: approaches that support model-based risk assessment, and approaches that support table-based risk assessment (see Table 7.1).

The most commonly used table-based risk assessment approach is Hazard and Operability (HazOp) analysis [68]. The approaches provided by Amland [4], Chen et al. [27, 28], Kumar et al. [86], Redmill [121, 122], Souza et al. [141, 142], and Yoon et

al. [168] use techniques inherited from HazOp analysis, while the approaches provided by Bai et al. [8, 9], Entin et al. [38], Felderer et al. [45, 47], Hosseingholizadeh [64], Huang et al. [66], Rosenberg et al. [123], Schneidewind [133], Stallbaum et al. [145], and Wong et al. [162] use techniques that can be regarded as specializations of HazOp analysis in which they mainly focus on metrics such as code complexity, functional complexity, and testability when identifying and estimating risks. The CORAL approach belongs to the group that supports model-based risk assessment, and we will therefore not further discuss approaches supporting table-based risk assessment. The reader is referred to Chapter 4 for details related to approaches that support table-based risk assessment. In the following, we relate the CORAL approach to the approaches supporting model-based risk assessment by discussing risk identification, risk estimation, and risk evaluation separately.

Table 7.1: Approaches supporting model-based risk assessment and approaches supporting table-based risk assessment.

<b>Risk-driven testing approaches</b>	
<b>Model-based risk assessment</b>	<b>Table-based risk assessment</b>
Botella et al. [15]	Amland [4]
Casado et al. [23, 24]	Bai et al. [8, 9]
Gleirscher [54, 55]	Chen et al. [27, 28]
Großmann et al. [57, 58]	Entin et al. [38]
Kloos et al. [84]	Felderer et al. [45, 47]
Murthy et al. [107]	Hosseingholizadeh [64]
Nazier et al. [109]	Huang et al. [66]
Ray et al. [120]	Kumar et al. [86]
Seehusen [135]	Redmill [121, 122]
Wendland et al. [159]	Rosenberg et al. [123]
Xu et al. [166]	Schneidewind [133]
Zech et al. [171, 172]	Souza et al. [141, 142]
Zimmermann et al. [174]	Stallbaum et al. [145]
	Wong et al. [162]
	Yoon et al. [168]

**Risk identification.** The approaches provided by Casado et al. [23, 24], Gleirscher [54, 55], Kloos et al. [84], Nazier et al. [109], Ray et al. [120], and Zimmermann et al. [174] make use of Fault Tree Analysis [155] for the purpose of identifying safety risks. Moreover, Casado et al. [23, 24], Kloos et al. [84], Nazier et al. [109], and Ray et al. [120] explicitly show how fault tree analysis is conducted in their approach, while Gleirscher [54, 55] and Zimmermann et al. [174] only refer to the usage of fault tree analysis for the purpose of identifying safety risks. While fault tree analyses in these approaches are useful for identifying specific safety risks, they do not include information such as the threat profile initiating the chain of events causing the risk, the

likelihood of an event occurring, and the consequence of a risk. These are constructs that are necessary in a risk assessment [70].

The approaches provided by Botella et al. [15], Großmann et al. [57, 58], and Seehusen [135] identify security risks by making use of the CORAS risk analysis language [91]. As already mentioned in Chapter 5, the graphical notation and the concepts used in the CORAL approach are based on constructs in the CORAS approach. These approaches are therefore closely related to the CORAL approach with respect to the process of identifying risks. However, the main difference is that the threat scenarios and risks identified in these approaches are described at a high-level of abstraction [135], and that the risk models are represented as directed acyclic graphs, while in the CORAL approach the risk models are represented as interaction sequence diagrams.

Murthy et al. [107] makes use of Microsoft's threat modeling process for identifying risks. However, it is unclear exactly how the authors use this process for identifying risks. Microsoft's threat modeling process is a comprehensive process which consists of different approaches to identify risks, such as attack tree modeling, system modeling combined with a security assessment technique referred to as STRIDE, making use of a categorized threat list, as well as making use of attack patterns [102].

The approach provided by Wendland et al. [159] makes use of behavior trees [59] for the purpose of identifying risks at the level of requirements engineering. This approach is not concerned about identifying risks caused by threat scenarios, but rather carrying out qualitative risk assessment with an overall question in mind: what are the risks if certain requirements are not fulfilled? Having this in mind, the behavior trees are annotated with qualitative risk exposure values.

Xu et al. [166] make use of Predicate/Transition Nets (based on Petri Nets) [53] to model possible attack paths, which in turn may be used to identify possible security risks that the system under test is exposed to. Although threat modeling as given by this approach is useful to identify possible security risks, it does not intend to focus the testing on the most risky paths. The purpose of the approach is to identify possible attack paths by threat modeling, and then identify the corresponding test cases.

Zech et al. [171, 172] identify security risks by matching attack patterns on the public interfaces of a system under test. The pattern matching is carried out automatically, which in turn produces risk models in terms of UML class diagrams [110]. The produced risk models contain information about possible attack scenarios that may be carried out on the public interfaces. However, it does not contain information regarding the threat initiating the attack, and the chain of events causing the security risk.

What is common for all the approaches discussed above is that they model risks and the system under test in separate models using separate modeling languages. This makes it difficult to get an intuitive understanding with respect to exactly how and where the risks affect the system under test. The risk models in the CORAL approach represent specific threat scenarios, security risks caused by the threat scenarios, and the relevant aspects of the system affected by the risks, within the same model. This enables testers to identify exactly how and where certain security risks may occur.

**Risk estimation.** None of the approaches using fault tree analysis [23, 24, 54, 55, 84, 109, 120, 174] make use of risk-measure annotations for estimating the likelihood of the occurrence of a risk (fault), or the consequence of the occurring risk. The approaches using fault tree analysis leave it to the tester to identify the most severe risks without

providing any basis for estimating the risks. This is also true for the approach provided by Xu et al. [166].

The approaches using the CORAS approach [15, 57, 58, 135] estimate risks in a similar manner as in the CORAL approach. This is because the process of estimating risks in the CORAL approach is inspired by the process of estimating risks in the CORAS approach. However, the main difference is related to what we refer to as conditional ratios in the CORAL approach. While the CORAS approach makes use of conditional probabilities to calculate the frequency of a threat scenario, we make use of conditional ratios for the same purpose.

The remaining approaches [107, 159, 171, 172] make use of qualitative values for representing likelihoods and consequences. The qualitative likelihood and consequence values are typically defined as High, Medium and Low. However, none of these approaches explain what is meant by High/Medium/Low.

**Risk evaluation.** The purpose of risk evaluation is to evaluate and prioritize risks that the system under test is exposed to, with respect to the likelihood and consequence estimates. The testing process is then focused on the most severe security risks. However, since none of the approaches using fault tree analysis, including the approach provided by Xu et al. [166], make use of risk-measures in terms of likelihood and consequence, risk evaluation with respect to risk-measures is not carried out in these approaches. Instead, these approaches leave it to testers to evaluate the risks in an ad hoc manner based on their experience and knowledge.

The approaches making use of the CORAS approach use the risk estimates assigned on a CORAS risk model to make a prioritized list of threat scenarios [15, 57, 58, 135]. The threat scenarios are then used as a basis for identifying/designing test cases. This is different from our approach. In the CORAL approach, we map the risks to a risk evaluation matrix based on the risk estimates, and then make a prioritized list of risks. We then select the most severe risks that the system under test is exposed to, and design test cases by making use of the threat scenarios causing the selected risks. Moreover, we also aggregate similar risks to select additional threat scenarios that need to be tested.

The remaining approaches make use of  $3 \times 3$  risk evaluation matrices that are constructed from qualitative likelihood and consequence values defined as High, Medium, and Low [107, 159, 171, 172]. In traditional risk analysis, risk evaluation matrices are designed to group the various combinations of likelihood and consequence into three to five risk levels (for example, high, medium, and low). Such risk levels cover a wide spectrum of likelihood and consequence combinations and are typically used as a basis for deciding whether to accept, monitor or treat risks. However, in the setting of risk-driven testing, one is concerned about prioritizing risks to test certain aspects of the system under test which is exposed to risks. A higher granularity with respect to risk levels, as presented in the CORAL approach, may therefore be more practical.

### 7.3.2 Relating the CORAL approach to other risk-driven testing approaches from a testing perspective

Risk assessment may help testers to focus the activities in a test process (that is, test planning, test design, test implementation, test execution, and test evaluation/reporting)

on the most severe risks posed on the system under test [48]. However, current risk-driven testing approaches use risk assessment mainly to focus the test planning process in terms of test selection, as well as to focus the test design process with respect to the most severe risks. We will therefore relate the CORAL approach to current risk-driven testing approaches by discussing how risk assessment is used to support test selection and test design. Moreover, since the CORAL approach supports model-based testing, we will relate the CORAL approach to the risk-driven testing approaches also supporting model-based testing (see Table 7.2).

The approach provided by Seehusen [135] focuses on the identification of test procedures expressed in natural language, and techniques for prioritizing the test procedures. The approach does not consider model-based testing, and is therefore not discussed in this section. Although the approaches provided by Chen et al. [27,28], Entin et al. [38], and Stallbaum et al. [145] do not support model-based risk assessment, they do however support model-based testing. We will therefore include these approaches in the following discussion.

Table 7.2: Approaches supporting model-based testing and approaches not supporting model-based testing.

<b>Risk-driven testing approaches</b>	
<b>Model-based testing</b>	<b>Not model-based testing</b>
Botella et al. [15]	Amland [4]
Casado et al. [23,24]	Bai et al. [8,9]
Chen et al. [27,28]	Felderer et al. [45,47]
Entin et al. [38]	Hosseingholizadeh [64]
Gleirscher [54,55]	Huang et al. [66]
Großmann et al. [57,58]	Kumar et al. [86]
Kloos et al. [84]	Redmill [121,122]
Murthy et al. [107]	Rosenberg et al. [123]
Nazier et al. [109]	Schneidewind [133]
Ray et al. [120]	Seehusen [135]
Stallbaum et al. [145]	Souza et al. [141,142]
Wendland et al. [159]	Wong et al. [162]
Xu et al. [166]	Yoon et al. [168]
Zech et al. [171,172]	
Zimmermann et al. [174]	

**Test selection.** Test selection with respect to risk assessment basically involves establishing the scope of the testing with respect to the most severe risks posed on the system under test. As explained in Section 7.3.1, the most severe risks are identified as a result of risk assessment. The approaches using fault tree analysis [23,24,54,55,84,109,120,174] focus on the identification of safety risks. Other approaches also considering safety risks are provided by Chen et al. [27,28], and Entin et al. [38] who make

use of techniques inherited from HazOp analysis. However, while these approaches do identify safety risks, they do not estimate nor evaluate risks, as explained in Section 7.3.1. Thus, they select tests based on safety risks perceived as the most severe in an ad hoc manner. That is, they do not employ systematic techniques for estimating and evaluating the identified safety risks in order to select tests accordingly.

The approaches using CORAS [15, 57, 58, 135] select tests with respect to the most severe risks by systematically identifying, estimating, and evaluating risks posed on the system under test. These approaches are closely related to the CORAL approach with respect to selecting tests. However, as mentioned in Section 7.3.1, these approaches identify risks at a high-level of abstraction, which is in contrast to our approach, where we identify risks at a low-level of abstraction and select tests accordingly.

The approaches provided by Murthy et al. [107], Stallbaum et al. [145], and Zech et al. [171, 172] test the most complex features of a system under test based on estimates related to code complexity, functional complexity and testability; the more complex features are, the more risky they are, which means that they should be tested accordingly. While code complexity and functional complexity are important factors for identifying risky parts/features of a system, they are not the only factors. In practice, other factors may be considered, for example, vulnerability statistics and incident reports.

Wendland et al. [159] select tests with respect to qualitative risk levels assigned to requirements. The aim is to plan the testing at the level of requirements engineering. In contrast to our approach, this is test selection at a high-level of abstraction.

Similar to the approaches addressing safety risks, Xu et al. [166] do not estimate nor evaluate risks, and therefore do not select tests with respect to a systematic risk assessment. They select tests based on security risks perceived as the most severe, in an ad hoc manner.

**Test design.** Test design with respect to risk assessment involves specifying test cases or test procedures that address the most severe risks. The various approaches use different techniques for modeling test cases. The approaches using fault tree analysis [23, 24, 54, 55, 84, 109, 120, 174], as well as the approach provided by Entin et al. [38], model test cases by making use of state machine diagrams similar to UML state machines [110]. Although most of the risk-driven testing approaches model state machines based on fault trees for the purpose of designing test cases, there are some existing gaps between fault trees and state machines that has to be taken into consideration [82]. The approach provided by Ray et al. [120] maps fault trees to state machines using three different modeling notations: use cases are defined based on fault trees, then sequence diagrams are modeled with respect to each use case, and finally the relevant aspects of the sequence diagrams are modeled as state machines. Moreover, while the approaches mentioned above focus on modeling state-based test cases, our approach focuses on modeling interaction-based test cases.

The approaches using CORAS [15, 57, 58] identify high level test procedures based on the CORAS risk models. Then, for each test procedure, they identify an associated test pattern. While Botella et al. [15] make use of UML class diagrams, object diagrams, and state machines to create the test model (instantiation of the test pattern), Großmann et al. [57, 58] incorporate guidelines for how certain test cases should be modeled as part of the test pattern instantiation (referred to as test design strategy). Similar to Großmann et al. [57, 58], Wendland et al. [159] also make use of test design strategies for

modeling test cases. In other words, they do not restrict the test design to one specific modeling notation, but rather use different modeling notations for different test pattern instantiations. However, as pointed out by Neto et al. [34], UML class diagrams, state machine diagrams, and UML interaction sequence diagrams are modeling notations most commonly used within model-based testing. In the CORAL approach, we create test models in terms of interaction sequence diagrams. Another important point is that these approaches combine different modeling notations for the purpose of modeling and representing the test model, while in the CORAL approach, we represent the system model, risk model, and test model using one notation.

The approaches provided by Chen et al. [27,28] and Stallbaum et al. [145] create test models using UML activity diagrams [110]. A test case in these approaches is a path in an activity diagram. Both approaches are concerned about identifying the most risky paths in the test model, and thereby selecting those paths for testing. However, the purpose of an activity diagram is to provide a sequential flow of related activities and conditions of the flow [110]. The activities are described at a high-level of abstraction and are useful for describing high-level test procedures, similar to the approaches using CORAS [15, 57, 58, 135].

The approaches provided by Murthy et al. [107] and Zech et al. [171,172] make use of misuse cases for the purpose of identifying test cases. Murthy et al. [107] make use of high-level misuse cases similar to UML use case diagrams [125,137] for the purpose of designing security test cases. However, these kinds of diagrams are mostly useful in the context of requirements engineering [125,137]. Zech et al. [171,172] make use of misuse cases that are represented as UML class diagrams, and are used to generate test case code. Similar to Zech et al. [171,172], Xu et al. [166] also generate test case code, but in the latter approach the authors make use of Predicate/Transition Nets for this purpose.

### 7.3.3 Relating our empirical evaluations to evaluations in other risk-driven testing approaches

As pointed out in our systematic literature review in Paper 1, there is very little empirical evidence regarding the usefulness of the various approaches. Most of the approaches either carry out evaluations/experiments based on in-house or industrial software [8, 9, 64, 66, 107, 120, 133, 162, 166, 168, 171, 172], or provide small practical examples in order to illustrate the applicability of the approach [15, 23, 24, 54, 55, 57, 58, 84, 86, 109, 135, 141, 142, 145, 174]. Some approaches do not report on any evaluations [121–123, 159]. However, a handful of approaches have been carried out and evaluated in industrial case studies/projects [4, 28, 38, 47]. We will therefore, in the following, relate our evaluations to the evaluations provided by Amland [4], Chen et al. [28], Entin et al. [38], and Felderer et al. [47].

Amland [4] evaluates the risk-driven testing approach by conducting an industrial case study on a retail banking application. This evaluation is mainly concerned about the usefulness of risk assessment to support the test management in focusing resources on the most risky areas of the system under test. Amland [4] highlights that the benefits of risk-driven testing are reduced resource consumption, and improved quality.

Chen et al. [28] evaluate their regression testing approach by applying it within an industrial project where they test three components of IBM WebSphere Commerce. The approach uses risk assessment to identify regression test suites addressing the most



severe risks. They evaluate their approach by comparing how well their test suites cover risks, versus how well non-risk-driven test suites cover risks.

Entin et al. [38] evaluate the effectiveness of their model-based testing approach within an industrial project that follows the Scrum development life cycle. As part of their approach, they apply risk assessment (as given by Stallbaum et al. [145]) for the purpose of prioritizing test suites. Based on their evaluation, Entin et al. [38] argues that risk-driven test prioritization makes the testing more efficient in terms of resource consumption in the test project.

Felderer et al. [47] evaluate their generic risk-driven testing approach by applying it in an industrial project, where they test a web application. However, this evaluation is not concerned about the effectiveness/efficiency of the risk-driven testing approach, but it rather focuses on the challenges related to *introducing* risk-driven testing in an industrial test project. Based on their experience, the authors provide guidelines for integrating risk-driven testing in industrial test projects.

What is common for all these evaluations is that they are mainly concerned about how risk-driven testing supports *test management* in focusing the resources on the most severe risks. In the evaluation of the CORAL approach (documented in Paper 4), we are mainly concerned about evaluating the effectiveness of our approach in terms of producing valid risk models, as well as producing risk models that are directly testable. That is, in our evaluation, we are mainly concerned about how the *tester* may benefit from the risk assessment results, and not only how the test management may focus the test project based on risk assessment results.

Moreover, in our first two industrial case studies reported in Paper 5, we evaluate how useful risk assessment results are for the purpose of identifying test cases, as well as how useful test results are for validating and correcting security risk models. That is, not only did we evaluate how testers may benefit from the risk assessment results, but also how risk analysts may benefit from test results.

### 7.3.4 Relating our systematic literature review to similar literature reviews

Chapter 4 gives an overview of state of the art approaches to risk-driven testing identified through our systematic literature review. Seven of the 28 approaches in our survey are also included in a recent study by Alam et al. [3]. The authors focus on risk-driven testing, and they describe each of the approaches in great detail. They also include example tables and figures from some of the approaches they review. In contrast, the main contribution in our literature review is a comparison of the approaches with respect to a set of predefined criteria. Using a systematic search process, our survey includes papers on risk-driven testing not discussed by Alam et al. [3], while at the same time excluding approaches not published in peer-reviewed journals/proceedings. In addition, we also include papers related to test-driven risk analysis.

Another recent literature review has been conducted by Felderer et al. [46]. This literature review discusses 17 risk-driven testing approaches. Fifteen of these approaches are also included in our literature review. The remaining two approaches are not included in our literature review because we exclude approaches not published in peer-reviewed journals/proceedings. However, while we analyze and discuss the approaches for risk-driven testing with respect to main goal, context of use, and maturity level (including how risk assessment and testing are carried out), Felderer et al. [46] fo-



cus specifically on discussing and analyzing how *risk estimation* is carried out in the approaches they review.

In our survey, we only consider the approaches combining risk analysis and testing. Using a search process similar to the one we carry out in the literature review documented in Paper 1, Sulaman et al. [147] review methods for risk analysis of IT systems. Their study may be seen as complementary to our literature review, as the sets of papers are disjoint.



## Conclusion

This chapter concludes Part I of the thesis by summarizing the results and pointing out directions for future work.

### 8.1 The CORAL approach

The continuous increase of sophisticated cyber security risks exposed to the public, industry, and government through the web, mobile devices, social media, as well as targeted attacks via state-sponsored cyberespionage, clearly show the need for software security. Security testing is one of the most important security practices in order to assure an acceptable level of software security. However, due to the complexity of systems and software it is impossible to exhaustively test every single aspect of any given system under test. In addition, security testing is limited by strict budget and time constraints. Moreover, when testing security-critical software, security testers face the problem of determining the tests that are most likely to reveal severe security vulnerabilities. In response to this challenge, the testing community has proposed a security testing approach which is supported by security risk assessment. The purpose is to focus the testing process on the most severe security risks that the system under test is exposed to. This is referred to as risk-driven security testing.

However, in our systematic literature review, which was conducted as part of the work leading up to this thesis, we discovered that current risk-driven testing approaches suffer from three main issues, namely:

- the field needs more formality and preciseness, as well as dedicated tool support,
- there is very little empirical evidence regarding the usefulness of the approaches, and
- risk assessment is carried out at a high-level of abstraction (for example, business level), while test cases are defined at a low-level of abstraction (for example, implementation level). This introduces a gap between identified risks and the test cases exploring the risks.

This thesis contributes specifically to the domain of risk-driven *security* testing from a model-based perspective. To address the above issues, our overall objective has been to develop a model-based approach to risk-driven security testing that is:

1. comprehensible to security testers,
2. useful for the purpose of selecting and designing test cases with respect to the risk assessment results,
3. effective in the sense that it produces risk models that are valid and directly testable, and
4. sufficiently rigorous to support the development of tools and methods.

To this end, we have put forward a model-based approach to risk-driven security testing named CORAL. The CORAL approach consists of a risk analysis language and a method for risk-driven security testing, which are specialized for security testers. There are six main reasons to why the CORAL approach is comprehensible to security testers.

- The risk analysis language is based on UML interactions, which are among the top three modeling languages within the model-based testing community and often used for testing purposes.
- The risk-related information in the language is represented by annotating the constructs inherited from UML interactions with appropriate graphical icons. This, in turn, brings risk assessment to the work bench of testers without the burden of a separate risk analysis language.
- The graphical notation of the language is in accordance with the principles for achieving effective visual notations [104], and is in addition based on corresponding symbols used in the CORAS language [91] which have been empirically shown to be cognitively effective [139].
- The resulting English prose from the natural-language semantics uses concepts that are known to security testers, it preserves the structure of CORAL risk models, and it keeps the user-defined text in CORAL risk models unchanged. In addition, the resulting English prose of the constructs inherited from UML interactions is consistent with their semantics in the UML standard. Moreover, the complexity of the resulting English prose scales linearly with the complexity of the CORAL risk models in terms of size.
- The resulting risk models are used as a basis for designing and subsequently executing security test cases.
- The method for risk-driven security testing is in line with standard testing processes, as well as activities typically carried out by security testers.

The risk analysis language is formalized in terms of an abstract syntax and a schematically defined natural-language semantics. The abstract syntax provides a set of rules, in terms of a context-free grammar, that defines the correct combinations of the constructs in the CORAL risk analysis language. The syntax may be used to create risk models that are syntactically correct. The natural-language semantics provides a set of rules for schematically translating CORAL risk models into English prose

to clearly and consistently document, communicate and analyze security risks. These points support the rigorousness of the CORAL approach.

The CORAL approach helps security testers in selecting test cases by systematically carrying out security risk assessment of a system under test. Based on the risk assessment, security testers are able to identify security risks that the system under test is exposed to, and then select the most severe risks to test. As part of the risk assessment, testers must create risk models using the CORAL language. The resulting risk models are used for designing and executing security test cases.

We have also carried out the CORAL approach in an industrial case study, from which we have gathered promising results indicating that the approach is effective in the sense that it produces valid and directly testable risk models. The empirical studies are discussed closer in the following section.

## 8.2 Empirical studies

As part of the development and evaluation process of the CORAL approach to risk-driven security testing, we conducted three industrial case studies. The first two case studies are reported in Paper 5, while the third case study is reported in Paper 4. In addition to the case studies, we have also carried out an analytical evaluation of the CORAL risk analysis language as explained in Section 7.1.

The first two industrial case studies provided valuable insight in how security testing may benefit from security risk assessment (that is, risk-driven security testing), as well as how security risk analysis may benefit from security testing (that is, test-driven security risk analysis).

With respect to risk-driven security testing, we found that threat scenarios are a good starting point for identifying security test cases. However, we also found that in order to fully benefit from this, there had to be an established process to refine high-level threat scenarios into detailed test cases. This was an important insight because this indicated the need for formality and preciseness in the process of designing test cases. These findings guided us to improve risk-driven security testing in terms of actively using the risk assessment for the purpose of designing test cases, instead of designing test cases after the risk assessment. By doing so, we bridge the gap between high-level security risks and low-level security test cases. This motivation shaped the development of the CORAL approach.

With respect to test-driven security risk analysis, we found that the test results are useful for correcting the risk models in terms of adding or deleting vulnerabilities, as well as editing likelihood values. Furthermore, the test results also proved to be useful for validating the risk models in terms of discovering the presence or absence of presumed vulnerabilities, and thereby increasing the trust in the risk models.

The third industrial case study provided valuable insight with respect to the applicability of the CORAL approach. In particular, the case study results indicated that the CORAL approach is capable of guiding security testers in identifying risk models that are valid and of high quality. Moreover, the CORAL approach proved to be applicable both in the context of black-box testing and white-box testing. One of the most important findings we did in the case study was that the CORAL approach is very useful for identifying security test cases. We made direct use of all threat scenarios represented by CORAL risk models for the purpose of security test case design and

execution. This, in turn, bridges the gap between high-level security risks and low-level security test cases.

### 8.3 Systematic literature review

In order to obtain a holistic view of the domain of risk-driven testing, as well as test-driven risk analysis, we conducted a systematic literature review, which is documented in Paper 1. In the literature review we include approaches addressing risk-driven testing and test-driven risk analysis in general. By doing so, we obtained a complete overview of the domain. Moreover, Paper 1 reports on a total of 22 approaches related to risk-driven testing, and 2 approaches related to test-driven risk analysis. In Chapter 4, we include 3 additional approaches related to risk-driven testing, and 1 additional approach related to test-driven risk analysis. These approaches were not included in Paper 1 because they were published after we had conducted our systematic literature review. This gives a total of 25 approaches related to risk-driven testing, and a total of 3 approaches related to test-driven risk analysis. The main finding of our systematic literature review are summarized in Section 8.1.

### 8.4 Directions for future work

There are a number of interesting directions for future work. As explained in Paper 4, we carried out the CORAL approach in an industrial case study where we focused mainly on web application security. However, it would be beneficial with more industrial case studies where we carry out the CORAL approach in different domains. For example cloud security, network protocol security, security in smart home systems, and so on. This would provide more empirical evidence with respect to the applicability of the CORAL approach and help us identify strengths and weaknesses of the CORAL approach with respect to the different domains.

Another interesting future work would be to investigate the comprehensibility of the graphical notation of the CORAL language by interviewing or surveying security testers. Although we have specifically designed the CORAL approach for security testers, and discussed how the CORAL approach is appropriate for security testers, as well as for the domain of risk-driven security testing, it is not obvious how the modeling notation may be interpreted by practitioners.

One obvious direction of future research is to develop a modeling tool for the CORAL approach. As pointed out in our literature review in Paper 1, the field of risk-driven testing needs more formality and proper tool support. The CORAL language is already formalized, and this opens for appropriate tool support for the CORAL approach. A supporting tool would obviously increase the efficiency of risk modeling. Moreover, it could also support automatic test execution since the CORAL approach makes direct use of the risk models for test identification and test execution purposes. Also, another interesting point worth exploring in the context of tools, is to investigate to what extent known attack patterns, such as those listed in Common Attack Pattern Enumeration and Classification (CAPEC) [22], may be automatically transformed into CORAL risk models. This would assist security testers in creating risk models based on known attack patterns.

The CORAL risk models are created during the security risk assessment process, and represent potential threat scenarios posed on a system under test, which in turn cause security risks that the system under test may be exposed to. In the CORAL approach we make use of the risk models for the purpose of risk-driven test selection, as well as for risk-driven test design. However, one interesting future work would be to investigate how the CORAL risk models may assist the domain of requirements engineering, for the purpose of identifying security requirements. Moreover, since we make use of the CORAL risk models to design security test cases, it would be interesting to investigate how this can be applied in the context of test-driven software development (TDD).





# Bibliography

- [1] H. Abelson and G.J. Sussman. The Structure and Interpretation of Computer Programs. *Journal of Curriculum Studies*, 19(1):91–103, 1987.
- [2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [3] M. Alam and A.I. Khan. Risk-based testing techniques: A perspective study. *International Journal of Computer Applications*, 65(1):33–41, 2013.
- [4] S. Amland. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53:287–295, 2000.
- [5] A. Armando, R. Carbone, L. Compagna, K. Li, and G. Pellegrino. Model-Checking Driven Security Testing of Web-Based Applications. In *Proc. 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW'10)*, pages 361–370. IEEE Computer Society, 2010.
- [6] J.L. Austin. *How to do things with words*. Oxford University Press, 1962.
- [7] C.W. Bachman. Data Structure Diagrams. *ACM SIGMIS Database*, 1(2):4–10, 1969.
- [8] X. Bai and R.S. Kenett. Risk-based adaptive group testing of semantic web services. In *Proc. 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, pages 485–490. IEEE Computer Society, 2009.
- [9] X. Bai, R.S. Kennett, and W. Yu. Risk assessment and adaptive group testing of semantic web services. *International Journal of Software Engineering and Knowledge Engineering*, pages 595–620, 2012.
- [10] P. Baker, Z.R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer, 2008.
- [11] R.V. Binder, A. Kramer, and B. Legeard. 2014 Model-based Testing User Survey: Results. [http://model-based-testing.info/wordpress/wp-content/uploads/2014\\_MBT\\_User\\_Survey\\_Results.pdf](http://model-based-testing.info/wordpress/wp-content/uploads/2014_MBT_User_Survey_Results.pdf), Model-Based Testing Community, 2014. Accessed June 11, 2015.

- [12] R.V. Binder, B. Legeard, and A. Kramer. Model-based testing: where does it stand? *Communications of the ACM*, 58(2):52–56, 2015.
- [13] M. Blackburn, R. Busser, A. Nauman, and R. Chandramouli. Model-based approach to security test automation. In *Proc. International Software Quality Week*, pages 1–8. NIST, 2001.
- [14] J. Botella, F. Bouquet, J.-F. Capuron, F. Lebeau, B. Legeard, and F. Schadle. Model-Based Testing of Cryptographic Components – Lessons Learned from Experience. In *Proc. 6th International Conference on Software Testing, Verification and Validation (ICST'13)*, pages 192–201. IEEE Computer Society, 2013.
- [15] J. Botella, B. Legeard, F. Peureux, and A. Vernotte. Risk-Based Vulnerability Testing Using Security Test Patterns. In *Proc. 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, pages 337–352. Springer, 2014.
- [16] P. Bourque and R. Dupuis. Guide to the software engineering body of knowledge 2004 version. Technical Report 19759, IEEE Computer Society, 2004.
- [17] J. Bozic, D.E. Simos, and F. Wotawa. Attack Pattern-based Combinatorial Testing. In *Proc. 9th International Workshop on Automation of Software Test (AST'14)*, pages 1–7. ACM, 2014.
- [18] J. Bozic and F. Wotawa. XSS Pattern for Attack Modeling in Testing. In *Proc. 8th International Workshop on Automation of Software Test (AST'13)*, pages 71–74. IEEE Computer Society, 2013.
- [19] J. Bozic and F. Wotawa. Security Testing Based on Attack Patterns. In *Proc. 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'14)*, pages 4–11. IEEE Computer Society, 2014.
- [20] F.P. Brooks Jr. The Computer Scientist As Toolsmith II. *Communications of the ACM*, 39(3):61–68, 1996.
- [21] M. Broy and K. Stølen. *Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement*. Springer, 2001.
- [22] Common Attack Pattern Enumeration and Classification (CAPEC). <https://capec.mitre.org/>. Accessed May 31, 2015.
- [23] R. Casado, J. Tuya, and M. Younas. Testing long-lived web services transactions using a risk-based approach. In *Proc. 10th International Conference on Quality Software (QSIC'10)*, pages 337–340. IEEE Computer Society, 2010.
- [24] R. Casado, J. Tuya, and M. Younas. A framework to test advanced web services transactions. In *Proc. 4th International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 443–446. IEEE Computer Society, 2011.
- [25] W.S. Chao. *System Analysis and Design: SBC Software Architecture in Practice*. Lambert Academic Publishing, 2009.

- 
- [26] P.P.-S. Chen. The Entity-relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
- [27] Y. Chen and R.L. Probert. A risk-based regression test selection strategy. In *Proc. 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'03), Fast Abstract*, pages 305–306. Chillarege Press, 2003.
- [28] Y. Chen, R.L. Probert, and D.P. Sims. Specification-based regression test selection with risk analysis. In *Proc. 2002 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'02)*, pages 1–14. IBM Press, 2002.
- [29] T.S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.
- [30] Cisco 2015 Annual Security Report. <http://www.cisco.com/web/offers/lp/2015-annual-security-report/index.html?KeyCode=000657658>. Accessed May 5, 2015.
- [31] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.
- [32] R. Davison, M.G. Martinsons, and N. Kock. Principles of canonical action research. *Information Systems Journal*, 14(1):65–86, 2004.
- [33] P.J. Denning. Is Computer Science Science? *Communications of the ACM*, 48(4):27–31, 2005.
- [34] A.C. Dias Neto, R. Subramanyan, M. Vieira, and G.H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *Proc. 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASEL'07)*, pages 31–36. ACM, 2007.
- [35] The Free Dictionary: Dictionary, Encyclopedia and Thesaurus. Definition of “method”. <http://www.thefreedictionary.com/method>. Accessed May 27, 2015.
- [36] businessdictionary.com: Definition of “objective”. <http://www.businessdictionary.com/definition/objective.html>. Accessed June 24, 2015.
- [37] G. Dodig-Crnkovic. Scientific Methods in Computer Science. In *Proc. Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, pages 1–7. M”alardalen University, 2002.
- [38] V. Entin, M. Winder, B. Zhang, and S. Christmann. Introducing model-based testing in an industrial scrum project. In *Proc. 7th International Workshop on Automation of Software Test (AST'12)*, pages 43–49. IEEE Computer Society, 2012.
- [39] G. Erdogan, Y. Li, R.K. Runde, F. Seehusen, and K. Stølen. Conceptual Framework for the DIAMONDS Project. Technical Report A22798, SINTEF Information and Communication Technology, 2012.

- [40] G. Erdogan, Y. Li, R.K. Runde, F. Seehusen, and K. Stølen. Approaches for the combined use of risk analysis and testing: a systematic literature review. *International Journal on Software Tools for Technology Transfer*, 16(5):627–642, 2014.
- [41] G. Erdogan, A. Refsdal, and K. Stølen. A Systematic Method for Risk-driven Test Case Design Using Annotated Sequence Diagrams. In *Proc. 1st International Workshop on Risk Assessment and Risk-driven Testing (RISK'13)*, pages 93–108. Springer, 2014.
- [42] G. Erdogan, A. Refsdal, and K. Stølen. Schematic Generation of English-prose Semantics for a Risk Analysis Language Based on UML Interactions. In *Proc. 2nd International Workshop on Risk Assessment and Risk-driven Testing (RISK'14)*, pages 205–310. IEEE Computer Society, 2014.
- [43] G. Erdogan, F. Seehusen, K. Stølen, J. Hofstad, and J.Ø. Aagedal. Assessing the Usefulness of Testing for Validating and Correcting Security Risk Models Based on Two Industrial Case Studies. *International Journal of Secure Software Engineering*, 6(2):90–112, 2015.
- [44] G. Erdogan and K. Stølen. Risk-driven Security Testing versus Test-driven Security Risk Analysis. In *Proc. 1st ESSoS Doctoral Symposium (ESSoS-DS'12)*, pages 5–10. Citeseer, 2012.
- [45] M. Felderer, C. Haisjackl, R. Breu, and J. Motz. Integrating manual and automatic risk assessment for risk-based testing. In *Proc. 4th International Conference on Software Quality (SWQD'12)*, pages 159–180. Springer, 2012.
- [46] M. Felderer, C. Haisjackl, V. Pekar, and R. Breu. An Exploratory Study on Risk Estimation in Risk-Based Testing Approaches. In *Proc. 7th Software Quality Days (SWQD'15)*, pages 32–43. Springer, 2015.
- [47] M. Felderer and R. Ramler. Experiences and challenges of introducing risk-based testing in an industrial project. In *Proc. 5th International Conference on Software Quality (SWQD'13)*, pages 10–29. Springer, 2013.
- [48] M. Felderer and I. Schieferdecker. A taxonomy of risk-based testing. *International Journal on Software Tools for Technology Transfer*, 16(5):559–568, 2014.
- [49] ETSI Technical Committee Methods for Testing and Specification (MTS). Methods for Testing and Specification (MTS); Security Testing; Basic Terminology. ETSI Technical Report 101 583 v1.1.1, European Telecommunications Standards Institute, 2015.
- [50] C.P. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice Hall Professional Technical Reference, 1979.
- [51] V. Garousi and J. Zhi. A survey of software testing practices in Canada. *Journal of Systems and Software*, 86(5):1354–1376, 2013.
- [52] M.-C. Gaudel and P. Le Gall. Testing data types implementations from algebraic specifications. In *Formal methods and testing*, pages 209–239. Springer, 2008.

- 
- [53] H.J. Genrich. Predicate/Transition Nets. In *Petri Nets: Central Models and Their Properties*, pages 207–247. Springer, 1987.
- [54] M. Gleirscher. Hazard-based selection of test cases. In *Proc. 6th International Workshop on Automation of Software Test (AST'11)*, pages 64–70. ACM, 2011.
- [55] M. Gleirscher. Hazard analysis of technical systems. In *Proc. 5th International Conference on Software Quality (SWQD'13)*, pages 104–124. Springer, 2013.
- [56] S. Gopalakrishnan, J. Krogstie, and G. Sindre. Adapting UML Activity Diagrams for Mobile Work Process Modelling: Experimental Comparison of Two Notation Alternatives. In *The Practice of Enterprise Modeling*, pages 145–161. Springer, 2010.
- [57] J. Großmann, M. Berger, and J. Viehmann. A Trace Management Platform for Risk-Based Security Testing. In *Proc. 1st International Workshop on Risk Assessment and Risk-driven Testing (RISK'13)*, pages 120–135. Springer, 2014.
- [58] J. Großmann, M. Schneider, J. Viehmann, and M.-F. Wendland. Combining Risk Analysis and Security Testing. In *Proc. 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, pages 322–336. Springer, 2014.
- [59] Behavior Tree Group. Behavior Tree Notation. Technical Report v1.0, ARC Center for Complex Systems, 2007.
- [60] S. Hagerman, A. Andrews, S. Elakeili, and A. Gario. Security Testing of an Aerospace Launch System. In *Proc. IEEE Aerospace Conference (AC'15)*, pages 1–11. IEEE Computer Society, 2015.
- [61] J. Hartmanis. Some Observations About the Nature of Computer Science. In *Foundations of Software Technology and Theoretical Computer Science*, pages 1–12. Springer, 1993.
- [62] J. Hartmanis. On computational complexity and the nature of computer science. *ACM Computing Surveys*, 27(1):7–16, 1995.
- [63] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A.J.H. Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using Formal Specifications to Support Testing. *ACM Computing Surveys*, 41(2):9:1–9:76, 2009.
- [64] A. Hosseingholizadeh. A source-based risk analysis approach for software test optimization. In *Proc. 2nd International Conference on Computer Engineering and Technology (ICCET'10)*, pages 601–604. IEEE Computer Society, 2010.
- [65] M. Howard and S. Lipner. *The Security Development Lifecycle: SDL, a Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.
- [66] P. Huang, X. Ma, D. Shen, and Y. Zhou. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proc. 36th International Conference on Software Engineering (ICSE'14)*, pages 60–71. ACM, 2014.

- [67] IEEE Computer Society. *IEEE 829 - Standard for Software and System Test Documentation*, 2008.
- [68] International Electrotechnical Commission. *IEC 61882, Hazard and Operability studies (HAZOP studies) - Application guide*, 2001.
- [69] International Organization for Standardization. *ISO/IEC 14977:1996(E), Information technology – Syntactic metalanguage – Extended BNF, first edition*, 1996.
- [70] International Organization for Standardization. *ISO 31000:2009(E), Risk management – Principles and guidelines*, 2009.
- [71] International Organization for Standardization. *ISO/IEC 27000:2009(E), Information technology – Security techniques – Information security management systems – Overview and vocabulary*, 2009.
- [72] International Organization for Standardization. *ISO/IEC 27005:2011(E), Information technology – Security techniques – Information security risk management*, 2011.
- [73] International Organization for Standardization. *ISO/IEC/IEEE 29119-1:2013(E), Software and system engineering - Software testing - Part 1: Concepts and definitions*, 2013.
- [74] International Organization for Standardization. *ISO/IEC/IEEE 29119-2:2013(E), Software and system engineering - Software testing - Part 2: Test process*, 2013.
- [75] W. Johansson, M. Svensson, U.E. Larson, M. Almgren, and V. Gulisano. T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols. In *Proc. 7th International Conference on Software Testing, Verification and Validation (ICST'14)*, pages 323–332. IEEE Computer Society, 2014.
- [76] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 1990.
- [77] J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *UML 2002 – The Unified Modeling Language*, pages 412–425. Springer, 2002.
- [78] J. Jürjens. Model-based security testing using umlsec: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93–104, 2008.
- [79] J. Jürjens and G. Wimmel. Specification-Based Testing of Firewalls. In *Perspectives of System Informatics*, pages 308–316. Springer, 2001.
- [80] C. Kaner. The Impossibility of Complete Testing. Technical report, <http://www.kaner.com/pdfs/imposs.pdf>, 1997. Accessed May 30, 2015.
- [81] K. Katkalov, N. Moebius, K. Stenzel, M. Borek, and W. Reif. Modeling test cases for security protocols with securemdd. *Computer Networks*, 58(0):99–111, 2014.

- 
- [82] H. Kim, W.E. Wong, V. Debroy, and D. Bae. Bridging the Gap Between Fault Trees and UML State Machine Diagrams for Safety Analysis. In *Proc. 17th Asia Pacific Software Engineering Conference (APSEC'10)*, pages 196–205. IEEE Computer Society, 2010.
- [83] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, Keele University, and University of Durham, 2007.
- [84] J. Kloos, T. Hussain, and R. Eschbach. Risk-based testing of safety-critical embedded systems driven by Fault Tree Analysis. In *Proc. 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 26–33. IEEE Computer Society, 2011.
- [85] J. Krogstie. *Model-Based Development and Evolution of Information Systems - A Quality Approach*. Springer, 2012.
- [86] N. Kumar, D. Sosale, S.N. Konuganti, and A. Rathi. Enabling the adoption of aspects-testing aspects: A risk model, fault model and patterns. In *Proc. 8th ACM International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 197–206. ACM, 2009.
- [87] J.H. Larkin and H.A. Simon. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, 11(1):65–100, 1987.
- [88] F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte. Model-Based Vulnerability Testing for Web Applications. In *Proc. 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13)*, pages 445–452. IEEE Computer Society, 2013.
- [89] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
- [90] C.Y. Lester and F. Jamerson. Incorporating Software Security into an Undergraduate Software Engineering Course. In *Proc. 3rd International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'09)*, pages 161–166. IEEE Computer Society, 2009.
- [91] M.S. Lund, B. Solhaug, and K. Stølen. *Model-Driven Risk Analysis: The CORAS Approach*. Springer, 2011.
- [92] S. Maag. Final Security Testing Techniques. Technical Report D5.WP2, [http://www.itea2-diamonds.org/\\_docs/D5\\_WP2\\_v10\\_FINAL\\_Final\\_Security\\_Testing\\_Techniques.pdf](http://www.itea2-diamonds.org/_docs/D5_WP2_v10_FINAL_Final_Security_Testing_Techniques.pdf), DIAMONDS Consortium, 2013. Accessed June 15, 2015.
- [93] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu. Security Test Generation using Threat Trees. In *Proc. ICSE Workshop on Automation of Software Test (AST'09)*, pages 62–69. IEEE Computer Society, 2009.
- [94] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu. A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258, 2013.



- [95] S.T. March and G.F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, 1995.
- [96] P. McBrien, A.H. Seltveit, and B. Wangler. An Entity-Relationship Model Extended to Describe Historical Information. In *Proc. Information Systems and Management of Data (CISMOD'92)*, pages 244–260. Citeseer, 1992.
- [97] J.E. McGrath. *Groups: interaction and performance*. Prentice-Hall, 1984.
- [98] G. McGraw. Software security. *Security & Privacy, IEEE*, 2(2):80–83, 2004.
- [99] G. McGraw. *Software Security - Building Security In*. Addison-Wesley, 2006.
- [100] G. McGraw, S. Miguez, and J. West. Building Security In Maturity Model (BSIMM-V). Technical Report Release 5.1.2, BSIMM Project, 2013.
- [101] Microsoft Threat Modeling Tool. <http://www.microsoft.com/en-us/sdl/adopt/threatmodeling.aspx>. Accessed June 16, 2015.
- [102] Threat Modeling. <https://msdn.microsoft.com/en-us/library/ff648644.aspx>. Accessed April 18, 2015.
- [103] Security Development Lifecycle. <http://www.microsoft.com/en-us/sdl/default.aspx>. Accessed May 5, 2015.
- [104] D.L. Moody. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *Transactions on Software Engineering, IEEE*, 35(6):756–779, 2009.
- [105] D.L. Moody, G. Sindre, T. Brasethvik, and A. Solvberg. Evaluating the Quality of Information Models: Empirical Testing of a Conceptual Model Quality Framework. In *Proc. 25th International Conference on Software Engineering (ICSE'03)*, pages 295–305. IEEE Computer Society, 2003.
- [106] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon. A Model-Based Framework for Security Policy Specification, Deployment and Testing. In *Model Driven Engineering Languages and Systems*, pages 537–552. Springer, 2008.
- [107] K.K. Murthy, K.R. Thakkar, and S. Laxminarayan. Leveraging risk based testing in enterprise systems security validation. In *Proc. 1st International Conference on Emerging Network Intelligence (EMERGING'09)*, pages 111–116. IEEE Computer Society, 2009.
- [108] G.J. Myers, T. Badgett, and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2011.
- [109] R. Nazier and T. Bauer. Automated risk-based testing by integrating safety analysis information into system behavior models. In *Proc. 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW'12)*, pages 213–218. IEEE Computer Society, 2012.
- [110] Object Management Group. *Unified Modeling Language (UML), superstructure, version 2.4.1*, 2011. OMG Document Number: formal/2011-08-06.



- 
- [111] Object Management Group. *UML Testing Profile (UTP), version 1.2*, 2013. OMG Document Number: formal/2013-04-03.
- [112] Object Management Group. *Object Constraint Language, version 2.4*, 2014. OMG Document Number: formal/2014-02-03.
- [113] P. Oehlert. Violating assumptions with fuzzing. *Security Privacy, IEEE*, 3(2):58–62, 2005.
- [114] The Open Group. *The Open Group Architecture Framework Version 9.1*, 2011.
- [115] Open Web Application Security Project. [https://www.owasp.org/index.php/Main\\_Page](https://www.owasp.org/index.php/Main_Page). Accessed January 13, 2015.
- [116] OWASP Zed Attack Proxy. [https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project). Accessed January 13, 2015.
- [117] B. Potter and G. McGraw. Software Security Testing. *Security & Privacy, IEEE*, 2(5):81–85, 2004.
- [118] A. Pretschner. Model-Based Testing. In *Proc. 27th International Conference on Software Engineering (ICSE'05)*, pages 722–723. ACM, 2005.
- [119] Managing cyber risks in an interconnected world - Key findings from The Global State of Information Security Survey 2015. <http://www.pwc.com/gx/en/consulting-services/information-security-survey/assets/the-global-state-of-information-security-survey-2015.pdf>. Accessed May 5, 2015.
- [120] M. Ray and D.P. Mohapatra. Risk analysis: A guiding force in the improvement of testing. *IET Software*, 7:29–46, 2013.
- [121] F. Redmill. Exploring risk-based testing and its implications. *Software Testing, Verification and Reliability*, 14:3–15, 2004.
- [122] F. Redmill. Theory and practice of risk-based testing. *Software Testing, Verification and Reliability*, 15:3–20, 2005.
- [123] L. Rosenberg, R. Stapko, and A. Gallo. Risk-based object oriented testing. In *Proc. 24th Annual Software Engineering Workshop*, pages 1–6. NASA, Software Engineering Laboratory, 1999.
- [124] P.S. Rosenbloom. A new framework for computer science and engineering. *IEEE Computer*, 37(11):23–28, 2004.
- [125] Lillian Røstad. *Access Control in Healthcare Information Systems*. PhD thesis, Norwegian University of Science and Technology, 2008.
- [126] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.
- [127] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, 2005.

- [128] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 2012.
- [129] I. Schieferdecker. Model-Based Fuzz Testing. In *Proc. 5th IEEE International Conference on Software Testing, Verification and Validation (ICST'12)*, page 814. IEEE Computer Society, 2012.
- [130] I. Schieferdecker. Model-based testing. *IEEE Software*, 29(1):14–18, 2012.
- [131] I. Schieferdecker, J. Großmann, and M. Schneider. Model-Based Security Testing. In *Proc. 7th Workshop on Model-Based Testing (MBT'12)*, pages 1–12. Electronic Proceedings in Theoretical Computer Science (EPTCS 80), 2012.
- [132] M. Schneider, J. Großmann, I. Schieferdecker, and A. Pietschker. Online Model-Based Behavioral Fuzzing. In *Proc. 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13)*, pages 469–475. IEEE Computer Society, 2013.
- [133] N.F. Schneidewind. Risk-driven software testing and reliability. *International Journal of Reliability Quality and Safety Engineering*, 14:99–132, 2007.
- [134] J.R. Searle. *Speech acts: An essay in the philosophy of language*. Cambridge University Press, 1969.
- [135] F. Seehusen. A Technique for Risk-Based Test Procedure Identification, Prioritization and Selection. In *Proc. 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, pages 277–291. Springer, 2014.
- [136] M.E. Senko, E.B. Altman, M.M. Astrahan, and P.L. Fehder. Data structures and accessing in data-base systems, I: Evolution of information systems. *IBM Systems Journal*, 12(1):30–44, 1973.
- [137] G. Sindre and A.L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.
- [138] Smartesting. <http://www.smartesting.com/en/>. Accessed June 16, 2015.
- [139] B. Solhaug and K. Stølen. The CORAS Language - Why it is designed the way it is. In *Proc. 11th International Conference on Structural Safety and Reliability (ICOSSAR'13)*, pages 3155–3162. Taylor and Francis, 2013.
- [140] I. Solheim and K. Stølen. Technology research explained. Technical Report A313, SINTEF Information and Communication Technology, 2007.
- [141] E. Souza, C. Gusmão, K. Alves, J. Venâncio, and R. Melo. Measurement and control for risk-based test cases and activities. In *Proc. 10th Latin American Test Workshop (LATW'09)*, pages 1–6. IEEE Computer Society, 2009.
- [142] E. Souza, C. Gusmão, and J. Venâncio. Risk-based testing: A case study. In *Proc. 7th International Conference on Information Technology: New Generations (ITNG'10)*, pages 1032–1037. IEEE Computer Society, 2010.

- 
- [143] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.
- [144] J.M. Spivey. *The Z Notation: A Reference Manual, 2nd Edition*. Prentice Hall International, 1992.
- [145] H. Stallbaum, A. Metzger, and K. Pohl. An automated technique for risk-based test case generation and prioritization. In *Proc. 3rd International Workshop on Automation of Software Test (AST'08)*, pages 67–70. ACM, 2008.
- [146] K. Stølen. *Teknologivitenskap*. Draft, 2015.
- [147] S.M. Sulaman, K. Weyns, and M. Höst. A Review of Research on Risk Analysis Methods for IT Systems. In *Proc. 17th International Conference on Evaluation and Assessment in Software Engineering (EASE'13)*, pages 86–96. ACM, 2013.
- [148] Symantec Internet Security Threat Report, Volume 20, April 2015. [http://www.symantec.com/security\\_response/publications/threatreport.jsp](http://www.symantec.com/security_response/publications/threatreport.jsp). Accessed May 5, 2015.
- [149] Testing Standards Working Party. *BS 7925-1 Vocabulary of terms in software testing*, 1998.
- [150] L. Thomas, W. Xu, and D. Xu. Mutation Analysis of Magento for Evaluating Threat Model-Based Security Testing. In *Proc. 35th Annual Computer Software and Applications Conference Workshops (COMPSACW'11)*, pages 184–189. IEEE Computer Society, 2011.
- [151] H.H. Thompson. Why Security Testing Is Hard. *Security & Privacy, IEEE*, 1(4):83–86, 2003.
- [152] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.
- [153] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.
- [154] A. van Lamsweerde. Formal Specification: A Roadmap. In *Proc. 22nd International Conference on Software Engineering (ICSE'00)*, pages 147–159. ACM, 2000.
- [155] W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. *Fault Tree Handbook*. Technical Report NUREG-0492, U.S. Nuclear Regulatory Commission, 1981.
- [156] J. Wang, T. Guo, P. Zhang, and Q. Xiao. A Model-Based Behavioral Fuzzing Approach for Network Service. In *Proc. 3rd International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC'13)*, pages 1129–1134. IEEE Computer Society, 2013.
- [157] J. Wang, P. Zhang, L. Zhang, H. Zhu, and Y. Xiaojun. A Model-Based Fuzzing Approach for DBMS. In *Proc. 8th International Conference on Communications and Networking in China (CHINACOM'13)*, pages 426–431. IEEE Computer Society, 2013.

- [158] L. Wang, E. Wong, and D. Xu. A Threat Model Driven Approach for Security Testing. In *Proc. 3rd International Workshop on Software Engineering for Secure Systems (SESS'07)*, pages 10–16. IEEE Computer Society, 2007.
- [159] M.-F. Wendland, M. Kranz, and I. Schieferdecker. A systematic approach to risk-based testing using risk-annotated requirements models. In *Proc. 7th International Conference on Software Engineering Advances (ICSEA'12)*, pages 636–642. IARA, 2012.
- [160] R.J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.
- [161] G. Wimmel and J. Jürjens. Specification-Based Test Generation for Security-Critical Systems Using Mutations. In *Formal Methods and Software Engineering*, pages 471–482. Springer, 2002.
- [162] W.E. Wong, Y. Qi, and K. Cooper. Source code-based software risk assessing. In *Proc. 2005 ACM Symposium on Applied Computing (SAC'05)*, pages 1485–1490. ACM, 2005.
- [163] C. Wysopal, L. Nelson, D.D. Zovi, and E. Dustin. *The Art of Software Security Testing*. Addison-Wesley, 2006.
- [164] D. Xu. A Tool for Automated Test Code Generation from High-Level Petri Nets. In *Applications and Theory of Petri Nets*, pages 308–317. Springer, 2011.
- [165] D. Xu, L. Thomas, M. Kent, T. Mouelhi, and Y. Le Traon. A Model-based Approach to Automated Testing of Access Control Policies. In *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT'12)*, pages 209–218. ACM, 2012.
- [166] D. Xu, M. Tu, M. Sandford, L. Thomas, D. Woodraska, and W. Xu. Automated Security Test Generation with Formal Threat Models. *IEEE Transactions on Dependable and Secure Computing*, 9(4):526–540, 2012.
- [167] R.K. Yin. *Case Study Research: Design and Methods (5th edition)*. SAGE Publications, 2013.
- [168] H. Yoon and B. Choi. A test case prioritization based on degree of risk exposure and its empirical study. *International Journal of Software Engineering and Knowledge Engineering*, 21:191–209, 2011.
- [169] J.W. Young Jr. and H.K. Kent. An Abstract Formulation of Data Processing Problems. In *Proc. 13th National Meeting of the Association for Computing Machinery*, pages 1–4. ACM, 1958.
- [170] E.S.K. Yu and J. Mylopoulos. Using Goals, Rules, and Methods to Support Reasoning in Business Process Reengineering. In *Proc. 27th Hawaii International Conference on System Sciences (HICSS'94)*, pages 234–243. IEEE Computer Society, 1994.

- [171] P. Zech. Risk-Based Security Testing in Cloud Computing Environments. In *Proc. 4th International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 411–414. IEEE Computer Society, 2011.
- [172] P. Zech, M. Felderer, and R. Breu. Towards a Model Based Security Testing Approach of Cloud Computing Environments. In *Proc. 6th International Conference on Software Security and Reliability Companion (SERE-C'12)*, pages 47–56. IEEE Computer Society, 2012.
- [173] M.V. Zelkowitz and D.R. Wallace. Experimental Models for Validating Technology. *Computer*, 31(5):23–31, 1998.
- [174] F. Zimmermann, R. Eschbach, J. Kloos, and T. Bauer. Risk-based statistical testing: A refinement-based approach to the reliability analysis of safety-critical systems. In *Proc. 12th European Workshop on Dependable Computing (EWDC'09)*, pages 1–8. The Open Archive HAL, 2009.
- [175] M. Zulkernine, M.F. Raihan, and M.G. Uddin. Towards Model-Based Automatic Testing of Attack Scenarios. In *Computer Safety, Reliability, and Security*, pages 229–242. Springer, 2009.



**Part II**  
**Research Papers**





Chapter **9**

Paper 1: Approaches for the combined use of risk analysis and testing: a systematic literature review



# Chapter 10

Paper 2: A systematic method for risk-driven test case design using annotated sequence diagrams

---

# Report

## A Systematic Method for Risk-driven Test Case Design Using Annotated Sequence Diagrams

**Author(s)**

Gencer Erdogan, Atle Refsdal, and Ketil Stølen

SINTEF IKT  
SINTEF ICT

Address:  
Postboks 124 Blindern  
NO-0314 Oslo  
NORWAY

Telephone:+47 73593000  
Telefax:+47 22067350

postmottak.IKT@sintef.no  
www.sintef.no  
Enterprise /VAT No:  
NO 948 007 029 MVA

**KEYWORDS:**

Risk-driven testing,  
Risk-based testing,  
Testing,  
Risk analysis,  
Test case,  
Test case design,  
Security testing,  
Security test case

# Report

## A Systematic Method for Risk-driven Test Case Design Using Annotated Sequence Diagrams

**VERSION**

Final

**DATE**

2014-03-24

**AUTHOR(S)**

Gencer Erdogan, Atle Refsdal, and Ketil Stølen

**CLIENT(S)**

Norwegian Research Council

**CLIENT'S REF.**

201579/S10

**PROJECT NO.**

102002253

**NUMBER OF PAGES/APPENDICES:**

36/0

**ABSTRACT**

Risk-driven testing is a testing approach that aims at focusing the testing on the aspects or features of the system under test that are most exposed to risk. Current risk-driven testing approaches succeed in identifying the aspects or features that are most exposed to risks, and thereby support testers in planning the testing process accordingly. However, they fail in supporting testers to employ risk analysis to systematically design test cases. Because of this, there exists a gap between risks, which are often described and understood at a high level of abstraction, and test cases, which are often defined at a low level of abstraction. In this report, we bridge this gap. We give an example-driven presentation of a novel method, intended to assist testers, for systematically designing test cases by making use of risk analysis.

**PREPARED BY**

Gencer Erdogan

SIGNATURE



**CHECKED BY**

Fredrik Seehusen

SIGNATURE



**APPROVED BY**

Bjørn Skjellaug

SIGNATURE



**REPORT NO.**

SINTEF A26036

**ISBN**

978-82-14-05349-4

**CLASSIFICATION**

Unrestricted

**CLASSIFICATION THIS PAGE**

Unrestricted

## Table of Contents

1	Introduction.....	4
2	Overview of Method.....	5
3	Example: Guest Book Application .....	8
4	Step 1: Threat Scenario Identification .....	9
4.1	Identifying Threat Scenarios with Respect to the Integrity of the Guest-Book's Source Code .....	9
4.2	Identifying Threat Scenarios with Respect to the Availability of the Guest Book Entries .....	14
5	Step 2: Threat Scenario Risk Estimation .....	18
5.1	Estimating Risks Posed on the Integrity of the Guest-book's Source Code .....	19
5.2	Estimating Risks Posed on the Availability of the Guest Book Entries .....	23
6	Step 3: Threat Scenario Prioritization .....	26
7	Step 4: Threat Scenario Test Case Design .....	28
8	Related Work .....	32
9	Conclusion .....	33
	Acknowledgments.....	33

## 1 Introduction

Risk-driven testing (or risk-based testing) is a testing approach that use risk analysis within the testing process [5]. The aim in risk-driven testing is to focus the testing process with respect to certain risks of the system under test (SUT).

However, current risk-driven testing approaches leave a gap between risks, which are often described and understood at a high level of abstraction, and test cases, which are often defined at a low level of abstraction. The gap exists because risk analysis, within risk-driven testing approaches, is traditionally used as a basis for planning the test process rather than designing the test cases. Making use of risk analysis when planning the test process helps the tester to focus on the systems, aspects, features, use-cases, etc. that are most exposed to risk, but it does not support test case design. In order to bridge the gap between risks and test cases, risk-driven testing approaches should not merely make use of the risk analysis when planning the test process, but also when designing test cases. Specifically, risk-driven testing approaches must provide testers with steps needed to design test cases by making use of the risk analysis.

In this report, we present a systematic and general method, intended to assist testers, for designing test cases by making use of risk analysis. A test case is a behavioral feature or behavior specifying tests [16]. We employ UML sequence diagrams [15] as the modeling language, conservatively extended with our own notation for representing risk information. In addition, we make use of the UML Testing Profile [16] to specify test cases in sequence diagrams. The reason for choosing sequence diagrams is that they are widely recognized and used within the testing community. In fact, it is among the top three modeling languages applied within the model-based testing community [14]. By annotating sequence diagrams with risk information, we bring risk analysis to the work bench of testers without the burden of a separate risk analysis language, thus reducing the effort needed to adopt the approach. Recent surveys on trends within software testing show that the lack of time and high costs are still the dominating barriers to a successful adoption of testing methods and testing tools within IT organizations [6].

Our method consists of four steps. In Step 1, we analyze the SUT and identify threat scenarios and unwanted incidents with respect to relevant assets. In Step 2, we estimate the likelihood of threat scenarios and unwanted incidents, as well as the consequence of unwanted incidents. In Step 3, we prioritize and select paths which consist of sequences of threat scenarios leading up to and including a risk. In Step 4, we design test cases with respect to the paths selected for testing.

Section 2 gives an overview of our method. Section 3 introduces the web application on which we apply our method to demonstrate its applicability. Sections 4, 5, 6 and 7 employ the four steps on the web application, respectively. Section 8 relates our method to current risk-driven testing approaches that also address test case design. Finally, we provide concluding remarks in Sect. 9.



## 2 Overview of Method

Before going into the details of our method, we explain the assumed context in which it is to be applied. A testing process starts with *test planning*, followed by *test design and implementation*, *test environment set-up and maintenance*, *test execution*, and finally *test incident reporting* [10]. Our method starts after test planning, but before test design and implementation. Furthermore, the first and the fourth step in our method expect as input a description of the SUT in terms of sequence diagrams and suspension criteria, respectively. Suspension criteria are used to stop all or a portion of the testing activities [9]. This is also known as test stopping criteria or exit criteria. Suspension criteria are used in our method to reflect the investable testing effort. We assume that these inputs are obtained during test planning. Next, we assume that the preparations for carrying out risk analysis have been completed, i.e., that assets have been identified, likelihood and consequence scales have been defined, and a risk evaluation matrix has been prepared with respect to the likelihood and consequence scales. Our method consists of four main steps as illustrated in Fig. 1; dashed document icons represent input prepared during test planning, solid document icons represent output from one step and acts as input to the following step.

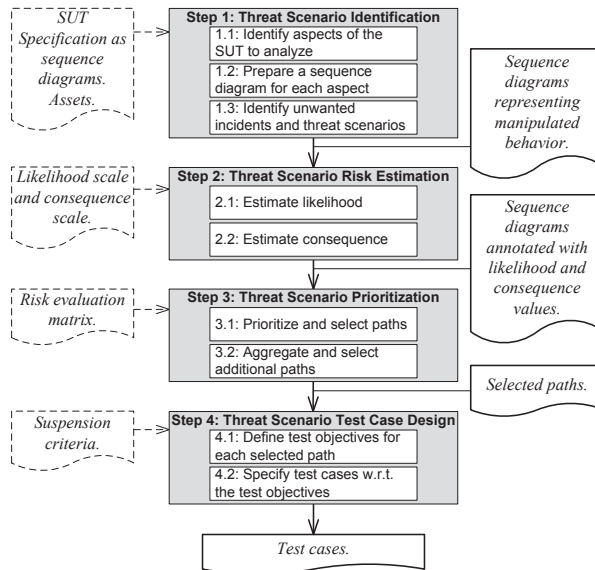


Fig. 1. Overview of the steps in the method.

In **Step 1**, we analyze the SUT to identify unwanted incidents with respect to a certain asset to be protected, as well as threat scenarios resulting from manipulations initiated by the threat. This step expects as input a sequence diagram specification of the SUT and the asset that is to be considered. **First**, we identify the aspects of the SUT we are interested in analyzing. We then annotate each aspect with a label, containing a unique identifier. **Second**, we prepare a corresponding sequence diagram to capture risk information for each aspect label. Each sequence diagram inherits the SUT specification encapsulated by the underlying aspect label. Additionally, it represents the asset as a lifeline. The threats that may initiate threat scenarios are also represented as lifelines. **Third**, we identify unwanted incidents that have an impact on the asset, and threat scenarios that may lead to the unwanted incidents. The output of this step is a set of annotated sequence diagrams that represent manipulated behavior of the SUT and its context, in terms of threat scenarios and unwanted incidents.

In **Step 2**, we estimate the likelihood for the occurrence of the threat scenarios and the unwanted incidents in terms of frequencies, the conditional probability for threat scenarios leading to other threat scenarios or to unwanted incidents, as well as the impact of unwanted incidents on the asset. The input for this step is the output of Step 1. Additionally, this step expects a predefined likelihood scale in terms of frequencies, and a predefined consequence scale in terms of impact on the asset. **First**, we estimate the likelihood for the occurrence of the threat scenarios and the unwanted incidents using the likelihood scale, as well as the conditional probability for threat scenarios leading to other threat scenarios or to unwanted incidents. **Second**, we estimate the consequence of unwanted incidents using the consequence scale. The output of this step is the same set of sequence diagrams given as the input for the step, annotated with likelihood estimates and consequence estimates as described above. A risk in our method is represented by an unwanted incident (i.e., a message to the asset lifeline) together with its likelihood value and its consequence value. A sequence of threat scenarios may lead up to one or more risks. Additionally, different sequences of threat scenarios may lead up to the same risk. We refer to a sequence of threat scenarios leading up to and including a risk as a *path*.









In **Step 3**, we prioritize and select paths for testing. The input for this step is the output of Step 2. Additionally, this step employs the predefined risk evaluation matrix. **First**, we prioritize the paths by mapping them to the risk evaluation matrix based on the likelihood (frequency) value and the consequence (impact) value of the risk included in the paths. We then select the paths based on their risk level, i.e., their position in the risk evaluation matrix. **Second**, we aggregate similar risks in different paths in order to evaluate whether to select additional paths for testing. The output of this step is a set of paths selected for testing.

In **Step 4**, we define test objectives for each path selected for testing, and then we specify test cases with respect to the test objectives. A test objective is a textual specification of a well-defined target of testing, focusing on a single requirement or a set of related requirements as specified in the specification of

the system under test [16]. A test objective merely describes what (logic) needs to be tested or how the system under test is expected to react to particular stimuli [16]. The input for this step is the output of Step 1 and the output of Step 3. Additionally, this step expects predefined suspension criteria. **First**, we define one or more test objectives for each path selected for testing. A path may have one or more test objectives, but one test objective is defined only for one path. We use one test objective as a basis for specifying one test case. **Second**, we specify a test case by first identifying the necessary interaction in the relevant path. By necessary interaction, we mean the interaction that is necessary in order to fulfill the test objective. Then, we copy the necessary interaction into a new sequence diagram. Finally, we annotate the new sequence diagram, with respect to the test objective, using the UML Testing Profile [16]. We continue designing test cases in this manner with respect to the predefined suspension criteria. The output of this step is a set of sequence diagrams representing test cases.

Table 1 shows the notation for annotating sequence diagrams with risk information. We have mapped some risk information to corresponding UML constructs for sequence diagrams. Assets and threats are represented as lifelines. Inspired by CORAS [12], we distinguish between three types of threats; deliberate threats (the leftmost lifeline in the Notation column), accidental threats (the center lifeline in the Notation column) and non-human threats (the rightmost lifeline in the Notation column). Manipulations and unwanted incidents are represented as messages. We distinguish between three types of manipulations; new messages in the sequence diagram (a message annotated with a filled triangle), alteration of existing messages in the sequence diagram (a message annotated with an unfilled triangle), and deletion of existing messages in the sequence diagram (a message annotated with a cross inside a triangle). Aspect labels, likelihoods, conditional probabilities and consequences do not have corresponding UML constructs for sequence diagrams. However, the following constraints apply: A likelihood can only be attached horizontally across lifelines. A likelihood assignment represents the likelihood, in terms of frequency, of the interaction preceding the likelihood assignment. The purpose of messages representing unwanted incidents is to denote that an unwanted incident has an impact on an asset. A consequence can therefore only be attached on messages representing unwanted incidents. A conditional probability may be attached on any kind of message except messages representing unwanted incidents. A conditional probability assignment represents the probability of the occurrence of the message on which it is assigned, given that the interaction preceding the message has occurred.

**Table 1.** Notation for annotating sequence diagrams with risk information.

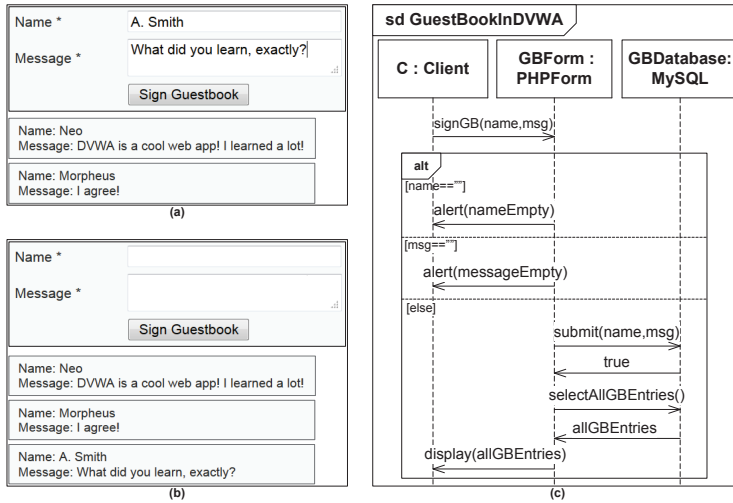
Risk information	UML construct	Notation
Aspect label	N/A	
Asset	Lifeline	
Threat	Lifeline	
Manipulation	Message	
Unwanted incident	Message	
Likelihood	N/A	
Conditional probability	N/A	
Consequence	N/A	

### 3 Example: Guest Book Application

As mentioned in Sect. 1, our method is a general method for designing test cases by making use of risk analysis. In this demonstration, we focus on security, and apply the steps presented in Sect. 2 on a guest book that is available in the Damn Vulnerable Web Application (DVWA) [4]. One of DVWA’s main goals is to be an aid for security professionals to test their skills and tools in a legal environment [4]. DVWA is programmed in the scripting language PHP and requires a dedicated MySQL server to function correctly. We are running DVWA version 1.8 on the HTTP server XAMPP version 1.8.2 [26], which provides the required execution environment.

The SUT in this demonstration is a guest book in DVWA. Figure 2a shows a screenshot of the guest book user interface before a guest book entry is submitted, while Fig. 2b shows a screenshot of the user interface after the guest book entry is successfully submitted. Figure 2c represents its behavioral specification expressed as a sequence diagram. A guest book user may use a web browser in a client to sign the guest book by typing a name and a message, and then submit the guest book entry by clicking the “Sign Guestbook” button. If the *name* input field is empty, the guest book form replies with a warning message. If the *name* input field is not empty, but the *message* input field is empty, the guest book form also replies with a warning message. If neither of the input fields are empty, the guest book form submits the entry to the guest book database. The guest book database stores the entry and replies with the message *true* indicating that the transaction was successful. Having received the message *true*, the guest book

form retrieves all of the guest book entries from the database, including the one just submitted, and displays them to the client.



**Fig. 2.** (a) Screenshot of the guest book before submitting a new entry. (b) Screenshot of the guest book after submitting the entry. (c) Specification of the guest book expressed as a sequence diagram.

## 4 Step 1: Threat Scenario Identification

The SUT in this demonstration is the guest book explained in Sect. 3. Let us assume that we are interested in analyzing the guest book with respect to the following two security assets:

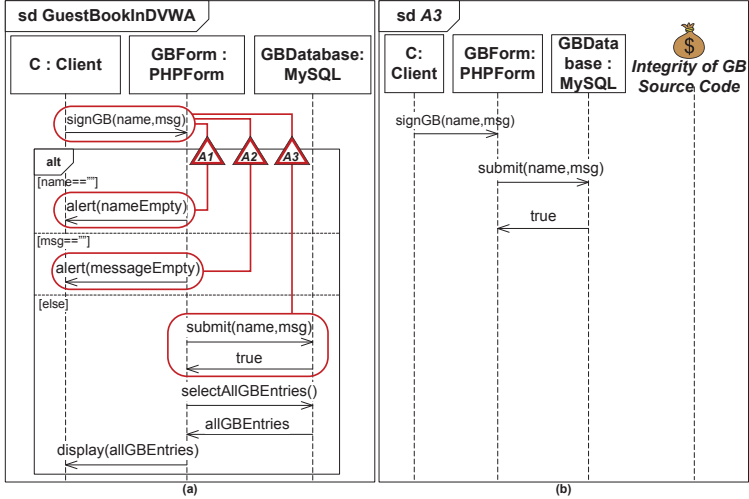
- *Integrity of the guest-book’s source code.*
- *Availability of the guest book entries.*

In Sect. 4.1, we identify threat scenarios with respect to the integrity of the guest-book’s source code, while in Sect. 4.2, we identify threat scenarios with respect to the availability of the guest book entries.

### 4.1 Identifying Threat Scenarios with Respect to the Integrity of the Guest-Book’s Source Code

As shown in Fig. 3a, we have identified three aspects labeled with aspect labels A1, A2 and A3. For the aspect represented by aspect label A1, we are interested

in analyzing the interaction composed of the messages  $signGB(name, msg)$  and  $alert(nameEmpty)$ , with respect to the integrity of the guest-book’s source code. The same reasoning applies for A2 and A3. The aspects identified in this example are small. In practice it may well be that one is interested in analyzing bigger and more complex aspects. The granularity level of an aspect is determined by the tester.



**Fig. 3.** (a) Specification of the guest book annotated with aspect labels. (b) Corresponding sequence diagram of the aspect encapsulated by aspect label A3, which also shows the security asset *integrity of the guest-book’s source code* as a lifeline.

Suppose we are only interested in analyzing the aspect encapsulated by aspect label A3. Figure 3b shows a sequence diagram corresponding to the interaction encapsulated by aspect label A3. Additionally, it represents the abovementioned security asset as a lifeline. We now have a sequence diagram we can use as a starting point to analyze the SUT aspect encapsulated by aspect label A3, with respect to integrity of the guest-book’s source code. We represent the risk related information in bold and italic font, in the sequence diagrams, to distinguish between the specification and the risk related information.

We proceed the analysis by identifying unwanted incidents that may have an impact on the security asset, and threat scenarios that may lead to the unwanted incidents. The integrity of the guest-book’s source code may be compromised if, for example, a malicious script is successfully stored (i.e., injected) in the guest book database. A malicious script that is injected in the guest book database is

executed by the web browser of the guest book user when accessed. This modifies the content of the HTML page on the user's web browser, thus compromising the integrity of the guest-book's source code. These kinds of script injections are also known as stored cross-site scripting (stored XSS) [18]. We identify the occurrence of an XSS script injection on the guest book database as an unwanted incident (*UII*), as represented by the last message in Fig. 4. An XSS script is successfully injected in the guest book database only if the database successfully carries out the transaction containing the XSS script. This is why *UII* occurs after the occurrence of message *true* on lifeline GBDatabase.

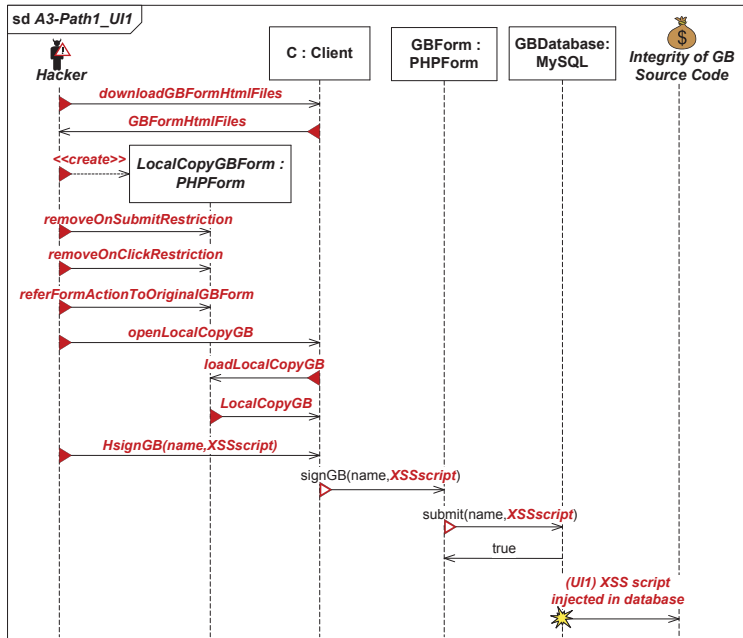


Fig. 4. Identifying a first path in which unwanted incident *UII* may occur.

*UII* may be caused by different sequences of threat scenarios that manipulates the expected behavior of the guest book. Recall that we refer to a sequence of threat scenarios leading up to and including a risk as a path. In each of the sequence diagrams in Figs. 4, 5, and 6, we identify a different path in which *UII* may occur.

The first path in which *UII* may occur, i.e., the sequence diagram in Fig. 4, shows that *UII* may occur if the *msg* parameter in messages *signGB(name, msg)* and *submit(name, msg)* is replaced with *XSSscript*, representing an XSS script. This is an alteration of the guest-book’s expected behavior. We therefore replace the messages *signGB(name, msg)* and *submit(name, msg)* with messages representing alterations.

These alterations may be initiated by different threats. Let us say we are interested in analyzing this further from a hacker perspective, which is categorized as a deliberate threat. A hacker may successfully carry out an XSS script injection by, for example, first downloading the HTML files of the guest book using the web browser, in order to create a local copy of the guest-book’s user interface (*downloadGBFormHtmlFiles*, *GBFormHtmlFiles* and *<<create>>*). Having successfully saved a local copy of the guest-book’s HTML files, the hacker removes all restrictions, such as the maximum number of characters allowed in the name and message input fields when submitting a guest book entry (*removeOnSubmitRestriction* and *removeOnClickRestriction*). Then, the hacker refers all actions to the original guest book by making use of its web address (*referFormActionToOriginalGBForm*). Finally, the hacker loads the local copy of the guest book in the web browser, writes an XSS script in the message field, and submits the guest book entry containing the XSS script (*openLocalCopyGB*, *loadLocalCopyGB*, *LocalCopyGB* and *HsignGB(name, XSSscript)*). Note that all of the messages described in this paragraph are annotated as new messages in the sequence diagram (message with a filled triangle).

The second path in which *UII* may occur, i.e., the sequence diagram in Fig. 5, also shows that *UII* may be caused by replacing the *msg* parameter in messages *signGB(name, msg)* and *submit(name, msg)* with *XSSscript*. However, we also see that the second path has some threat scenarios different from the first path, thus representing a different path in which *UII* may occur.

In the second path, we first assume that the hacker gathers information about the setup of the URLs that are sent from a client to the guest book form. The hacker exploits this information to prepare a valid URL that contains an XSS script and that targets the guest book form. The process of preparing URLs in this way is also known as URL forging. This is commonly carried out by hackers, or other malicious users, with the objective to force legitimate users of a web application to execute actions on their behalf. These kinds of attacks are known as cross-site request forgery attacks [19]. Having successfully forged the URL containing the XSS script, the hacker sends it to a legitimate user of the guest book (*forgedURLReplacingMsgWithXSSscript*). We choose not to model how the hacker forges the URL and by what means the hacker sends the forged URL. The assumption is that a hacker successfully forges a URL capable of injecting an XSS script into the guest book database, and that the URL is successfully sent to a legitimate user of the guest book.

Having received the URL, the legitimate user executes it via the web browser of the client (*executeForgedURL*). Consequently, this results in the execution



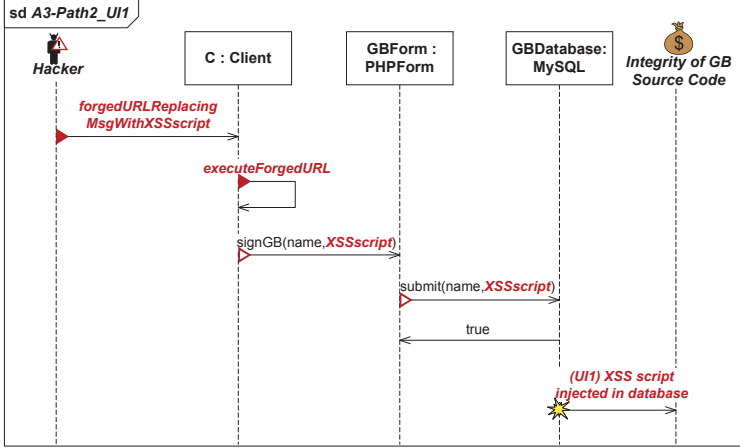


Fig. 5. Identifying a second path in which *UII* may occur.

of the messages  $signGB(name, XSSscript)$ ,  $submit(name, XSSscript)$  and  $true$ , which finally leads to the occurrence of *UII*.

In the case of the third path, i.e., the sequence diagram in Fig. 6, we assume that the hacker is able to intercept the HTTPS connection between the client and the guest book form using a proxy tool by, for example, following the guidelines explained in [1]. The hacker first configures the tool to automatically inject an XSS script in a certain part of the HTTPS request sent to the guest book form ( $\ll create \gg$  and  $configureAutoInjectXSSscriptInMsgInHTTPSrequest$ ). Then, the hacker starts the interception feature of the tool for intercepting the HTTPS request between the client and the guest book form ( $interceptClientHTTPSrequest$  and  $interceptHTTPSrequest$ ). The consequence of intercepting the HTTPS requests sent from the client is the redirection of message  $signGB(name, msg)$  to the proxy tool. The redirection of message  $signGB(name, msg)$  is an alteration of the expected behavior of the guest book. Thus, we replace message  $signGB(name, msg)$  with a message representing an alteration.

Having successfully intercepted the HTTPS request sent from the client, the proxy tool automatically injects the XSS script into the HTTPS request ( $injectXSSscriptInMsg$ ). Then, the proxy tool sends the HTTPS request containing the XSS script to the guest book form ( $PTsignGB(name, XSSscript)$ ). Consequently, this results in the execution of the messages  $submit(name, XSSscript)$  and  $true$  as in the first and the second path, which finally leads to the occurrence of *UII*.

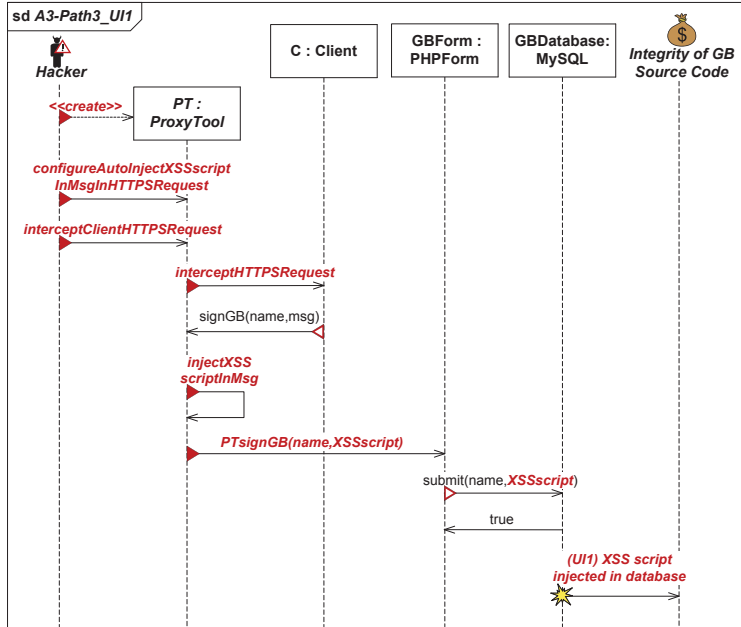
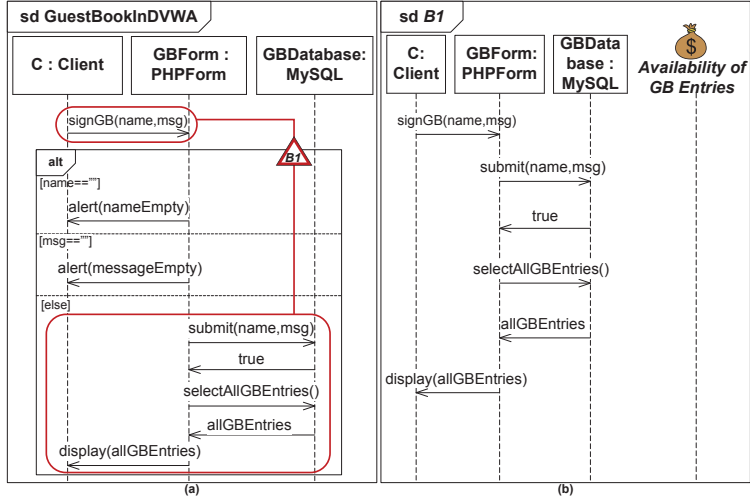


Fig. 6. Identifying a third path in which *UI1* may occur.

#### 4.2 Identifying Threat Scenarios with Respect to the Availability of the Guest Book Entries

The identification of threat scenarios, with respect to the availability of the guest book entries, is carried out in a similar manner as explained in Sect. 4.1. As shown in Fig. 7a, we have identified one aspect labeled with aspect label B1. In this case, we are interested in analyzing the interaction composed of messages  $signGB(name,msg)$ ,  $submit(name,msg)$ ,  $true$ ,  $selectAllGBEntries()$ ,  $allGBEntries$  and  $display(allGBEntries)$ , with respect to the availability of the guest book entries. Figure 7b shows the sequence diagram corresponding to the interaction encapsulated by aspect label B1. We use the sequence diagram in Fig. 7b as a starting point for analyzing the SUT aspect encapsulated by aspect label B1, with respect to the availability of the guest book entries.

The availability of the guest book entries is compromised if, for example, the guest book entries in the guest book database are somehow deleted. The guest book entries may be deleted by executing an SQL query, on the guest book



**Fig. 7.** (a) Specification of the guest book annotated with an aspect label. (b) Corresponding sequence diagram of the aspect encapsulated by aspect label B1, which also shows the security asset *availability of the guest book entries* as a lifeline.

database, which is constructed for deleting the guest book entries. Such SQL queries may be executed, for example, by submitting the queries to the database via the guest book form. This way of executing SQL queries is known as SQL injections. We identify this as unwanted incident *UI2*, as shown by message (*UI2*) *GB entries deleted due to SQL injection* in Fig. 8.

An SQL injection may be caused, from a hacker perspective, based on a similar path as the one presented in Fig. 4. The difference between the path in Fig. 4 and the path in Fig. 8 is that the hacker initiates an SQL injection (*HSignGB(name,SQLinjection)*), and that the *msg* parameter in messages *signGB(name, msg)* and *submit(name,msg)* in Fig. 8 is replaced with *SQLinjection*, representing an SQL query constructed for deleting guest book entries. Additionally, we see from Fig. 8 that the occurrence of unwanted incident *UI2* leads to some additional manipulations of the expected behavior of the guest book.

Given that unwanted incident *UI2* occurs, the guest book database no longer contains any guest book entries. However, having requested all guest book entries from the database (*selectAllGBEntries()*), the guest book form expects guest book entries. Naturally, the database does not return any guest book entries because there are none (*noGBEntries*), which in turn leads the guest book form not to display any guest book entries (*display(noGBEntries)*). These are al-

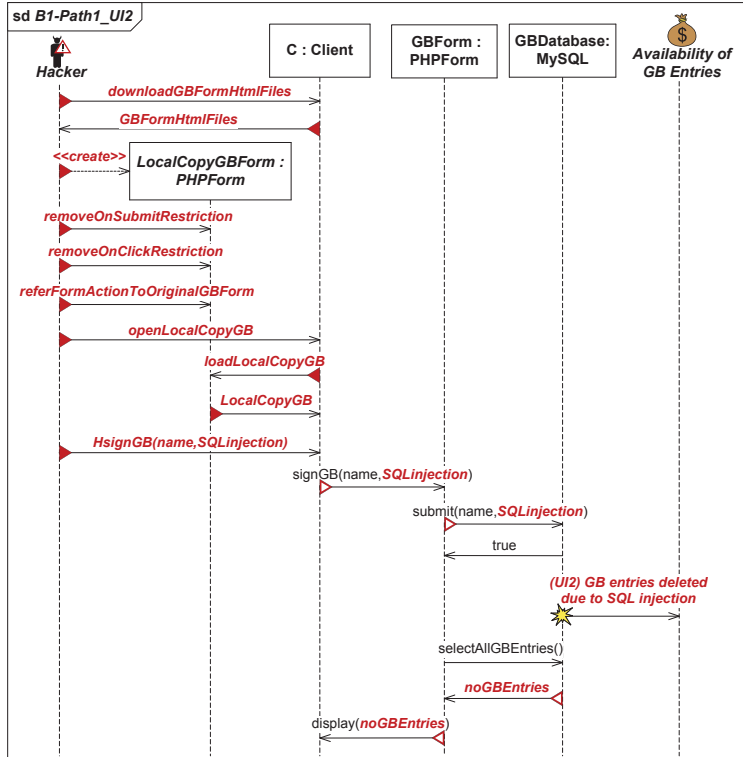


Fig. 8. Identifying a first path in which *UI2* may occur.

terations of the expected behavior of the guest book, as a result of the occurrence of *UI2*, and are therefore shown as messages representing alterations.

A second example of an unwanted incident, that compromises the availability of the guest book entries, is the deletion of the guest book entries before it reaches the client expecting them. This may be achieved by, for example, first intercepting the HTTPS response transmitted from the guest book form, and then deleting the guest book entries situated inside the captured HTTPS response. We identify this as unwanted incident *UI3*, as represented by message *(UI3) GB entries deleted by intercepting HTTPS response* in Fig. 9.

Similar to the third path in which *UI1* occurs, we assume that a hacker uses a proxy tool for intercepting the HTTPS connection between the guest book form and the client. The hacker first configures the tool for automatically deleting

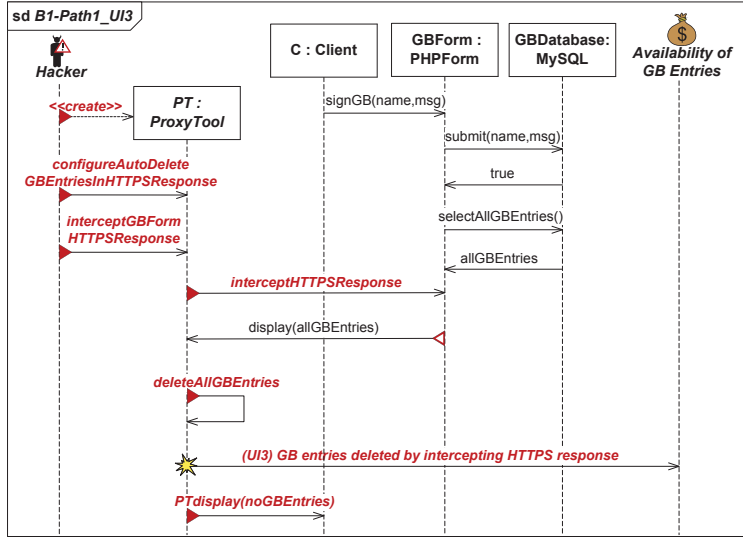


Fig. 9. Identifying a first path in which *UI3* may occur.

the guest book entries situated inside the HTTPS responses (*<<create>>* and *configureAutoDeleteGBEntriesInHTTPSResponse*). Then, the hacker starts the interception feature of the tool for intercepting the HTTPS response between the guest book form and the client (*interceptGBFormHTTPSResponse* and *interceptHTTPSResponse*). The consequence of intercepting the HTTPS responses sent from the guest book form is the redirection of message *display(allGBEntries)* to the proxy tool. The redirection of message *display(allGBEntries)* is an alteration of the expected behavior of the guest book. Thus, we replace message *display(allGBEntries)* with a message representing an alteration.

Having successfully intercepted the HTTPS response from the guest book form, the proxy tool automatically deletes all guest book entries situated in the HTTPS response (*deleteAllGBEntries*). This leads to the occurrence of unwanted incident *UI3*. Finally, the proxy tool sends the altered HTTPS response containing no guest book entries to the client (*PTdisplay(noGBEntries)*).

## 5 Step 2: Threat Scenario Risk Estimation

Table 2 shows the likelihood scale that we assume has been established during preparation of the risk analysis. The likelihood scale is given in terms of frequency intervals.

**Table 2.** Likelihood scale.

Likelihood	Description
Rare	[0, 10>:1y Zero to less than ten times per year
Unlikely	[10, 50>:1y Ten to less than fifty times per year
Possible	[50, 150>:1y Fifty to less than one hundred and fifty times per year
Likely	[150, 300>:1y One hundred and fifty to less than three hundred times per year
Certain	[300, ∞>:1y Three hundred times or more per year

In practice, it is common to use one likelihood scale when estimating the likelihood for the occurrence of threat scenarios and unwanted incidents. It is also possible to use one consequence scale, when estimating the consequence unwanted incidents have on certain assets. However, this may be difficult and impractical because the consequence unwanted incidents have on different assets may be difficult to measure by the same means. As mentioned in Sect. 4, we consider two different assets in this demonstration, namely the *integrity of the guest-book's source code* and the *availability of the guest book entries*.

Table 3 shows the consequence scale for security asset *integrity of the guest-book's source code*. The consequence scale in Table 3 is given in terms of impact on the integrity of certain categories of the guest-book's source code. For example, an unwanted incident has a catastrophic impact on the security asset if it compromises the integrity of the guest-book's source code that carries out database transactions. Similar interpretations apply for the other consequences in Table 3. Table 4, on the other hand, shows the consequence scale for security asset *availability of the guest book entries*. The consequence scale in Table 4 is given in terms of impact on the availability of the guest book entries. For example, an unwanted incident has a catastrophic impact on the security asset if it makes the guest book entries unavailable for one week or more. Similar interpretations apply for the other consequences in Table 4. We assume that these consequence scales have been established during preparation of the risk analysis.

In Sects. 5.1 and 5.2, we make use of Table 2 for estimating the likelihood for the occurrence of threat scenarios and unwanted incidents. When estimating the consequence unwanted incidents have on the security assets, however, we make use of the consequence scale addressing the asset under consideration. That is, in Sect. 5.1 we use Table 3 and in Sect. 5.2 we use Table 4.

**Table 3.** Consequence scale for security asset *integrity of the guest-book’s source code*.

Consequence	Description
Insignificant	The integrity of the source code that generates the aesthetics is compromised
Minor	The integrity of the source code that retrieves third party ads is compromised
Moderate	The integrity of the source code that generates the user interface is compromised
Major	The integrity of the source code that manages sessions and cookies is compromised
Catastrophic	The integrity of the source code that carries out database transactions is compromised

**Table 4.** Consequence scale for security asset *availability of the guest book entries*.

Consequence	Description
Insignificant	Guest book entries are unavailable in range [0, 1 minute>
Minor	Guest book entries are unavailable in range [1 minute, 1 hour>
Moderate	Guest book entries are unavailable in range [1 hour, 1 day>
Major	Guest book entries are unavailable in range [1 day, 1 week>
Catastrophic	Guest book entries are unavailable in range [1 week, ∞>

### 5.1 Estimating Risks Posed on the Integrity of the Guest-book’s Source Code

Figure 10 shows likelihood estimates for the first path in which unwanted incident *UII* occurs, as well as a consequence estimate for *UII*. The tester may estimate likelihood values and consequence values based on expert judgment, statistical data, a combination of both, etc. Let us say we have acquired information indicating that hackers most likely prepare injection attacks in the manner described by the interaction starting with message *downloadGBFormHtmlFiles*, and ending with message *LocalCopyGB* in Fig. 10. For this reason, we choose to assign likelihood Likely on this interaction. Note that Likely corresponds to the frequency interval [150, 300>:1y (see Table 2).

XSS script injection attacks are less likely to be initiated by hackers compared to other kinds of injection attacks they initiate (such as SQL-injection attacks) [22]. For this reason, we choose to assign a probability 0.8 on message *HsignGB(name,XSSscript)*, indicating that it will occur with probability 0.8 given that the messages preceding it has occurred. This probability assignment leads to a different frequency interval for the interaction starting with message *downloadGBFormHtmlFiles* and ending with message *HsignGB(name,XSSscript)*. The frequency interval for the aforementioned interaction is calculated by multiplying [150, 300>:1y with 0.8, which results in the frequency interval [120, 240>:1y. This frequency interval is in turn used to calculate the subsequent

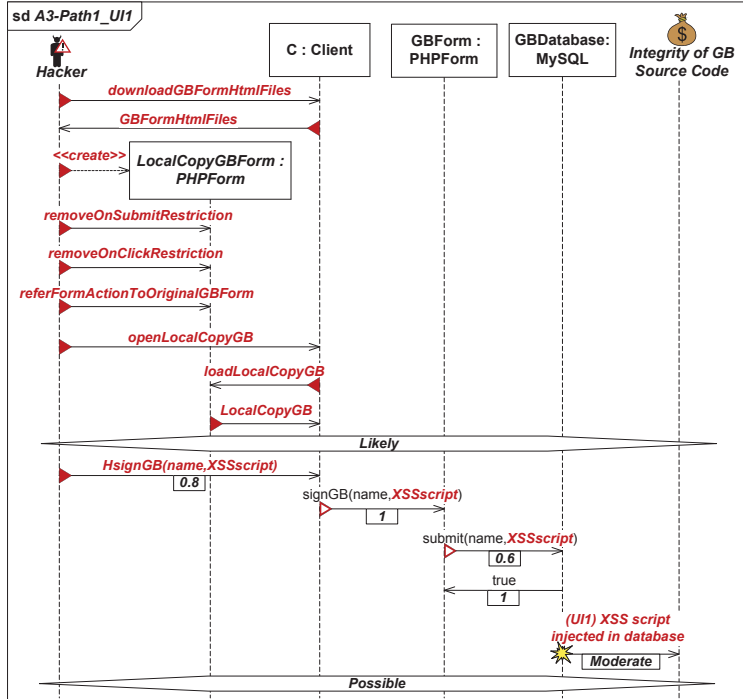


Fig. 10. Estimating the likelihood of the first path in which unwanted incident *UI1* occurs, as well as the consequence of *UI1*.

frequency interval, in the path, in a similar manner. This procedure is carried out until the frequency interval for the whole path is calculated. The frequency interval for the whole path is then mapped to the likelihood scale in Table 2 in order to deduce a likelihood value. The deduced likelihood value represents the likelihood value for the whole path, and thereby the likelihood value for the unwanted incident included in the path.

We proceed the estimation by identifying conditional probabilities for the remaining messages. We assume message  $signGB(name, XSSscript)$  will occur with probability 1 since the hacker has removed all restrictions on the local copy of the guest book form. The guest book form is programmed in the scripting language PHP. Although PHP makes use of what is known as “prepared statements” to validate input directed to the database, bypassing the validation is still possible if the prepared statements are not handled correctly [23]. These kinds of bypasses require insight into the structure of the source code and are



therefore harder to exploit. For this reason, we choose to assign a probability  $0.6$  on message  $submit(name, XSSscript)$ . We assume message  $true$  will occur with probability  $1$ , as there is nothing that prevents the database from executing the query containing the XSS script if it has made all its way into the database.

We calculate the frequency interval for the whole path by multiplying  $[150, 300>:1y$  with the product of the abovementioned conditional probabilities. That is, we multiply  $[150, 300>:1y$  with  $0.48$ , which results in the frequency interval  $[72, 144>:1y$ . By mapping this frequency interval to the likelihood scale in Table 2, we see that the frequency interval is within the boundaries of likelihood Possible. This means that the path represented by the sequence diagram in Fig. 10, and thereby unwanted incident  $UII$ , may occur with likelihood Possible. Finally, an XSS script injected in the database has the objective to execute a script on the end user’s web browser for different purposes. This means that the injected XSS script modifies the source code that generates the user interface. Thus,  $UII$  has an impact on the security asset with a moderate consequence.

Figure 11 shows likelihood estimates for the second path in which  $UII$  occurs. As explained in Sect. 4.1, the second path shows an example of how the hacker may inject an XSS script in the guest book database by performing a cross-site request forgery attack. The detection of whether a web application is vulnerable to cross-site request forgery attacks is easy [19], and thus an attack hackers most likely will try to exploit. Based on this, we assign likelihood Likely on the interaction composed of message  $forgedURLReplacingMsgWithXSSscript$ . However, the increased awareness of cross-site request forgery attacks has, in turn, brought about an increased usage of countermeasures preventing successful execution of such attacks [21]. For this reason, we choose to assign a probability  $0.5$  on message  $executeForgedURL$ .

Given that the forged URL is successfully executed, then there is nothing preventing the client in submitting the guest book entry containing the XSS script ( $signGB(name, XSSscript)$ ). The probability for the occurrence of message  $signGB(name, XSSscript)$  is therefore  $1$ . The probability for the occurrence of messages  $submit(name, XSSscript)$  and  $true$  is  $0.6$  and  $1$ , respectively, for the same reason as given for the first path. The likelihood value for the second path is calculated in a similar way as explained for the first path. That is, we multiply the frequency interval  $[150, 300>:1y$  (likelihood Likely) with the product of the conditional probabilities assigned on the messages succeeding the likelihood assignment in the path, which in this case is  $0.3$ . This results in the frequency interval  $[45, 90>:1y$ . By mapping this frequency interval to the likelihood scale in Table 2, we see that it overlaps Unlikely and Possible. However, we also see that the frequency interval is skewed more towards Possible than Unlikely. For this reason, we choose to assign Possible on the second path, which means that  $UII$  may occur with likelihood Possible in the second path. If a frequency interval overlaps several likelihood values, as it does for the second path, then the tester has to decide on which likelihood value to assign. In this demonstration, we decide to assign a likelihood value by analyzing the skewness of the frequency

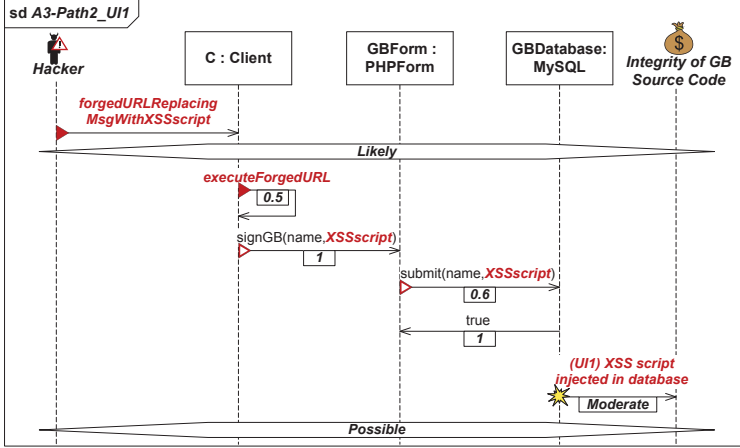


Fig. 11. Estimating the likelihood of the second path in which *UI1* occurs.

interval. Such decisions may vary from situation to situation and has to be made and justified by the tester.

Figure 12 shows likelihood estimates for the third path in which *UI1* occurs. Intercepting HTTPS connections is possible and in some situations easy to carry out [1]. However, the exploitability of vulnerabilities in encrypted communication protocols, such as HTTPS, is difficult on a large scale [20]. For this reason, we choose to assign likelihood Possible on the interaction starting with message `<<create>>` and ending with message `interceptHTTPSRequest`.

Let us, for the sake of the example, assume that the guest book is making use of proper countermeasures, e.g., as presented in [17], in order to significantly mitigate the possibility for successful HTTPS interceptions. Assuming this, we choose to assign a probability  $0.2$  on message `signGB(name,msg)`. If the HTTPS connection between the client and the guest book form is successfully intercepted, however, the proxy tool injects the `msg` parameter of message `signGB(name, msg)` with an XSS script (`injectXSSscriptInMsg`). Then, the proxy tool sends the guest book entry containing the XSS script to the guest book form (`PTsignGB(name,XSSscript)`). The probability for the occurrence of messages `injectXSSscriptInMsg` and `PTsignGB(name,XSSscript)` is therefore  $1$ . The probability for the occurrence of messages `submit(name,XSSscript)` and `true` is  $0.6$  and  $1$ , respectively, for the same reasons as given for the first path.

We calculate the likelihood value for the third path as explained for the first and the second path. That is, we multiply frequency interval  $[50, 150]:1y$  (likelihood Possible) with the product of the conditional probabilities assigned on the messages succeeding the likelihood assignment in the path, which in this

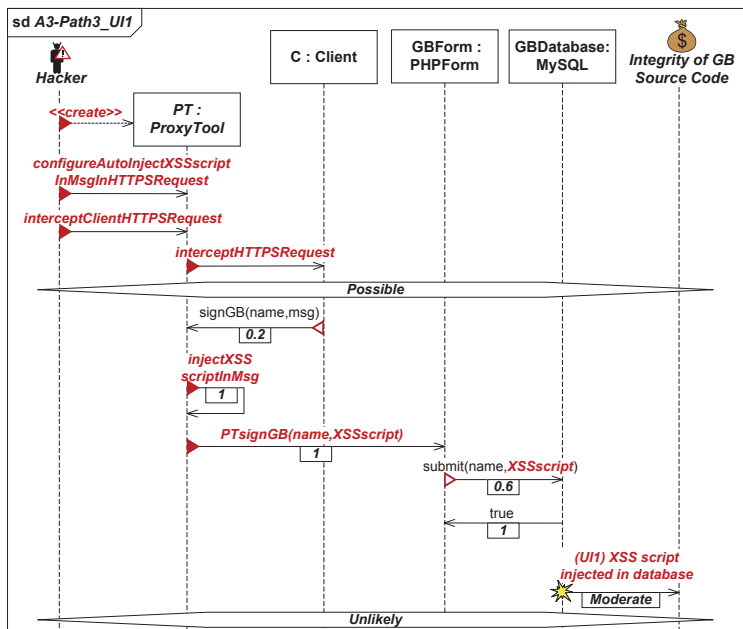


Fig. 12. Estimating the likelihood of the third path in which *UI1* occurs.

case is  $0.12$ . This results in the frequency interval  $[6, 18]:1y$ . We map this frequency interval to the likelihood scale in Table 2, and see that it is skewed more towards Unlikely than Rare. Based on this, we choose to assign likelihood Unlikely on the third path. Hence, *UI1* occurs with likelihood Unlikely in the third path.

## 5.2 Estimating Risks Posed on the Availability of the Guest Book Entries

As pointed out in Sect. 4.2, the path in which *UI2* occurs (see Fig. 13) is similar to the first path in which *UI1* occurs (see Fig. 10). In fact, the interaction starting with message *downloadGBFormHtmlFiles* and ending with message *LocalCopyGB* is identical in both paths. As shown in Fig. 10, we assigned likelihood Likely on the aforementioned interaction. Given that the aforementioned interaction is identical in both paths, we also assign likelihood Likely on the same interaction in the path where *UI2* occurs.

Hackers performing injection attacks will most likely carry out SQL-injection attacks [22]. We therefore choose to assign probability  $1$  on message *HsignGB*(

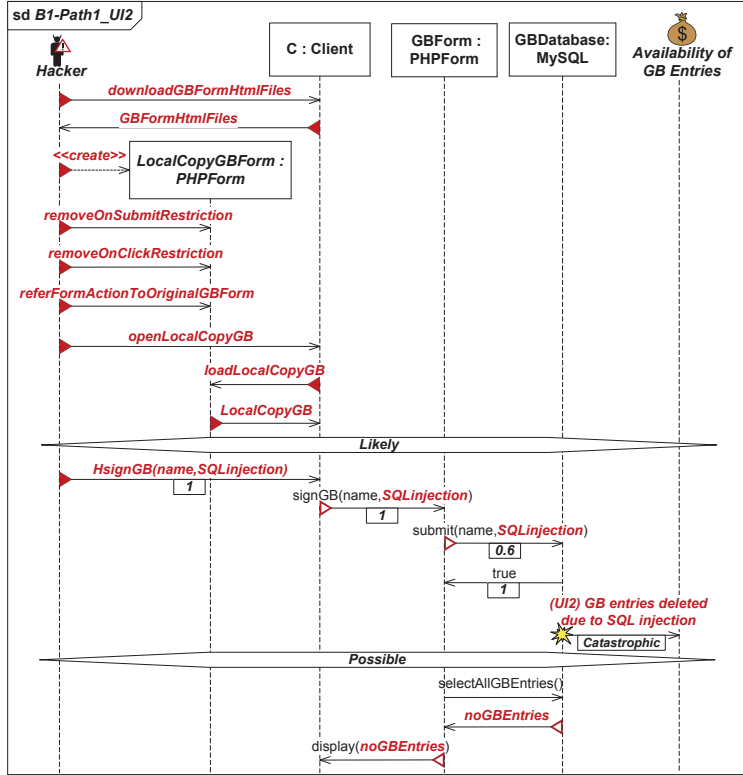


Fig. 13. Estimating the likelihood of the path in which *UI2* occurs, as well as the consequence of *UI2*.

*name, SQLinjection*). The probability for the occurrence of messages *signGB(name, SQLinjection)*, *submit(name, SQLinjection)* and *true* is 1, 0.6 and 1, respectively. The justification for assigning these three probability values is the same as the justification given for messages *signGB(name, XSSscript)*, *submit(name, XSSscript)* and *true* in the path shown in Fig. 10. Based on these conditional probabilities and likelihood Likely, we calculate the frequency interval for the whole path, i.e., the frequency interval for the occurrence of *UI2*, in a similar manner as explained throughout Sect. 5.1. The frequency interval for the occurrence of *UI2* is  $[90, 180]:1y$ , from which we have deduced likelihood Possible as shown in Fig. 13. Finally, the occurrence of *UI2* implies that the guest book entries in the database are deleted. Since the guest book in this demon-

stration does not have any mechanisms for creating a backup of the guest book entries, the deleted guest book entries will most likely never be available again. Thus, *UI2* has an impact on the security asset with a catastrophic consequence.

Figure 14 shows likelihood estimates for the path in which *UI3* occurs, as well as a consequence estimate for *UI3*. Similar to the third path in which *UI1* occurs (see Fig. 12), we assume that a hacker uses a proxy tool for intercepting the HTTPS connection between the client and the guest book form. Based on the same justification given for the third path where *UI1* occurs, we assign likelihood Possible on the interaction starting with message `<<create>>` and ending with message `interceptHTTPSResponse`.

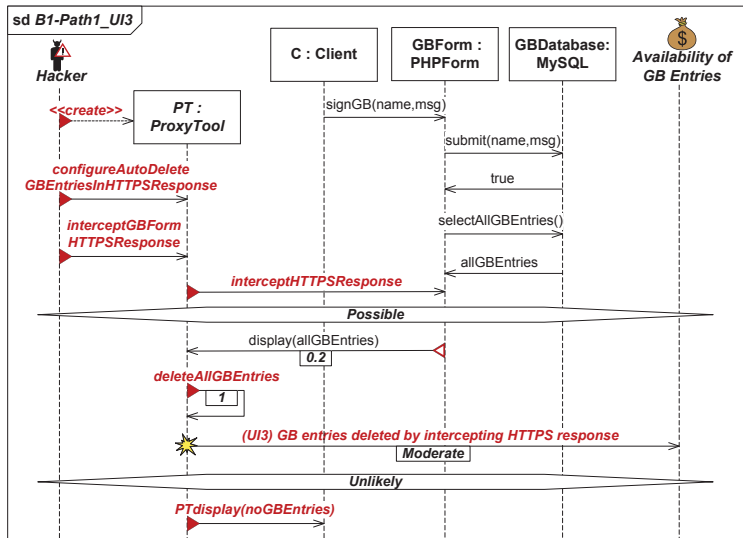


Fig. 14. Estimating the likelihood of the path leading to unwanted incident *UI3*, as well as the consequence of *UI3*.

Given that the guest book makes use of proper countermeasures against HTTPS interceptions, we assign probability *0.2* on message `display(allGBEntries)`. If, however, the HTTPS response gets intercepted, then there is nothing preventing the proxy tool from deleting the guest book entries situated inside the HTTPS response. Thus, we assign probability *1* on message `deleteAllGBEntries`. We calculate likelihood values as explained throughout Sect. 5.1 and see from Fig. 14 that *UI3* occurs with likelihood Unlikely. The occurrence of *UI3* implies that the guest book entries are deleted at an HTTPS response level. This

means that the guest book entries are only deleted while in transit from the guest book to the client. Since the purpose of the guest book is to read and submit guest book entries, it is easily noticeable if the guest book constantly produces responses containing no guest book entries. Based on this observation, and because the guest book in this demonstration is rather simple and easy to administrate, one should be able to apply a fix within a day. Thus, *UI3* has an impact on the security asset with a moderate consequence.

## 6 Step 3: Threat Scenario Prioritization

Figure 15 shows the risk evaluation matrix established during preparation of the risk analysis. The risk evaluation matrix is composed of the likelihood scale in Table 2 and the consequence scale in Tables 3 and 4. In traditional risk analysis, risk evaluation matrices are designed to group the various combinations of likelihood and consequence into three to five risk levels (e.g., low, medium and high). Such risk levels cover a wide spectrum of likelihood and consequence combinations and are typically used as a basis for deciding whether to accept, monitor or treat risks. However, in the setting of risk-driven testing, one is concerned about prioritizing risks to test certain aspects of the SUT exposed to risks. A higher granularity with respect to risk levels may therefore be more practical. The risk evaluation matrix in Fig. 15 represents nine risk levels, horizontally on the matrix. The tester defines the interpretation of the risk levels. In this demonstration we let numerical values represent risk levels; [1] represents the lowest risk level and [9] represents the highest risk level.

In Step 1, we identified five different paths. Three of these paths, i.e., the paths shown in Figs. 4, 5 and 6 includes *UI1*, which is a risk posed on the integrity of the guest-book's source code. Let us name these paths *UI1P1*, *UI1P2* and *UI1P3*, respectively. Similarly, let us name the path including *UI2* (see Fig. 8) as *UI2P1*, and the path including *UI3* (see Fig. 9) as *UI3P1*. *UI2* and *UI3* are risks posed on the availability of the guest book entries.

In Step 2, we estimated that *UI1P1* and *UI1P2* occur with likelihood Possible, and that *UI1P3* occurs with likelihood Unlikely. The risk caused by these paths, i.e., *UI1*, was estimated to have a moderate consequence on the integrity of the guest-book's source code. The likelihood for the occurrence of *UI2P1* was estimated to Possible, while the likelihood for the occurrence of *UI3P1* was estimated to Unlikely. Moreover, *UI2* and *UI3* were estimated to have a catastrophic and moderate consequence, respectively, on the availability of the guest book entries.

We map each path to the risk evaluation matrix with respect to the likelihood value and the consequence value of the risk included in the path. The result is shown in the risk evaluation matrix in Fig. 15. Let us say we are only interested in testing the paths that have a risk level [5] or higher. Based on this, we see from the risk evaluation matrix in Fig. 15 that we need to select *UI1P1*, *UI1P2* and *UI2P1* for testing. However, this selection excludes *UI1P3*, which is the third path that leads to *UI1*, and which is only one risk level less than the other

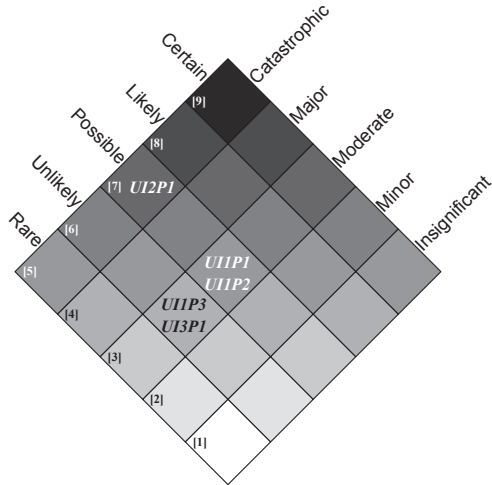


Fig. 15. Risk evaluation matrix.

two paths leading to *UI1*. Such clear cut selections are often difficult to justify because it is not obvious why some paths are selected for testing, while others are excluded. One way to come up with supporting evidence for confirming or refuting such clear cut selections, is to aggregate the likelihood values in the paths leading to the same risk.

*UI1P1*, *UI1P2* and *UI1P3* are separate paths. By separate paths, we mean paths that do not overlap in content such that no possible instance of one path can be an instance of the other. This also means that one path cannot be a special case of the other. We may therefore identify an aggregated likelihood value by summing up the frequency interval in each path. The frequency interval in *UI1P1*, *UI1P2* and *UI1P3* is  $[72, 144 > :1y]$ ,  $[45, 90 > :1y]$  and  $[6, 18 > :1y]$ , respectively. We sum up these frequency intervals and get the new frequency interval  $[123, 252 > :1y]$ . We map this frequency interval to the likelihood scale in Table 2 and see that it is skewed more towards Likely than Possible. This means that the aggregated likelihood value for the paths *UI1P1*, *UI1P2* and *UI1P3* is Likely. However, we see from the frequency interval for *UI1P3* that it has an insignificant contribution for the aggregated likelihood value. In fact, we still get Likely as the aggregated likelihood value if we exclude the frequency interval for *UI1P3* from the aggregation. Because of this, we choose not to select *UI1P3* for testing. We select *UI1P1*, *UI1P2* and *UI2P1* for testing.

## 7 Step 4: Threat Scenario Test Case Design

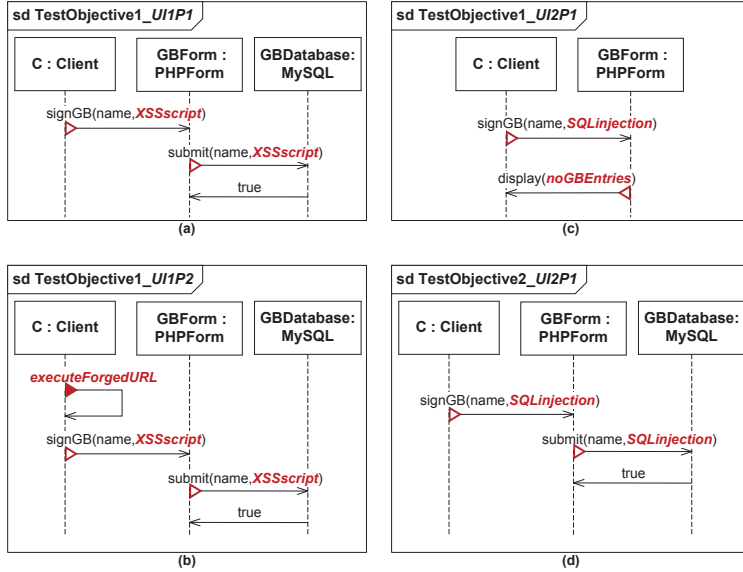
Suppose, for the sake of the example, the following suspension criteria is given: “Define no more than two test objectives per path selected for testing, and specify a test case with respect to each test objective you define”. The paths we selected for testing in Step 3 are *UI1P1*, *UI1P2* and *UI2P1*. The following lists one test objective for path *UI1P1*, one test objective for path *UI1P2*, and two test objectives for path *UI2P1*.

- **Test objective 1 for path *UI1P1***: Verify whether the guest book database (lifeline *GBDatabase*) stores an XSS script, by submitting an XSS script via the client (lifeline *C*).
- **Test objective 1 for path *UI1P2***: Verify whether the guest book database (lifeline *GBDatabase*) stores an XSS script by executing a forged URL, containing an XSS script, on the client (lifeline *C*).
- **Test objective 1 for path *UI2P1***: Verify whether the guest book form (lifeline *GBForm*) displays no guest book entries by submitting an SQL query, via the client (lifeline *C*), that is constructed for deleting the guest book entries.
- **Test objective 2 for path *UI2P1***: Verify whether the guest book database (lifeline *GBDatabase*) deletes guest book entries by submitting an SQL query, via the client (lifeline *C*), that is constructed for deleting the guest book entries.

We proceed by specifying test cases with respect to the test objectives. First, for each test objective, we identify the necessary interaction in the relevant path. By necessary interaction, we mean the interaction that is necessary in order to fulfill the test objective. Then, we copy the necessary interaction into a new sequence diagram. Finally, we annotate the new sequence diagrams, with respect to the test objectives, using the UML Testing Profile [16]. Because the tester defines the test objectives, it is the tester who knows which interactions are necessary to fulfill the test objectives. In test objective 1 for path *UI1P1*, we are interested in testing whether the guest book database stores an XSS script injected via the client. That is, we are interested in testing the interaction consisting of messages *signGB(name, XSSscript)*, *submit(name, XSSscript)* and *true* in path *UI1P1* (Fig. 4). Thus, we copy this interaction from path *UI1P1* into a new sequence diagram. The result is shown in Fig. 16a. Note that we choose not to copy other messages from path *UI1P1* because they are not needed for fulfilling the test objective.

In test objective 1 for path *UI1P2* (Fig. 5), we are interested in testing the interaction consisting of messages *executeForgedURL*, *signGB(name, XSSscript)*, *submit(name, XSSscript)* and *true*. In test objective 1 for path *UI2P1* (Fig. 8), we are interested in testing the interaction consisting of messages *signGB(name, SQLinjection)* and *display(noGBEntries)*. Finally, in test objective 2 for path *UI2P1* (Fig. 8), we are interested in testing the interaction consisting of messages *signGB(name, SQLinjection)*, *submit(name, SQLinjection)* and *true*. We





**Fig. 16.** (a) The interaction necessary to fulfill test objective 1 for path *UI1P1*. (b) The interaction necessary to fulfill test objective 1 for path *UI1P2*. (c) The interaction necessary to fulfill test objective 1 for path *UI2P1*. (d) The interaction necessary to fulfill test objective 2 for path *UI2P1*.

follow the same procedure as described above and model sequence diagrams containing the interactions necessary for fulfilling each of the test objectives in this paragraph. Figures 16b, 16c, and 16d show the interactions necessary for fulfilling test objective 1 for path *UI1P2*, test objective 1 for path *UI2P1*, and test objective 2 for path *UI2P1*, respectively.

We specify test cases by annotating the sequence diagrams in Fig. 16 using the stereotypes given in the UML Testing Profile [16]: The stereotype `<<SUT>>` is applied to one or more properties of a classifier to specify that they constitute the system under test. The stereotype `<<TestComponent>>` is used to represent a component that is a part of the test environment which communicates with the SUT or other test components. Test components are used in test cases for stimulating the SUT with test data and for evaluating whether the responses of the SUT adhere with the expected ones. The stereotype `<<ValidationAction>>` is used on execution specifications, on lifelines representing test components, to set verdicts in test cases. The UML Testing Profile defines the following five verdicts: None (the test case has not been executed yet), pass (the SUT adheres to the expectations), inconclusive (the evaluation cannot be evaluated to be

pass or fail), fail (the SUT differs from the expectation) and error (an error has occurred within the testing environment). The number of verdicts may be extended, if required.

The system under test in Fig. 16a is the guest book database (lifeline GB-Database), because we are testing whether the guest book database stores an XSS script submitted via the client. The system under test in Figs. 16b and 16d is also the guest book database. In the former, we again test whether the guest book database stores an XSS script, but this time we execute a forged URL containing an XSS script via the client. In the latter, we test whether the guest book database deletes the guest book entries by submitting an SQL query via the client. The system under test in Fig. 16c is the guest book form (lifeline GB-Form), because we are testing whether the guest book form displays no guest book entries as a result of executing an SQL injection. Based on this, we annotate lifeline GBDatabase in Figs. 16a, 16b and 16d, and lifeline GBForm in Fig. 16c with stereotype <<SUT>>.

The client (lifeline C) and the guest book form (lifeline GBForm) in Figs. 16a, 16b and 16d stimulate the system under test, i.e., the guest book database, with test data in terms of *XSSscript*, *XSSscript*, and *SQLinjection*, respectively. Thus, we annotate lifelines C and GBForm in Figs. 16a, 16b and 16d with stereotype <<TestComponent>>. In Fig. 16c, however, it is only the client that stimulates the system under test, which in this case is the guest book form. Thus, we annotate the client (lifeline C) in Fig. 16c with stereotype <<TestComponent>>.

As mentioned above, test components are also used for evaluating whether the responses of the SUT adhere with the expected ones. We see from Figs. 16a, 16b and 16d that the test components receiving the responses of the SUT is the guest book form. Thus, we add an execution specification on lifeline GBForm in Figs. 16a, 16b and 16d, annotated with stereotype <<ValidationAction>> to set the verdict for the test case. Similarly, we add an execution specification on lifeline C in Fig. 16c, annotated with stereotype <<ValidationAction>>. The verdict is set to fail meaning that the SUT differs from the expected behavior. For example, if XSS script injection is successfully carried out then the SUT differs from the expected behavior, which should be to prevent XSS injections.

The outcome of these annotations is one test case, per test objective, as shown in Figs. 17, 18, 19 and 20. Figure 17 represents a test case specified with respect to test objective 1 for path *UI1P1*. Figure 18 represents a test case specified with respect to test objective 1 for path *UI1P2*. Figure 19 represents a test case specified with respect to test objective 1 for path *UI2P1*. Figure 20 represents a test case specified with respect to test objective 2 for path *UI2P1*.

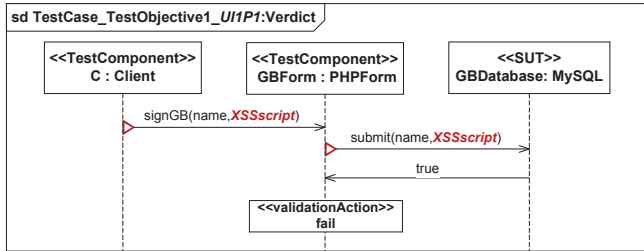


Fig. 17. Test case specified with respect to test objective 1 for path *UI1P1*.

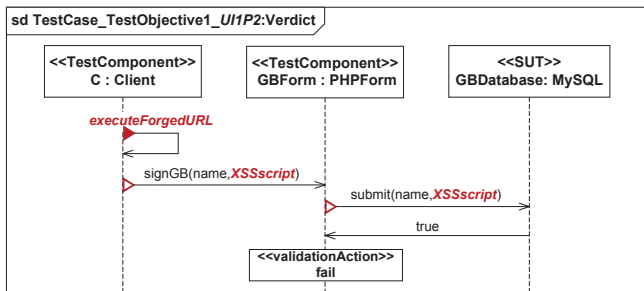


Fig. 18. Test case specified with respect to test objective 1 for path *UI1P2*.

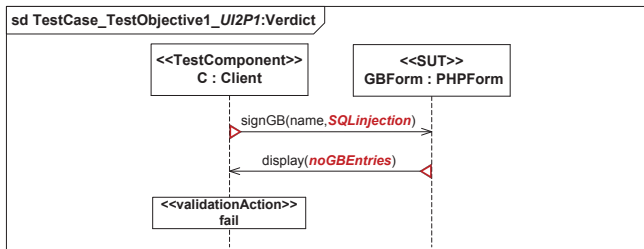


Fig. 19. Test case specified with respect to test objective 1 for path *UI2P1*.

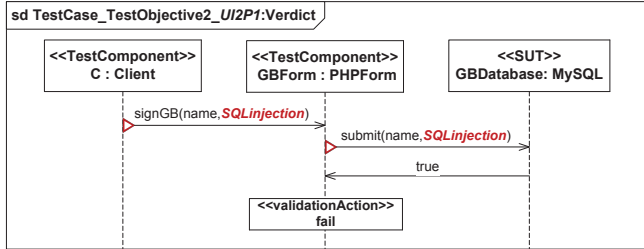


Fig. 20. Test case specified with respect to test objective 2 for path *UI2P1*.

## 8 Related Work

Although risk analysis, within risk-driven testing, is traditionally used as a basis for planning the test process, few approaches also provide guidelines for deriving test cases as part of the approach. These approaches explain the process of identifying, estimating and prioritizing risks either partly or by briefly mentioning it. In [2, 11], risks are identified by making use of fault tree analysis, however, there is no explanation on how to estimate and prioritize the risks. In [7], the authors refer to fault tree analysis for identifying risks. There is no explanation on how to estimate and prioritize risks. In [13], the authors refer to a risk analysis approach published by NIST [25] for identifying security risks. However, there is no further explanation on how to identify and estimate the security risks, yet, security risks are prioritized with respect to a predefined risk assessment matrix. In [27], security risks are identified solely by matching attack patterns on the public interfaces of a SUT. The estimation and prioritization of risks are only based on a complexity factor for specific operations in the SUT. In practice, other factors may be considered, e.g., vulnerability statistics and incident reports. In [3], test cases are prioritized by calculating a risk exposure for test cases, with the objective to quantitatively measure the quality of test cases. Risk estimation is carried out by multiplying the probability of a fault occurring with the costs related to the fault. However, there is no explanation about how risks are identified. In [24], risks are estimated by multiplying the probability that an entity contains fault with the associated damage. Similar to [3], this value is used to prioritize test cases, and there is no explanation about how risks are identified.

All of these approaches use separate modeling languages or techniques for representing the risk analysis and the test cases: In [2, 7, 11], fault trees are used to identify risks, while test cases are derived from state machine diagrams with respect to information provided by the fault trees. In [13], high level risks are detailed by making use of threat modeling. Misuse cases are developed with respect to the threat models, which are then used as a basis for deriving test cases represented textually. In [27], risk models are generated automatically by

making use of a vulnerability knowledge database. The risk models are used as input for generating misuse cases, which are also identified in similar manner. Misuse cases are used as a basis for deriving test cases. In [3, 24], a test case is a path in an activity diagram, starting from the activity diagram's initial node and ending at its final node. In [3], risks are estimated using tables, while in [24], risk information is annotated on the activities of an activity diagram, only in terms of probability, damage and their product.

## 9 Conclusion

In order to bridge the gap between high level risks and low level test cases, risk-driven testing approaches must provide testers with a systematic method for designing test cases by making use of the risk analysis. Our method is specifically designed to meet this goal.

The method starts after test planning, but before test design, according to the testing process presented by ISO/IEC/IEEE 29119 [10]. It brings risk analysis to the work bench of testers because it employs UML sequence diagrams as the modeling language, conservatively extended with our own notation for representing risk information. Sequence diagrams are widely recognized and used within the testing community and it is among the top three modeling languages applied within the model based testing community [14]. Risk identification, estimation and prioritization in our method are in line with what is referred to as risk assessment in ISO 31000 [8]. Finally, our approach makes use of the UML Testing Profile [16] to specify test cases in sequence diagrams. This means that our method is based on widely accepted standards and languages, thus facilitating adoption among the software testing community.

**Acknowledgments.** This work has been conducted as a part of the DIAMONDS project (201579/S10) funded by the Research Council of Norway, the NESSoS network of excellence (256980) and the RASEN project (316853) funded by the European Commission within the 7th Framework Programme, as well as the CONCERTO project funded by the ARTEMIS Joint Undertaking (333053) and the Research Council of Norway (232059).

## References

1. F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-Middle Attack to the HTTPS Protocol. *IEEE Security & Privacy*, 7(1):78–81, 2009.
2. R. Casado, J. Tuya, and M. Younas. Testing Long-lived Web Services Transactions Using a Risk-based Approach. In *Proc. 10th International Conference on Quality Software (QSIC'10)*, pages 337–340. IEEE Computer Society, 2010.
3. Y. Chen, R.L. Probert, and D.P. Sims. Specification-based Regression Test Selection with Risk Analysis. In *Proc. 2002 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'02)*, pages 1–14. IBM Press, 2002.

4. Damn Vulnerable Web Application (DVWA). <http://www.dvwa.co.uk/>. Accessed August 11, 2013.
5. G. Erdogan, Y. Li, R.K. Runde, F. Seehusen, and K. Stølen. Conceptual Framework for the DIAMONDS Project. Technical Report A22798, SINTEF Information and Communication Technology, 2012.
6. V. Garousi and J. Zhi. A survey of software testing practices in Canada. *Journal of Systems and Software*, 86(5):1354–1376, 2013.
7. M. Gleirscher. Hazard-based Selection of Test Cases. In *Proc. 6th International Workshop on Automation of Software Test (AST'11)*, pages 64–70. ACM, 2011.
8. International Organization for Standardization. *ISO 31000:2009(E), Risk management – Principles and guidelines*, 2009.
9. International Organization for Standardization. *ISO/IEC/IEEE 29119-1:2013(E), Software and system engineering - Software testing - Part 1: Concepts and definitions*, 2013.
10. International Organization for Standardization. *ISO/IEC/IEEE 29119-2:2013(E), Software and system engineering - Software testing - Part 2: Test process*, 2013.
11. J. Kloos, T. Hussain, and R. Eschbach. Risk-based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis. In *Proc. 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 26–33. IEEE Computer Society, 2011.
12. M.S. Lund, B. Solhaug, and K. Stølen. *Model-Driven Risk Analysis: The CORAS Approach*. Springer, 2011.
13. K.K. Murthy, K.R. Thakkar, and S. Laxminarayan. Leveraging Risk Based Testing in Enterprise Systems Security Validation. In *Proc. 1st International Conference on Emerging Network Intelligence (EMERGING'09)*, pages 111–116. IEEE Computer Society, 2009.
14. A.C. Dias Neto, R. Subramanyan, M. Vieira, and G.H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *Proc. 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07)*, pages 31–36. ACM, 2007.
15. Object Management Group. *Unified Modeling Language (UML), superstructure, version 2.4.1*, 2011. OMG Document Number: formal/2011-08-06.
16. Object Management Group. *UML Testing Profile (UTP), version 1.2*, 2013. OMG Document Number: formal/2013-04-03.
17. R. Oppliger, R. Hauser, and D. Basin. SSL/TLS session-aware user authentication - Or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12):2238–2246, 2006.
18. Open Web Application Security Project (OWASP). [https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)). Accessed September 5, 2013.
19. Open Web Application Security Project (OWASP). [https://www.owasp.org/index.php/Top\\_10\\_2013-A8-Cross-Site\\_Request\\_Forgery\\_\(CSRF\)](https://www.owasp.org/index.php/Top_10_2013-A8-Cross-Site_Request_Forgery_(CSRF)). Accessed December 16, 2013.
20. OWASP Top 10 2013 – A6 – Sensitive Data Exposure. [https://www.owasp.org/index.php/Top\\_10\\_2013-A6-Sensitive\\_Data\\_Exposure](https://www.owasp.org/index.php/Top_10_2013-A6-Sensitive_Data_Exposure). Accessed December 17, 2013.
21. OWASP Top 10 2013 – Release Notes. [https://www.owasp.org/index.php/Top\\_10\\_2013-Release\\_Notes](https://www.owasp.org/index.php/Top_10_2013-Release_Notes). Accessed September 6, 2013.
22. OWASP Top 10 Application Security Risks – 2013. [https://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project). Accessed September 6, 2013.

23. PHP manual. <http://php.net/manual/en/pdo.prepared-statements.php>. Accessed September 6, 2013.
24. H. Stallbaum, A. Metzger, and K. Pohl. An Automated Technique for Risk-based Test Case Generation and Prioritization. In *Proc. 3rd International Workshop on Automation of Software Test (AST'08)*, pages 67–70. ACM, 2008.
25. G. Stoneburner, A. Goguen, and A. Feringa. Risk Management Guide for Information Technology Systems. NIST Special Publication 800-30, National Institute of Standards and Technology, 2002.
26. XAMPP. <http://www.apachefriends.org/en/xampp.html>. Accessed August 11, 2013.
27. P. Zech, M. Felderer, and R. Breu. Towards a Model Based Security Testing Approach of Cloud Computing Environments. In *Proc. 6th International Conference on Software Security and Reliability Companion (SERE-C'12)*, pages 47–56. IEEE Computer Society, 2012.



Technology for a better society

[www.sintef.no](http://www.sintef.no)



Chapter **1 1**

Paper 3: Schematic generation of  
English-prose semantics for a risk analysis  
language based on UML interactions



# Report

## Schematic Generation of English-prose Semantics for a Risk Analysis Language Based on UML Interactions

**Author(s)**

Gencer Erdogan, Atle Refsdal, and Ketil Stølen

SINTEF IKT  
SINTEF ICT

Address:  
Postboks 124 Blindern  
NO-0314 Oslo  
NORWAY

Telephone:+47 73593000  
Telefax:+47 22067350  
postmottak.IKT@sintef.no  
www.sintef.no  
Enterprise /VAT No:  
NO 948 007 029 MVA

**KEYWORDS:**  
Risk analysis language,  
Risk-driven testing,  
UML interactions,  
Sequence diagram,  
CORAL diagram

# Report

## Schematic Generation of English-prose Semantics for a Risk Analysis Language Based on UML Interactions

**VERSION**  
Final

**DATE**  
2014-10-27

**AUTHOR(S)**  
Gencer Erdogan, Atle Refsdal, and Ketil Stølen

**CLIENT(S)**  
Norwegian Research Council

**CLIENT'S REF.**  
201579/S10

**PROJECT NO.**  
102002253


**NUMBER OF PAGES/APPENDICES:**  
18/3

### ABSTRACT

To support risk-driven testing, we have developed CORAL, a language for risk analysis based on UML interactions. In this report, we present its semantics as a translation of CORAL diagrams into English prose. The CORAL semantics is developed to help software testers to clearly and consistently document, communicate and analyze risks in a risk-driven testing process. We first provide an abstract syntax and a translation algorithm. Then, we evaluate the approach based on some examples. We argue that the resulting English prose is comprehensible by testers, is consistent with the semantics of UML interactions, and has a complexity that is linear to the complexity of CORAL diagrams in terms of size.

**PREPARED BY**  
Gencer Erdogan

SIGNATURE



**CHECKED BY**  
Bjørnar Solhaug

SIGNATURE



**APPROVED BY**  
Bjørn Skjellaug

SIGNATURE



**REPORT NO.**  
SINTEF A26407

**ISBN**  
978-82-14-05367-8

**CLASSIFICATION**  
Unrestricted

**CLASSIFICATION THIS PAGE**  
Unrestricted

## CONTENTS

<b>I</b>	<b>Introduction</b>	4
<b>II</b>	<b>Success Criteria</b>	4
<b>III</b>	<b>Approach</b>	4
III-A	Abstract syntax of CORAL . . . . .	5
III-B	English-prose semantics of CORAL . . . . .	5
<b>IV</b>	<b>Discussion</b>	6
IV-A	The English-prose semantics of CORAL diagrams must be comprehensible to software testers when conducting risk analysis . . . . .	6
IV-B	The CORAL semantics of the constructs inherited from UML interactions must be consistent with their semantics in the UML standard . . . . .	8
IV-C	The complexity of the resulting English prose must scale linearly with the complexity of CORAL diagrams in terms of size . . . . .	10
<b>V</b>	<b>Related Work</b>	10
<b>VI</b>	<b>Conclusion</b>	10
	<b>References</b>	11
	<b>Appendix A: Abstract Syntax of CORAL</b>	12
A-A	Messages . . . . .	12
A-B	Lifelines . . . . .	12
A-C	Risk-measure annotations . . . . .	13
A-D	Interaction operators . . . . .	13
	<b>Appendix B: English-prose Semantics of CORAL</b>	14
B-A	Messages . . . . .	14
B-B	Lifelines . . . . .	14
B-C	Risk-measure annotations . . . . .	15
B-D	Interaction operators . . . . .	15
	<b>Appendix C: Overview of the Graphical Notation of CORAL</b>	15

## I. INTRODUCTION

Risk-driven testing is an approach that uses risk analysis to focus the testing process with respect to certain risks posed on the system under test. When conducting risk-driven testing, testers need to clearly and consistently document, communicate and analyze risks, in order to correctly focus the testing with respect to the most severe risks.

In earlier work, we presented a systematic method for designing test cases by making use of risk analysis [1], [2]. As part of the method, we also introduced a risk analysis language based on UML interactions which we refer to as CORAL. CORAL extends UML interactions with constructs for representing risk-related information in sequence diagrams, and it is specifically developed to support software testers in a risk-driven testing process.

As we explain in [1], [2], testers may use CORAL in three consecutive steps to identify, estimate, and evaluate risks. The graphical icons representing risk-related information in CORAL are based on corresponding graphical icons in CORAS, which is a model-driven approach to risk analysis [3]. This is a deliberate design decision because the graphical icons in CORAS are empirically shown to be cognitively effective [4]. Appendix C gives an overview of the graphical notation of CORAL.

However, situations may arise where the information conveyed by CORAL diagrams, i.e., interactions represented by CORAL constructs, are interpreted differently by different testers. Thus, in order to help software testers to clearly and consistently document, communicate and analyze risks, we present a structured approach to generate the semantics of CORAL diagrams in terms of English prose. We evaluate the approach based on some examples.

The CORAL language is also accompanied by a formal semantics, but as indicated above, this report presents only the natural-language semantics of CORAL. We present the natural-language semantics and the formal semantics of CORAL in different reports, because their purposes and target audiences are different. The main target audience of the natural-language semantics is software testers, while the main target audiences of the formal semantics are method developers or tool developers.

The remainder of this report is organized as follows. Section II lists the success criteria our approach aims to fulfill. Section III gives a stepwise explanation of the approach, and presents the examples on which we base our evaluation. Section IV elaborates on the fulfillment of the success criteria. Section V provides an overview of related work, while Section VI gives some concluding remarks. Appendix A and Appendix B provide the complete abstract syntax and the complete English-prose semantics of the CORAL language, respectively. Finally, Appendix C gives an overview of the graphical notation of the CORAL language.

## II. SUCCESS CRITERIA

There are three key design decisions that shape our success criteria.

**First**, the main target audience of the natural-language semantics of CORAL is software testers. CORAL is supposed to be used by testers to document, communicate and analyze risks in a risk-driven testing process. Thus, our first success criterion is: The English-prose semantics of CORAL diagrams must be comprehensible to software testers when conducting risk analysis.

**Second**, CORAL is based on UML interactions and only *extends* UML interactions with constructs representing risk-related information. Thus, our second success criterion is: The CORAL semantics of the constructs inherited from UML interactions must be consistent with their semantics in the UML standard.

**Third**, the approach must ensure scalability. Thus, our third success criterion is: The complexity of the resulting English prose must scale linearly with the complexity of CORAL diagrams in terms of size.

## III. APPROACH

Inspired by CORAS [3], we generate the English-prose semantics in three consecutive steps, as shown in Figure 1. In **Step 1**, we translate a CORAL diagram into a corresponding textual representation. This step takes a CORAL diagram as input. First, for each construct in the CORAL diagram, we identify its corresponding syntactical element in the abstract syntax of CORAL. Second, we replace the variables in the syntactical element with content, i.e., user-defined text, from the construct in the diagram. The output of this step is a textual representation of the CORAL diagram given as input to the step. The abstract syntax of CORAL is defined in Section III-A.

In **Step 2**, we translate the textual representation of a CORAL diagram into English prose, by making use of the translation algorithm defined in Section III-B. The translation algorithm is defined in terms of a function that takes syntactical elements as input and provides their English prose translation.

Before presenting the translation function, we need to explain weak sequencing, which is a key construct in UML interactions. Weak sequencing is the implicit composition mechanism combining the constructs of an interaction, and is defined as follows [5]:

- 1) The transmission of a message must occur before its reception.
- 2) Events on the same lifeline are ordered in time, where time proceeds from the top of the lifeline towards the bottom of the lifeline, and where an event is either the transmission of a message or the reception of a message.

In the translation function, we use the term '*weakly sequenced by*' to denote weak sequencing as defined above.

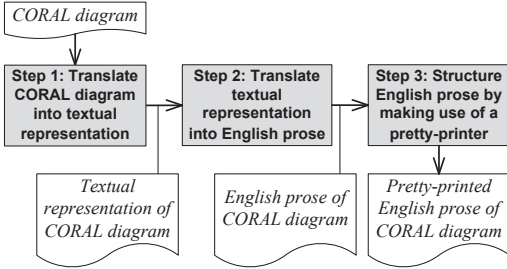


Figure 1. Generating English-prose semantics of CORAL diagrams.

In **Step 3**, we make use of a pretty-printer to format the English prose in a structured manner. The technical details of such a pretty-printer are outside the scope of this report, and are therefore not discussed here.

#### A. Abstract syntax of CORAL

In this section, we define the abstract syntax of CORAL expressed in the Extended Backus-Naur Form [6]. The syntax defined in this section is an excerpt of the complete syntax, but it is sufficient for walking through the examples in the report. The complete syntax is defined in Appendix A.

We use the following undefined terms in the grammar: *identifier*, *asset lifeline*, *exact*, *interval*, and *time unit*. The term *identifier* is assumed to represent any alphanumeric string. The term *asset lifeline* is assumed to represent an alphanumeric string describing the name of an asset lifeline. The term *exact* is assumed to represent a non-negative real number, including 0. That is,  $exact \in \mathbb{R}_{\geq 0}$ . The term *interval* is assumed to represent an interval of non-negative real numbers, including 0. The intervals are represented in standard mathematical notation. That is, one of the following:

- $[a, b]$
- $[a, b)$
- $\langle a, b]$
- $\langle a, b)$

where  $a, b \in \mathbb{R}_{\geq 0}$ , and  $a \leq b$ . The term *time unit* is assumed to represent an alphanumeric string describing a unit of time, e.g., second(s), minute(s), hour(s), day(s), year(s), etc.

In the abstract syntax, we use different fonts to distinguish between the non-terminals and the terminals. Non-terminals are written in font *math mode*, while terminals are written in font **Sans Serif**. The terminals written in font **Bold Sans Serif** represent the type of a syntactical element. For each terminal representing the **type** of a syntactical element, there is an associated English-prose semantics defined in Section III-B.

$$\begin{aligned} \textit{risk interaction} = & \textit{message} \mid \textit{weak sequencing} \\ & \mid \textit{potential alternatives} \\ & \mid \textit{referred interaction} \\ & \mid \textit{parallel execution}; \end{aligned}$$

$$\begin{aligned} \textit{message} = & \textit{risky message} \\ & \mid \textit{unwanted incident message}; \end{aligned}$$

$$\begin{aligned} \textit{risky message} = & \mathbf{rm}(\textit{identifier}, \\ & \textit{transmitter lifeline}, \\ & \textit{receiver lifeline}, \\ & \textit{risky message category}, \\ & \textit{transmission frequency}, \\ & \textit{conditional ratio}, \\ & \textit{reception frequency}); \end{aligned}$$

$$\begin{aligned} \textit{unwanted incident message} = & \mathbf{uim}(\textit{identifier}, \\ & \textit{transmitter lifeline}, \\ & \textit{asset lifeline}, \\ & \textit{transmission frequency}, \\ & \textit{consequence}); \end{aligned}$$

$$\begin{aligned} \textit{transmitter lifeline} = & \textit{general lifeline} \\ & \mid \textit{deliberate threat lifeline}; \end{aligned}$$

$$\begin{aligned} \textit{receiver lifeline} = & \textit{general lifeline} \\ & \mid \textit{deliberate threat lifeline}; \end{aligned}$$

$$\textit{general lifeline} = \mathbf{gl}(\textit{identifier});$$

$$\textit{deliberate threat lifeline} = \mathbf{dtl}(\textit{identifier});$$

$$\textit{risky message category} = \mathbf{general} \mid \mathbf{new} \mid \mathbf{alter};$$

$$\textit{transmission frequency} = \textit{frequency};$$

$$\textit{reception frequency} = \textit{frequency};$$

$$\textit{frequency} = \mathbf{f}(\textit{interval}, \textit{time unit});$$

$$\textit{conditional ratio} = \mathbf{cr}(\textit{exact});$$

$$\textit{consequence} = \mathbf{c}(\textit{identifier});$$

$$\textit{weak sequencing} = \mathbf{seq}(\{\textit{risk interaction}\}^-);$$

$$\textit{potential alternatives} = \mathbf{alt}(\{\textit{risk interaction}\}^-);$$

$$\textit{referred interaction} = \mathbf{ref}(\textit{identifier});$$

$$\textit{parallel execution} = \mathbf{par}(\{\textit{risk interaction}\}^-);$$

#### B. English-prose semantics of CORAL

The English-prose semantics of a syntactical element is given by the function  $\llbracket \cdot \rrbracket$ , which is defined below for the excerpt of the abstract syntax presented in Section III-A. Let the syntactical variables

- $d$  range over *risk interaction*
- $id$  range over *identifier*
- $t$  range over *transmitter lifeline*
- $r$  range over *receiver lifeline*

- $al$  range over *asset lifeline*
- $f$  range over *frequency*
- $cr$  range over *conditional ratio*
- $c$  range over *consequence*
- $e$  range over *exact*
- $i$  range over *interval*
- $tu$  range over *time unit*

Undefined values are represented by  $\perp$ . The pair of square brackets, '[' and ']', is a part of the semantics that is used to enclose an operand.

$\llbracket \text{seq}(d_1, d_2, \dots, d_m) \rrbracket = [ \llbracket d_1 \rrbracket ]$  weakly sequenced by  
 $[ \llbracket d_2 \rrbracket ]$  weakly sequenced by ...  
 weakly sequenced by  $[ \llbracket d_m \rrbracket ]$

$\llbracket \text{alt}(d_1, d_2, \dots, d_m) \rrbracket =$  either  $[ \llbracket d_1 \rrbracket ]$  or  $[ \llbracket d_2 \rrbracket ]$  or ...  
 or  $[ \llbracket d_m \rrbracket ]$

$\llbracket \text{ref}(id) \rrbracket =$  Refer to interaction:  $id$ .

$\llbracket \text{par}(d_1, d_2, \dots, d_m) \rrbracket = [ \llbracket d_1 \rrbracket ]$  parallelly merged with  
 $[ \llbracket d_2 \rrbracket ]$  parallelly merged with ...  
 parallelly merged with  $[ \llbracket d_m \rrbracket ]$

$\llbracket \text{rm}(id, t, r, \text{general}, f_1, cr, f_2) \rrbracket =$   
 The message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  
 $\llbracket r \rrbracket [ \llbracket f_1 \rrbracket ]$ , the transmission leads to its reception  
 $\llbracket cr \rrbracket$ , and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \text{rm}(id, t, r, \text{new}, f_1, cr, f_2) \rrbracket =$   
 The new message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  
 $\llbracket r \rrbracket [ \llbracket f_1 \rrbracket ]$ , the transmission leads to its reception  
 $\llbracket cr \rrbracket$ , and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \text{rm}(id, t, r, \text{alter}, f_1, cr, f_2) \rrbracket =$   
 The altered message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  
 $\llbracket r \rrbracket [ \llbracket f_1 \rrbracket ]$ , the transmission leads to its reception  
 $\llbracket cr \rrbracket$ , and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \text{uim}(id, t, al, f, c) \rrbracket =$   
 The unwanted incident  $id$  occurs on  $\llbracket t \rrbracket [ \llbracket f \rrbracket ]$ ,  
 and impacts asset  $al [ \llbracket c \rrbracket ]$ .

$\llbracket \text{gl}(id) \rrbracket = id$

$\llbracket \text{dtl}(id) \rrbracket =$  the deliberate threat  $id$

$\llbracket \text{f}(i, tu) \rrbracket =$  with frequency interval  $i$  per  $tu$

$\llbracket \text{f}(\perp, \perp) \rrbracket =$  with undefined frequency

$\llbracket \text{cr}(e) \rrbracket =$  with conditional ratio  $e$

$\llbracket \text{cr}(\perp) \rrbracket =$  with undefined conditional ratio

$\llbracket \text{c}(id) \rrbracket =$  with consequence  $id$

$\llbracket \text{c}(\perp) \rrbracket =$  with undefined consequence

Figure 2 illustrates some examples of CORAL diagrams which we obtained by applying our method [1], [2] on a guest book that is available in the Damn Vulnerable Web

Application [7]. We demonstrate the schematic translation of CORAL diagrams into English prose by, first, translating the diagrams in Figure 2 into their corresponding textual representation. The resulting textual representation is shown in Figure 3. Then, we translate the textual representation of the diagrams into its corresponding English prose, by using the translation function presented in this section. The resulting (pretty-printed) English prose of the diagrams in Figure 2 is shown in Figure 4.

#### IV. DISCUSSION

In this section, we discuss the fulfillment of the three success criteria given in Section II.

*A. The English-prose semantics of CORAL diagrams must be comprehensible to software testers when conducting risk analysis*

The comprehensibility of the resulting English prose is supported both from a general viewpoint and from a software testing viewpoint.

From a general viewpoint, we observe the following two points. **First**, the structure of the translations in Figure 4 is similar to the structure of their corresponding CORAL diagrams in Figure 2. In particular, the ordering of the translated CORAL constructs is maintained. For example, let us consider the translation in Figure 4a. The first sentence states: “The new message forgedURLReplacingMsgWithXSSscript is transmitted from the deliberate threat Hacker to C with undefined frequency, the transmission leads to its reception with undefined conditional ratio, and the reception occurs with undefined frequency”. By comparing the translation in Figure 4a to its corresponding diagram in Figure 2a, we see that the first sentence corresponds to the first message in the diagram. Similarly, we see that the second sentence in Figure 4a corresponds to the second message in Figure 2a, and so on. **Second**, the user-defined text is unchanged in the translations. By user-defined text, we mean the text typed in CORAL diagrams, such as the text on messages, lifelines, frequency assignments, consequence assignments, and so on.

From a software testing viewpoint, we observe that risk-related concepts from CORAL are integrated with concepts from UML interactions in the resulting English prose. UML interactions are among the top three modeling languages within the testing community, and often used for testing purposes [8]. It is therefore reasonable to assume that testers understand the concepts from UML interactions. Moreover, we find it reasonable to assume that testers also comprehend the risk-related concepts we introduce in CORAL, such as *altered* messages and messages representing *unwanted incidents*, because these are concepts that are also known within the testing community. For example, in fuzz testing, the expected behavior of a system is altered by providing invalid, unexpected, or random data, which may lead to



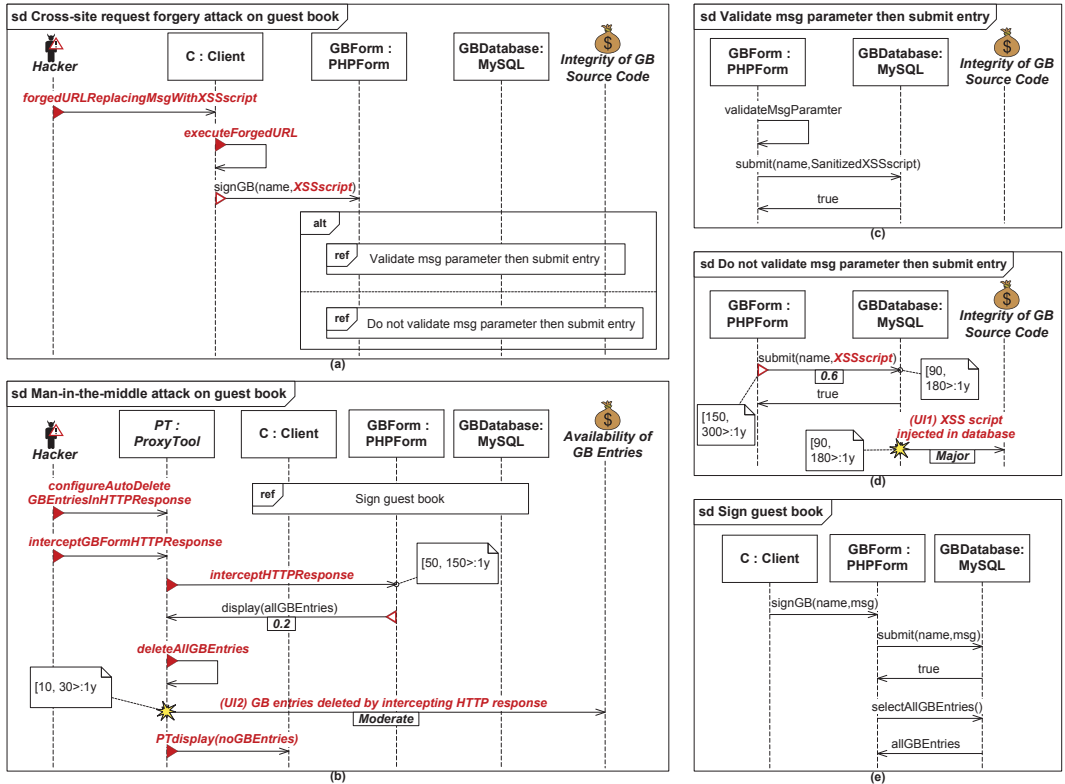


Figure 2. Examples of CORAL diagrams.

unwanted incidents [9]. Table I lists the UML interaction concepts and the risk-related concepts used in CORAL.

To illustrate how UML interaction concepts and risk related concepts in CORAL are integrated, let us consider the first message in Figure 2d. This message represents an altered message. In CORAL, an altered message is a message in the system model which has been altered due to unexpected system behavior or unexpected input data. Figure 4d shows the corresponding translation as: “The altered message submit(name,XSSscript) is transmitted from GBForm to GBDatabase with frequency interval [150, 300> per 1y, the transmission leads to its reception with conditional ratio 0.6, and the reception occurs with frequency interval [90, 180> per 1y”. The translation shows that we have a *message* that is transmitted between two *lifelines* (UML interaction concepts). Furthermore, the translation also shows that the message is *altered*, transmitted and received with a given *frequency*, and that the transmission of the message leads to its reception with a given *conditional*

Table I  
UML INTERACTION CONCEPTS AND RISK-RELATED CONCEPTS USED IN CORAL

UML interaction concepts	Risk-related concepts
Message	New message Altered message Deleted message Unwanted incident message
Lifeline	Deliberate threat lifeline Accidental threat lifeline Non-human threat lifeline Asset lifeline
<i>Interaction operators:</i> Weak sequencing Potential alternatives Referred interaction Parallel Loop	<i>Risk-measure annotations assigned on messages:</i> Frequency Conditional ratio Consequence

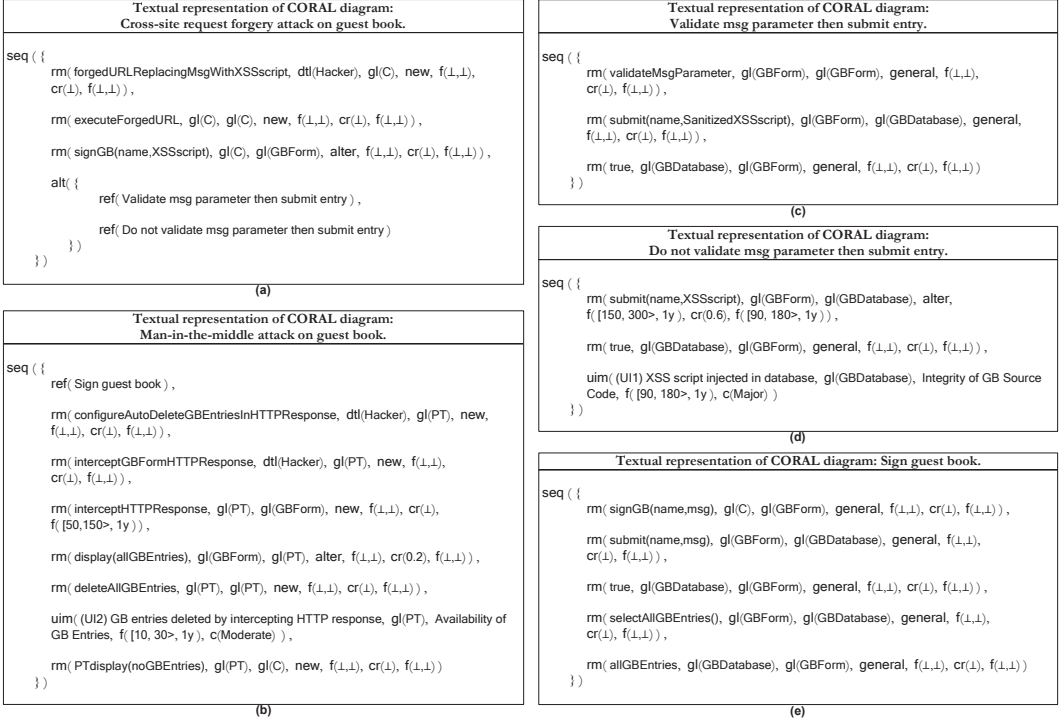


Figure 3. Textual representation of the corresponding CORAL diagrams in Figure 2.

*ratio* (risk-related concepts).

*B. The CORAL semantics of the constructs inherited from UML interactions must be consistent with their semantics in the UML standard*

The CORAL constructs inherited from UML interactions are messages, lifelines and the interaction operators: seq, ref, alt, par and loop. The interaction operator **weak sequencing (seq)** is defined and related to CORAL in Section III.

According to the UML standard, a “**message** defines a particular communication between lifelines of an interaction,” and “the signature of a message is the specification of its content” [5] (pp. 505–506). A message also defines its transmission event (which occurs on the transmitter lifeline) and its reception event (which occurs on the receiver lifeline) [5] (p. 506). Thus, a message may be defined as the triple  $(id, t, r)$ , where  $id$  represents the signature,  $t$  represents the transmitter lifeline, and  $r$  represents the receiver lifeline. We define a message in a similar manner. However, as explained in Section III, we also distinguish between the category of a message, i.e., whether it is a general, new, altered, deleted or an unwanted incident message. In

addition, we allow the assignment of a frequency value on the transmission/reception of general, new and altered messages, as well as the transmission of unwanted incident messages. Conditional ratios are assigned on general, new and altered messages, while consequences are assigned only on unwanted incident messages. Deleted messages have no risk-measure annotations. The syntax and semantics of a deleted message is given in Appendices A and B, respectively. As we can see from the translations in Figure 4, the English prose of messages are generated according to their category, and contain information about the message signature, the lifeline transmitting the message, the lifeline receiving the message, and the risk-measure annotations assigned on the message if they are defined.

According to the UML standard, an “**interaction use (ref)** refers to an interaction. The interaction use is shorthand for copying the contents of the referred interaction where the interaction use is. To be accurate the copying must take into account substituting parameters with arguments and connect the formal gates with the actual ones.” [5] (p. 501). Figure 2b shows an example of an interaction use named Sign guest

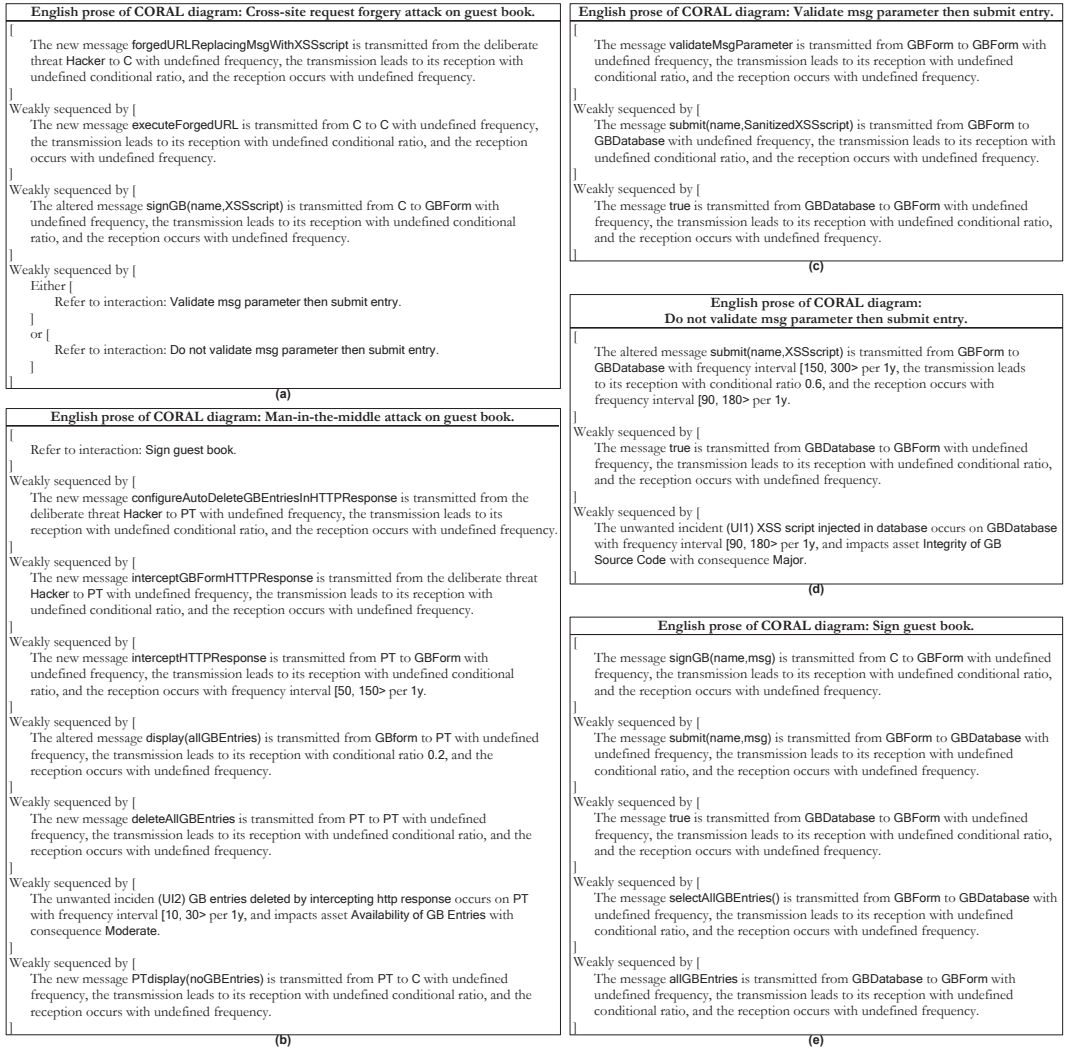


Figure 4. English prose of the corresponding CORAL diagrams in Figure 2.

book. The interaction referred to by this interaction use is shown in Figure 2e. We use the term ‘*refer to interaction*’ to denote an interaction use, as shown in the translations in Figures 4a and 4b.

According to the UML standard, the “interaction operator **potential alternatives (alt)** designates that the operands represent a choice of behavior” [5] (p. 482). The UML standard requires that the chosen operand must have an explicit or implicit guard expression that evaluates to true. An implicit true guard is implied if the operand has no explicit guard. In CORAL, we currently allow only the usage of implicit true guards. However, the syntax and semantics of CORAL is easily extendable to support explicit guards as well. As shown in Figure 4a, we use the term ‘*either*’ in front of the first operand of an alt operator, and then the term ‘*or*’ between each subsequent operand to reflect the disjunctive behavior of the alt operator.

According to the UML standard, the “interaction operator **parallel execution (par)** designates a parallel merge between the behaviors of the operands. A parallel merge defines a set of traces that describes all the ways that events of the operands may be interleaved without obstructing the order of the events within the operands” [5] (p. 483). We use the term ‘*parallelly merged with*’ between each operand to denote a parallel merge between the behaviors of the operands.

The above paragraphs show that the CORAL semantics of the constructs inherited from UML interactions are consistent with their semantics in the UML standard.

### *C. The complexity of the resulting English prose must scale linearly with the complexity of CORAL diagrams in terms of size*

As illustrated by Figure 2 and Figure 4, the definition of the translation function in Section III-B ensures that the structure of its output mirrors the input diagram, and that there is a linear relationship between the size of input and output. A formal argument that this would hold for any diagram  $d$  could be given based on induction over the syntactical structure of  $d$ .

## V. RELATED WORK

To the best of our knowledge, no risk-driven testing approach provides a similar schematic generation of natural language semantics as presented in this report. Most approaches use risk tables/matrices or risk annotated models as a means for documenting, communicating and analyzing risks posed on the system under test. However, some approaches provide guidelines for documenting risk-related information in natural-language semantics.

Redmill [10] provides a set of guide words with associated definitions, which may be used as a basis for documenting risk-related information. The set of guide words are used to describe different ways in which system services may fail,

and they are designed to focus the testing on the various types of failures that may occur. What Redmill [10] refers to as failure is similar to what we refer to as unwanted incident in CORAL. However, the resulting description of a failure, which is obtained by making use of the guide words, does neither describe the likelihood nor the consequence of the failure.

Gleirscher [11] makes use of a safety analysis pattern for describing informal test cases. An informal test case is described in terms of a chain of events that may lead to a hazard (or hazardous state). What Gleirscher [11] refers to as hazard is similar to what we refer to as unwanted incident in CORAL. However, the informal test cases do neither describe the likelihood nor the consequence of hazards. Furthermore, the events that lead up to a hazard are similar to what we refer to as the transmission/reception of messages in CORAL. As shown in previous sections, we describe the likelihood of the transmission/reception of messages (in terms of frequencies), as well as the likelihood and consequence of unwanted incidents.

Nazier and Bauer [12] provide a template for documenting safety risk information, while Kumar et al. [13] provide a template for documenting risk-related information within the domain of aspect oriented programming. Both approaches extract risk-related information provided by fault trees. The risk-related information consists of the expected causes of failures and the combination of these causes which may lead to the root node (the fault) of the fault tree. A fault is similar to what we refer to as unwanted incident in CORAL. None of these approaches consider the likelihood or the consequence of the faults when documenting the risk-related information using their templates.

Souza et al. [14] use a taxonomy-based questionnaire for documenting risk-related information. The taxonomy-based questionnaire is answered by those involved in the risk-based testing approach suggested by the authors, and the objective is to “identify only technical risks that are commonly related to software functionalities or requirements” [14]. The approach makes sure to gather and document the likelihood of risks (in terms of risk exposure values), but it does not consider the consequence of risks.

## VI. CONCLUSION

CORAL is a risk analysis language based on UML interactions, and it is specifically developed to support software testers in a risk-driven testing process. CORAL extends UML interactions with constructs for representing risk-related information in sequence diagrams.

In this report, we presented a structured approach to generate the semantics of CORAL diagrams in terms of English prose. The CORAL semantics is developed to help testers to clearly and consistently document, communicate and analyze risks in a risk-driven testing process. In particular, it helps testers to: (1) obtain a correct understanding

of CORAL diagrams, (2) analyze risks posed on the system under test in a clear and consistent manner, and (3) clearly communicate risks posed on the system under test.

We argue that the resulting English prose is comprehensible by testers because: (1) it preserves the structure of CORAL diagrams, (2) it keeps the user-defined text in CORAL diagrams unchanged, and (3) it uses concepts that are known to software testers. In addition, the resulting English prose of the constructs inherited from UML interactions is consistent with their semantics in the UML standard. Moreover, the complexity of the resulting English prose scales linearly with the complexity of the CORAL diagrams in terms of size.

#### ACKNOWLEDGMENT

This work has been conducted as a part of the DIAMONDS project (201579/S10) funded by the Research Council of Norway, the NESSoS network of excellence (256980) and the RASEN project (316853) funded by the European Commission within the 7th Framework Programme, as well as the CONCERTO project funded by the ARTEMIS Joint Undertaking (333053) and the Research Council of Norway (232059).

#### REFERENCES

- [1] G. Erdogan, A. Refsdal, and K. Stølen, "A Systematic Method for Risk-Driven Test Case Design Using Annotated Sequence Diagrams," in *Proc. 1st International Workshop on Risk Assessment and Risk-driven Testing (RISK'13)*. Springer, 2014, pp. 93–108.
- [2] —, "A Systematic Method for Risk-Driven Test Case Design Using Annotated Sequence Diagrams," SINTEF Information and Communication Technology, Technical Report A26036, 2014.
- [3] M. S. Lund, B. Solhaug, and K. Stølen, *Model-Driven Risk Analysis: The CORAS Approach*. Springer, 2011.
- [4] B. Solhaug and K. Stølen, "The CORAS Language - Why it is designed the way it is," in *Proc. 11th International Conference on Structural Safety and Reliability (ICOSSAR'13)*. CRC Press, 2013, pp. 3155–3162.
- [5] *Unified Modeling Language (UML), superstructure, version 2.4.1*, Object Management Group, 2011, OMG Document Number: formal/2011-08-06.
- [6] *ISO/IEC 14977:1996(E), Information technology – Syntactic metalanguage – Extended BNF, first edition*, International Organization for Standardization, 1996.
- [7] "Damn Vulnerable Web Application," accessed September 16, 2014. [Online]. Available: <http://www.dvwa.co.uk/>
- [8] A. D. Neto, R. Subramanyan, M. Vieira, and G. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *Proc. 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07)*. ACM, 2007, pp. 31–36.
- [9] P. Oehlert, "Violating assumptions with fuzzing," *Security Privacy, IEEE*, vol. 3, no. 2, pp. 58–62, 2005.
- [10] F. Redmill, "Theory and practice of risk-based testing," *Software Testing, Verification and Reliability*, vol. 15, no. 1, pp. 3–20, 2005.
- [11] M. Gleirscher, "Hazard-based selection of test cases," in *Proc. 6th International Workshop on Automation of Software Test (AST'11)*. ACM, 2011, pp. 64–70.
- [12] R. Nazier and T. Bauer, "Automated risk-based testing by integrating safety analysis information into system behavior models," in *Proc. 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW'12)*. IEEE, 2012, pp. 213–218.
- [13] N. Kumar, D. Sosale, S. N. Konuganti, and A. Rathi, "Enabling the adoption of aspects-testing aspects: A risk model, fault model and patterns," in *Proc. 8th ACM International Conference on Aspect-Oriented Software Development (AOSD'09)*. ACM, 2009, pp. 197–206.
- [14] E. Souza, C. Gusmão, and J. Venâncio, "Risk-based testing: A case study," in *Proc. 7th International Conference on Information Technology: New Generations (ITNG'10)*. IEEE, 2010, pp. 1032–1037.

APPENDIX A.  
ABSTRACT SYNTAX OF CORAL

In this appendix, we define the abstract syntax for the CORAL language using the Extended Backus-Naur Form (EBNF) [6]. The abstract syntax is presented by grouping the syntactical elements that are closely related.

We use the following undefined terms in the grammar: *identifier*, *asset lifeline*, *int*, *minint*, *maxint*, *exact*, *interval*, and *time unit*.

- The term *identifier* is assumed to represent any alphanumeric string.
- The term *asset lifeline* is assumed to represent an alphanumeric string describing the name of an asset lifeline.
- The terms *int*, *minint* and *maxint* are assumed to represent non-negative natural numbers, including 0, where *minint* is less than, or equal to, *maxint*. That is,  $int, minint, maxint \in \mathbb{N}_0, minint \leq maxint$ .
- The term *exact* is assumed to represent a non-negative real number, including 0. That is,  $exact \in \mathbb{R}_{\geq 0}$ .
- The term *interval* is assumed to represent an interval of non-negative real numbers, including 0. The intervals are represented in standard mathematical notation. That is, one of the following:

- $[a, b]$
- $[a, b)$
- $\langle a, b \rangle$
- $\langle a, b \rangle$

where  $a, b \in \mathbb{R}_{\geq 0}$ , and  $a \leq b$ .

- The term *time unit* is assumed to represent an alphanumeric string describing a unit of time, e.g., second(s), minute(s), hour(s), day(s), year(s), etc.

Throughout the definition of the abstract syntax, we use different fonts to distinguish between the non-terminals and the terminals. Non-terminals are written in font *math mode*, while terminals are written in font **Sans Serif**. The terminals written in font **Bold Sans Serif** represent the type of a syntactical element. For each terminal representing the **type** of a syntactical element, there is an associated English-prose semantics defined in Appendix B. We start by defining the term *risk interaction*, which is a collective term for the various constructs of CORAL.

$$risk\ interaction = message \mid weak\ sequencing \mid potential\ alternatives \\ \mid referred\ interaction \mid parallel\ execution \mid loop;$$

#### A. Messages

In the following, we define the syntax of the five different messages in CORAL: general, new, alter, delete, and unwanted incident messages. The collective term for general, new and alter messages is *risky message*, the term for a deleted message is *deleted message*, and the term for an unwanted incident message is *unwanted incident message*.

$$message = risky\ message \mid unwanted\ incident\ message \mid deleted\ message; \\ risky\ message = \mathbf{rm}(identifier, transmitter\ lifeline, receiver\ lifeline, \\ risky\ message\ category, transmission\ frequency, \\ conditional\ ratio, reception\ frequency); \\ unwanted\ incident\ message = \mathbf{uim}(identifier, transmitter\ lifeline, asset\ lifeline, \\ transmission\ frequency, consequence); \\ deleted\ message = \mathbf{dm}(identifier, transmitter\ lifeline, receiver\ lifeline); \\ risky\ message\ category = \mathbf{general} \mid \mathbf{new} \mid \mathbf{alter};$$

#### B. Lifelines

In the following, we define the syntax of the lifelines in CORAL. The term *transmitter lifeline* represents the transmitter lifeline for all message categories defined in Appendix A-A, while *receiver lifeline* represents the receiver lifeline for the *risky message* and the *deleted message* categories. The receiver lifeline of an unwanted incident message is *asset lifeline*, because the purpose of an unwanted incident message is to denote that an unwanted incident has an

impact on an asset.

$$\begin{aligned}
\text{transmitter lifeline} &= \text{general lifeline} \mid \text{deliberate threat lifeline} \\
&\quad \mid \text{accidental threat lifeline} \mid \text{non-human threat lifeline}; \\
\text{receiver lifeline} &= \text{general lifeline} \mid \text{deliberate threat lifeline} \\
&\quad \mid \text{accidental threat lifeline} \mid \text{non-human threat lifeline}; \\
\text{general lifeline} &= \mathbf{gl}(\text{identifier}); \\
\text{deliberate threat lifeline} &= \mathbf{dtl}(\text{identifier}); \\
\text{accidental threat lifeline} &= \mathbf{atl}(\text{identifier}); \\
\text{non-human threat lifeline} &= \mathbf{ntl}(\text{identifier});
\end{aligned}$$

### C. Risk-measure annotations

In the following, we define the syntax of the risk-measure annotations in CORAL. Frequencies may be assigned on the transmission and the reception of risky messages, as well as on the transmission of unwanted incident messages. Conditional ratios are assigned only on risky messages, and consequences are assigned only on unwanted incident messages. Deleted messages have no risk-measure annotations. CORAL allows the assignment of exact frequencies, as well as frequency intervals. An exact frequency may for example be expressed as “10:1y” meaning “10 occurrences per year”, while a frequency interval may be expressed as “[10,50]:1y” meaning “from and including 10 up to and including 50 occurrences per year”. CORAL also allows the assignment of exact conditional ratios and conditional ratio intervals. An exact conditional ratio is simply a non-negative real number (including zero), while a conditional ratio interval is an interval of non-negative real numbers (including zero).

$$\begin{aligned}
\text{transmission frequency} &= \text{frequency}; \\
\text{reception frequency} &= \text{frequency}; \\
\text{frequency} &= \mathbf{f}(\text{exact}, \text{time unit}) \mid \mathbf{f}(\text{interval}, \text{time unit}); \\
\text{conditional ratio} &= \mathbf{cr}(\text{exact}) \mid \mathbf{cr}(\text{interval}); \\
\text{consequence} &= \mathbf{c}(\text{identifier});
\end{aligned}$$

### D. Interaction operators

In the following, we define the syntax of the interaction operators in CORAL. The interaction operators *seq*, *alt*, *ref* and *par* are discussed in Section IV. According to UML, there are three syntactical definitions of the interaction operator *loop* depending on whether there are no integers, one integer, or a pair of a maximum and a minimum integer given together with the operator [5] (pp. 485–486). If only *loop* is given, the operand represents a loop with zero as lower bound and infinity as upper bound. If *loop* is accompanied by an integer *int*, the operand represents a loop that loops exactly *int* times. Finally, if *loop* is accompanied by two integers, *minint* and *maxint*, the operand represents a loop that loops minimum *minint* times and maximum *maxint* times.

In EBNF, “{ }<sup>-</sup>” means an ordered sequence of one or more repetitions of the enclosed element [6]. This means that the interaction operators *seq*, *alt* and *par* may consist of an ordered sequence of one or more risk interactions. The term *risk interaction* is defined initially in this appendix.

$$\begin{aligned}
\text{weak sequencing} &= \mathbf{seq}(\{\text{risk interaction}\}^-); \\
\text{potential alternatives} &= \mathbf{alt}(\{\text{risk interaction}\}^-); \\
\text{referred interaction} &= \mathbf{ref}(\text{identifier}); \\
\text{parallel execution} &= \mathbf{par}(\{\text{risk interaction}\}^-); \\
\text{loop} &= \mathbf{loop}(\text{risk interaction}) \mid \mathbf{loop}(\text{int}, \text{risk interaction}) \\
&\quad \mid \mathbf{loop}(\text{minint}, \text{maxint}, \text{risk interaction});
\end{aligned}$$

APPENDIX B.  
ENGLISH-PROSE SEMANTICS OF CORAL

In this appendix, we define the English-prose semantics for the CORAL language. The English-prose semantics is defined by a function  $\llbracket \cdot \rrbracket$  that takes a syntactical element as input, expressed in the textual syntax defined with EBNF in Appendix A, and provides English prose of the syntactical element.

This appendix is structured similar to Appendix A; we define the English-prose semantics for messages, lifelines, risk-measure annotations and interaction operators in the same order.

*A. Messages*

In the following, we define the English-prose semantics for the five different messages in CORAL. The syntax of messages is defined in Appendix A-A. Let the syntactical variables

- $id$  range over *identifier*
- $t$  range over *transmitter lifeline*
- $r$  range over *receiver lifeline*
- $al$  range over *asset lifeline*
- $f$  range over *frequency*
- $cr$  range over *conditional ratio*
- $c$  range over *consequence*

$\llbracket \mathbf{rm}(id, t, r, \mathbf{general}, f_1, cr, f_2) \rrbracket$  = The message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  $\llbracket r \rrbracket$   $\llbracket f_1 \rrbracket$ ,  
the transmission leads to its reception  $\llbracket cr \rrbracket$ ,  
and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \mathbf{rm}(id, t, r, \mathbf{new}, f_1, cr, f_2) \rrbracket$  = The new message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  $\llbracket r \rrbracket$   $\llbracket f_1 \rrbracket$ ,  
the transmission leads to its reception  $\llbracket cr \rrbracket$ ,  
and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \mathbf{rm}(id, t, r, \mathbf{alter}, f_1, cr, f_2) \rrbracket$  = The altered message  $id$  is transmitted from  $\llbracket t \rrbracket$  to  $\llbracket r \rrbracket$   $\llbracket f_1 \rrbracket$ ,  
the transmission leads to its reception  $\llbracket cr \rrbracket$ ,  
and the reception occurs  $\llbracket f_2 \rrbracket$ .

$\llbracket \mathbf{uim}(id, t, al, f, c) \rrbracket$  = The unwanted incident  $id$  occurs on  $\llbracket t \rrbracket$   $\llbracket f \rrbracket$ ,  
and impacts asset  $al$   $\llbracket c \rrbracket$ .

$\llbracket \mathbf{dm}(id, t, r) \rrbracket$  = The message  $id$  transmitted from  $\llbracket t \rrbracket$  to  $\llbracket r \rrbracket$  is deleted.

*B. Lifelines*

In the following, we define the English-prose semantics for the lifelines in CORAL. The syntax of lifelines is defined in Appendix A-B. Let the syntactical variable

- $id$  range over *identifier*

$\llbracket \mathbf{gl}(id) \rrbracket = id$

$\llbracket \mathbf{dtl}(id) \rrbracket = \text{the deliberate threat } id$

$\llbracket \mathbf{atl}(id) \rrbracket = \text{the accidental threat } id$

$\llbracket \mathbf{ntl}(id) \rrbracket = \text{the non-human threat } id$



### C. Risk-measure annotations

In the following, we define the English-prose semantics for the risk-measure annotations in CORAL. The syntax of risk-measure annotations is defined in Appendix A-C. Let the syntactical variables

- $id$  range over *identifier*
- $e$  range over *exact*
- $i$  range over *interval*
- $tu$  range over *time unit*

Undefined values are represented by  $\perp$ .

$$\begin{aligned} \llbracket \mathbf{f}(e, tu) \rrbracket &= \text{with frequency } e \text{ per } tu \\ \llbracket \mathbf{f}(i, tu) \rrbracket &= \text{with frequency interval } i \text{ per } tu \\ \llbracket \mathbf{f}(\perp, \perp) \rrbracket &= \text{with undefined frequency} \end{aligned}$$

$$\begin{aligned} \llbracket \mathbf{cr}(e) \rrbracket &= \text{with conditional ratio } e \\ \llbracket \mathbf{cr}(i) \rrbracket &= \text{with conditional ratio interval } i \\ \llbracket \mathbf{cr}(\perp) \rrbracket &= \text{with undefined conditional ratio} \end{aligned}$$

$$\begin{aligned} \llbracket \mathbf{c}(id) \rrbracket &= \text{with consequence } id \\ \llbracket \mathbf{c}(\perp) \rrbracket &= \text{with undefined consequence} \end{aligned}$$

### D. Interaction operators

In the following, we define the English-prose semantics for the interaction operators in CORAL. The syntax of interaction operators is defined in Appendix A-D. Let the syntactical variables

- $d$  range over *risk interaction*
- $id$  range over *identifier*
- $x$  range over *int*
- $a$  range over *minint*
- $b$  range over *maxint*

The pair of square brackets, '[' and ']', is a part of the resulting English-prose semantics and it is used to enclose an operand.

$$\llbracket \mathbf{seq}(d_1, d_2, \dots, d_m) \rrbracket = [ \llbracket d_1 \rrbracket ] \text{ weakly sequenced by } [ \llbracket d_2 \rrbracket ] \text{ weakly sequenced by } \dots \text{ weakly sequenced by } [ \llbracket d_m \rrbracket ]$$

$$\llbracket \mathbf{alt}(d_1, d_2, \dots, d_m) \rrbracket = \text{either } [ \llbracket d_1 \rrbracket ] \text{ or } [ \llbracket d_2 \rrbracket ] \text{ or } \dots \text{ or } [ \llbracket d_m \rrbracket ]$$

$$\llbracket \mathbf{ref}(id) \rrbracket = \text{Refer to interaction: } id.$$

$$\llbracket \mathbf{par}(d_1, d_2, \dots, d_m) \rrbracket = [ \llbracket d_1 \rrbracket ] \text{ parallelly merged with } [ \llbracket d_2 \rrbracket ] \text{ parallelly merged with } \dots \text{ parallelly merged with } [ \llbracket d_m \rrbracket ]$$

$$\llbracket \mathbf{loop}(d) \rrbracket = \text{loop minimum zero times and maximum infinitely } [ \llbracket d \rrbracket ]$$




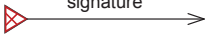
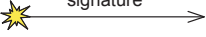
$$\llbracket \mathbf{loop}(x, d) \rrbracket = \text{loop exactly } x \text{ times } [ \llbracket d \rrbracket ]$$

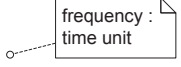

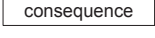
$$\llbracket \mathbf{loop}((a, b), d) \rrbracket = \text{loop minimum } a \text{ times and maximum } b \text{ times } [ \llbracket d \rrbracket ]$$


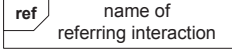


## APPENDIX C.

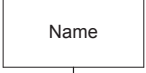




### OVERVIEW OF THE GRAPHICAL NOTATION OF CORAL

Figure 5 shows an overview of the graphical notation of the CORAL language.

Messages	
Node type	Notation
General message	 →
New message	 →
Altered message	 →
Deleted message	 →
Unwanted incident message	 →

Risk-measure annotations	
Node type	Notation
Frequency	
Conditional ratio	
Consequence	

Interaction operators	
Node type	Notation
Potential alternatives	
Referred interaction	
Parallel execution	
Loop	

Lifelines	
Node type	Notation
General lifeline	
Deliberate threat lifeline	
Accidental threat lifeline	
Non-human threat lifeline	
Asset lifeline	

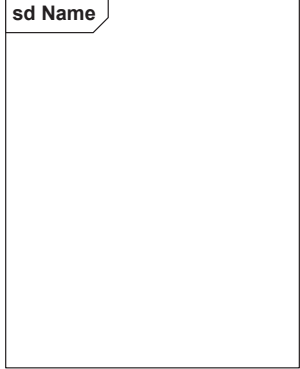
CORAL diagram frame	
Node type	Notation
Frame	

Figure 5. Graphical Notation of CORAL.





Technology for a better society

[www.sintef.no](http://www.sintef.no)

# Chapter 12

Paper 4: Evaluation of the CORAL approach for risk-driven security testing based on an industrial case study



# Report

## Evaluation of the CORAL Approach for Risk-Driven Security Testing Based on an Industrial Case Study

**Author(s)**

Gencer Erdogan<sup>1</sup>, Ketil Stølen<sup>1</sup>, and Jan Øyvind Aagedal<sup>2</sup>

<sup>1</sup> Department for Networked Systems and Services, SINTEF ICT, PO Box 124 Blindern, N-0314 Oslo, Norway

<sup>2</sup> Accurate Equity, Martin Linges vei 25, N-1364 Fornebu, Norway

SINTEF IKT  
SINTEF ICT

Address:  
Postboks 124 Blindern  
NO-0314 Oslo  
NORWAY

Telephone:+47 73593000  
Telefax:+47 22067350

postmottak.IKT@sintef.no  
www.sintef.no  
Enterprise /VAT No:  
NO 948 007 029 MVA

# Report

## Evaluation of the CORAL Approach for Risk-Driven Security Testing Based on an Industrial Case Study

### KEYWORDS:

Case study,  
Risk-driven,  
Risk-based,  
Security testing

VERSION  
Final

DATE  
2015-07-28

AUTHOR(S)  
Gencer Erdogan, Ketil Stølen, and Jan Øyvind Aagedal

CLIENT(S)  
Norwegian Research Council

CLIENT'S REF.  
201579/S10

PROJECT NO.  
102002253

NUMBER OF PAGES/APPENDICES:  
20/0

### ABSTRACT

The CORAL approach is a model-based method to security testing employing risk assessment to help security testers select and design test cases based on the available risk picture. In this report we present experiences from using CORAL in an industrial case. The results indicate that CORAL supports security testers in producing risk models that are valid and threat scenarios that are directly testable. This, in turn, helps testers to select and design test cases according to the most severe security risks posed on the system under test.

PREPARED BY  
Gencer Erdogan

SIGNATURE



CHECKED BY  
Atle Refsdal

SIGNATURE



APPROVED BY  
Fredrik Seehusen

SIGNATURE



REPORT NO. SINTEF A27097  
ISBN 978-82-14-05907-6

CLASSIFICATION  
Unrestricted

CLASSIFICATION THIS PAGE  
Unrestricted



## Table of Contents

1	Introduction.....	4
2	The CORAL Approach.....	4
3	Research Method.....	6
4	Overview of Industrial Case Study.....	7
	4.1 Test Planning (Phase 1).....	7
	4.2 Security Risk Assessment (Phase 2).....	8
	4.3 Security Testing (Phase 3).....	11
5	Case Study Results.....	12
6	Discussion.....	14
7	Related Work and Conclusions.....	15

## 1 Introduction

Security testers face the problem of determining the tests that are most likely to reveal severe security vulnerabilities. Risk-driven security testing has been proposed in response to this challenge. Potter and McGraw [20] argue that security testers must use a risk-driven approach to security testing, because by identifying risks in the system and creating tests driven by those risks, a security tester can properly focus on aspects of the system in which an attack is likely to succeed. Unfortunately, only a handful approaches to risk-driven testing specifically address security, and the field is immature and needs more formality and preciseness [4].

We have developed a method for risk-driven security testing supported by a domain-specific language which we refer to as the CORAL approach, or just CORAL [5, 6]. It aims to help security testers to select and design test cases based on the available risk picture.

In this report we present experiences from applying CORAL in an industrial case. We evaluate to what extent the CORAL approach helps security testers in selecting and designing test cases. The system under test is a comprehensive web-based e-business application designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans, and is used by a large number of customers across Europe. The system owner, which is also the party that commissioned the case study (often referred to as party in the following), require full confidentiality. The results presented in this report are therefore limited to the experiences from applying the CORAL approach.

The report is organized as follows. In Sect. 2 we give an overview the CORAL approach. In Sect. 3 we present our research method. In Sect. 4 we give an overview of the case study, and in Sect. 5 we present the obtained results organized according to our research questions. In Sect. 6 we discuss these results, and finally in Sect. 7 we relate our work to other approaches and conclude by highlighting key findings.

## 2 The CORAL Approach

The CORAL approach consists of a domain-specific risk analysis language and a method for risk-driven security testing within which the language is tightly integrated. Figure 1 presents an example of a risk model expressed in the CORAL language. The dashed arrows are not part of the model, but used to point out the various constructs explained below.

A *threat* is a potential cause of an unwanted incident. A *threat scenario* is a chain or series of events that is initiated by a threat and that may lead to an unwanted incident. An *asset* is something to which the party on whose behalf we are testing assigns value and hence for which the party requires protection. A *new message* is a message initiated by a threat, and is represented by a red triangle at the transmission end. An *altered message* is a message in the system under test (SUT) that has been altered by a threat to deviate from its expected behavior.

It is represented by a triangle with red borders and white fill. A *deleted message* is a message in the SUT that has been deleted by a threat. It is represented by a triangle with red borders and a red cross in the middle. An *unwanted incident* is a message modeling that an asset is harmed or its value is reduced. It is represented by a yellow explosion sign. A *frequency* is the frequency of either the transmission or the reception of a message. A *conditional ratio* is the ratio by which a message is received, given that it is transmitted. A *consequence* is the consequence an unwanted incident has on an asset. A risk, in our approach, is the frequency of an unwanted incident and its consequence for a specific asset.

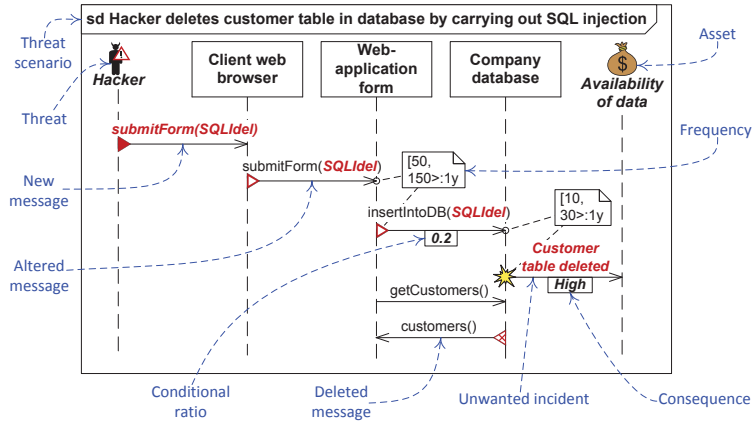


Fig. 1. Example of a CORAL risk model.

As illustrated in Fig. 2, the CORAL method expects a description of the SUT as input. The description may be in the form of system diagrams and models, use case documentation, source code, executable versions of the system, and so on. The CORAL method involves seven steps grouped into three phases: Test planning, security risk assessment, and security testing. The output from applying CORAL is a test report.

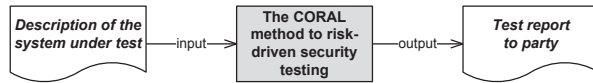


Fig. 2. Input and output of the CORAL method.

In Phase 1, we identify security assets to be protected, define frequency and consequence scales, and define the risk evaluation matrix based on the frequency and consequence scales.

In Phase 2, we carry out the risk modeling in three consecutive steps. First, we identify security risks by analyzing the models of the SUT with respect to the security assets. Then we identify threat scenarios that may cause the security risks. Second, we estimate the frequency of the identified threat scenarios and risks by making use of the frequency scale, and the consequence of risks by making use of the consequence scale. Third, we evaluate the risks with respect to their frequency and consequence estimates and select the most severe risks to test.

In Phase 3, we conduct security testing in three consecutive steps. First, for each risk selected for testing, we identify the threat scenario in which the risk occurs and specify a test objective for that threat scenario. To obtain a test case fulfilling the test objective, we use stereotypes from the UML Testing Profile [17] to annotate the threat scenario with respect to the test objective. Second, we carry out security testing with respect to the test cases. The test cases may be executed manually, semi automatically, or automatically. Third, based on the test results, we write a test report.

### 3 Research Method

As illustrated by Fig. 3 we conducted the case study in four main steps. First, we designed the case study by defining the objective, the units of analysis, as well as the research questions. Second, we carried out the CORAL approach within an industrial setting. Third, we collected the relevant data produced by executing the CORAL approach. Fourth, we analyzed the collected data with respect to our research questions. This research approach is inspired by the guidelines for case study research in software engineering provided by Runeson et al. [22].

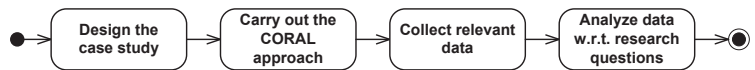


Fig. 3. The main activities of the research method.

As pointed out in Sect. 1, the objective of the case study was to evaluate to what extent the CORAL approach helps security testers in selecting and designing test cases. The test report delivered to the party that commissioned the case study describes, in addition to the test results, risk models and security tests designed with respect to the risk models. Our hypothesis was that the report is good in the sense that (1) the risk models are valid, and (2) the threat scenarios are directly testable. By a directly testable threat scenario, we mean

a threat scenario that can be reused and specified as a test case based on its interactions. Thus, the units of analysis in this case study are the risk models.

With respect to point (1), we defined two research questions (RQ1 and RQ2). With respect to point (2), we defined one research question (RQ3). Additionally, we carried out both black-box and white-box testing of the SUT, because we were interested in investigating the usefulness of the CORAL approach for both black-box and white-box testing (RQ4).

**RQ1** To what extent is the risk level of identified risks correct?

**RQ2** To what extent are relevant risks identified compared to previous penetration tests?

**RQ3** To what extent are the threat scenarios that causes the identified risks directly testable?

**RQ4** To what extent is the CORAL approach useful for black-box testing and white-box testing, respectively?

## 4 Overview of Industrial Case Study

As mentioned in Sect. 1, the system under test was a web-based application providing services related to equity-based compensation plans. The web application was deployed on the servers of a third party service provider and maintained by the same service provider with respect to infrastructure. However, the web application was completely administrated by the client commissioning the case study for business purposes, such as customizing the web application for each customer, as well as patching and updating various features of the web application.

In order to limit the scope of the testing, we decided to test two features available to customers: a feature for selling shares (named Sell Shares), and a feature for exercising options for the purpose of buying shares in a company (named Exercise Options). In the following, we explain how we carried out the CORAL approach by taking you through a fragment of the case study. We consider only Exercise Options and two potential threat scenarios: one from a black-box perspective and one from a white-box perspective.

### 4.1 Test Planning (Phase 1)

We modeled Exercise Options from a black-box perspective by observing its behavior. That is, we executed Exercise Options using a web browser, observed its behavior, and created the model based on that. We also modeled Exercise Options from a white-box perspective by executing and analyzing its source code. Figures 4a and 4b show the black-box model and the white-box model of Exercise Options, respectively.

Together with the party we decided not to consider security risks related to infrastructure because this was a contractual responsibility of the service provider hosting the web application. Instead, we focused on security risks that may be

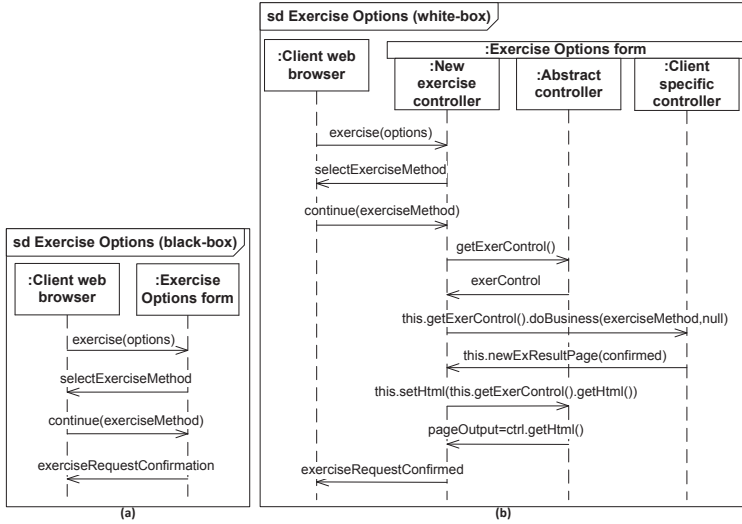


Fig. 4. (a) Black-box model of feature Exercise Options. (b) White-box model of feature Exercise Options.

introduced via the application layer. Thus, the threat profile is someone who has access to Exercise Options, but who resides outside the network boundaries of the service provider. We identified security assets by consulting the party. The security asset identified for Exercise Options was *integrity of data*.

We also defined a frequency scale and a consequence scale together with the party. The frequency scale consisted of five values (Certain, Likely, Possible, Unlikely, and Rare), where each value was defined as a frequency interval. For example, the frequency interval for likelihood Possible was  $[5,20):1y$ , which means “from and including 5 to less than 20 times per year.” The consequence scale also consisted of five values (Catastrophic, Major, Moderate, Minor, and Insignificant), where each value described the impact by which the security asset is harmed. For example, consequence Major with respect to security asset *integrity of data* was defined as “the integrity of customer shares is compromised.” The scales were also used to construct the risk evaluation matrix illustrated in Fig. 7.

## 4.2 Security Risk Assessment (Phase 2)

We identified security risks by analyzing the black-box and white-box models of Exercise Options with respect to security asset *integrity of data*. We did this by first identifying unwanted incidents that have an impact on the security asset.

Second, we identified alterations in the messages that had to take place in order to cause the unwanted incidents (to be represented as altered messages). Third, we identified messages initiated by the threat which in turn could cause the alterations (to be represented as new messages).

Let us consider a threat scenario for the black-box model of Exercise Options. Assume that a malicious user attempts to access another system feature, say an administrative functionality, by altering certain parameters in the HTTP request sent to Exercise Options. The malicious user could achieve this, for example, by first intercepting the request containing the message *continue(exerciseMethod)* using a network proxy tool such as OWASP ZAP [19], and then altering the parameter *exerciseMethod* in the message. This alteration, could in turn give the malicious user access to another system feature. This unwanted incident occurs if the alteration is successfully carried out, and Exercise Options responds with another system feature instead of the expected message *exerciseRequestConfirmation*. Thus, the unwanted incident may occur after the reception of the last message in the black-box model (Fig. 4a). The resulting threat scenario is shown in Fig. 5. We carried out a similar analysis during white-box testing by analyzing the model in Fig. 4b. The resulting threat scenario for the white-box model is shown in Fig. 6.

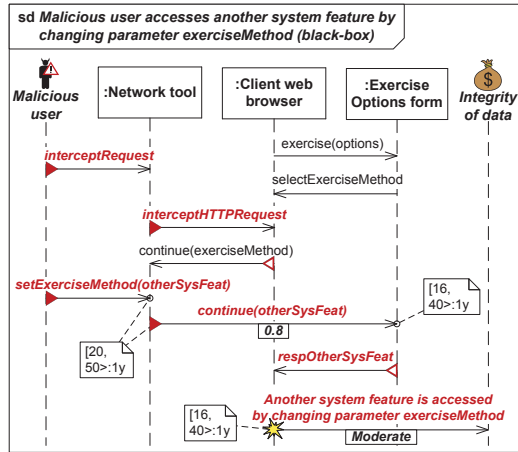


Fig. 5. A threat scenario for the black-box model of feature Exercise Options.

In order to estimate how often threat scenarios may occur, in terms of frequency, we based ourselves on knowledge data bases such as OWASP [18], reports and papers published within the software security community, as well as expert knowledge within security testing. We see from Fig. 5 that the malicious

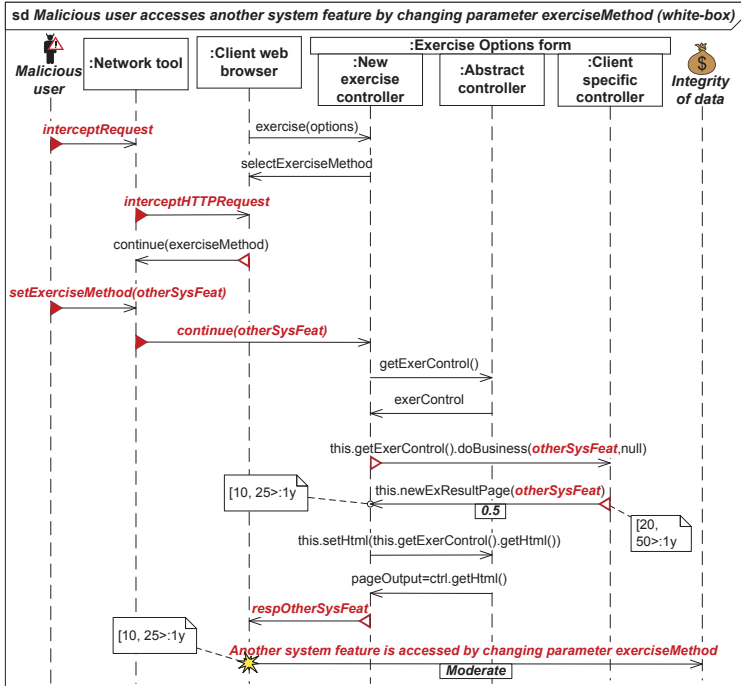


Fig. 6. A threat scenario for the white-box model of feature Exercise Options.

user successfully alters the parameter *exerciseMethod* with frequency  $[20, 50]:1y$ . Given that parameter *exerciseMethod* is successfully altered and transmitted, it will be received by Exercise Options with conditional ratio  $0.8$ . The conditional ratio causes the new frequency  $[16, 40]:1y$  for the reception of message *continue(otherSysFeat)*. This is calculated by multiplying  $[20, 50]:1y$  with  $0.8$ . Given that message *continue(otherSysFeat)* is processed by Exercise Options, it will respond with another system feature. This, in turn, causes the unwanted incident (security risk) to occur with frequency  $[16, 40]:1y$ . The unwanted incident has an impact on security asset *integrity of data* with consequence *Moderate*.

Figure 7 shows the obtained risk evaluation matrix. The numbers in the matrix represent the 21 risks identified in the case study. Each risk was plotted in the matrix according to its frequency and consequence estimate. Risks are grouped in nine levels horizontally on the matrix where Risk Level 1 is the lowest risk level and Risk Level 9 is the highest risk level. The risk level of a risk is identified by mapping the underlying color to the column on the left-hand side



of the matrix. For example, Risks 11 and 19 have Risk Level 8, while Risk 20 has Risk Level 4. The risk aggregation did not lead to an increase in risk level for any of the risks. The suspension criterion in this case study was defined as “test all risks of Risk Level 6 or more.” Based on this criterion, we selected 11 risks to test from the risk evaluation matrix.

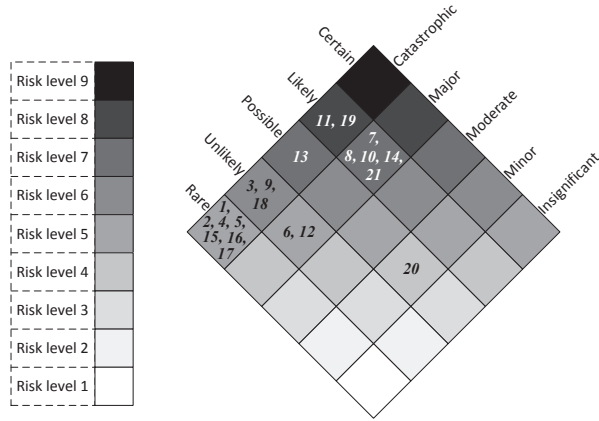


Fig. 7. Risk evaluation matrix.

### 4.3 Security Testing (Phase 3)

The test objective for the threat scenarios in Figs. 5 and 6 was defined as: “Verify whether the malicious user is able to access another system feature by changing parameter *exerciseMethod* into a valid system parameter”. Based on this test objective, we annotated the threat scenarios with stereotypes from the UML testing profile [17]. For example, the resulting security test case for the threat scenario in Fig. 5 is shown in Fig. 8. Needless to say, the security tester takes the role as “malicious user” in the test case.

We carried out all black-box tests manually and used the OWASP Zed Attack Proxy tool [19] to intercept the HTTP requests and responses. We carried out all white-box tests semi automatically supported by the debug mode in Eclipse IDE, which was integrated with a web-server and a database. We also carried out automatic source code review using static source code analysis tools for the purpose of identifying potential threat scenarios. The tools we used for this purpose were Find Security Bugs V1.2.1 [7], Lapse plus V2.8.1 [13], and Visual Code Grepper (VCG) V2.0.0 [24].

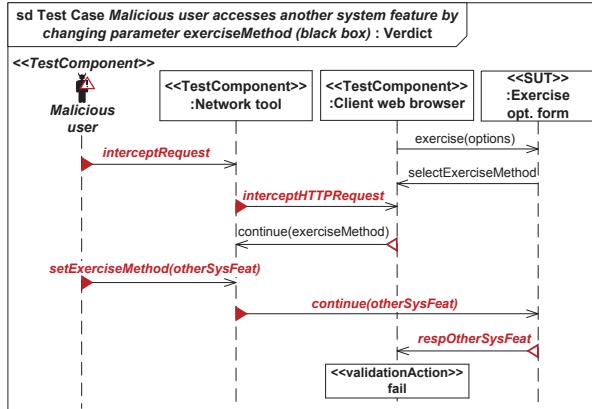


Fig. 8. Security test case based on the threat scenario in Fig. 5.

## 5 Case Study Results

In this section we present the results from the case study. We group the results with respect to our research questions.

**RQ1 To what extent is the risk level of identified risks correct?** As shown in the risk evaluation matrix in Fig. 7, we identified in total 21 security risks. The risk aggregation did not lead to an increase in risk level for any of the risks. Based on the suspension criterion defined by the party, we tested 11 risks (all risks of Risk Level 6 or more).

The testing of these 11 risks revealed 11 vulnerabilities. The vulnerabilities were assigned a severity level based on a scale of three values (Low, Medium, and High). High severity means that the vulnerability should be treated as soon as possible, and one should consider taking the system offline while treating the vulnerability (“show stopper”). Medium severity means that the vulnerability is serious and should be treated based on available time and resources. Low severity means that the vulnerability is not serious and it should be treated if seen necessary. Four vulnerabilities were assigned severity Medium, and the remaining 7 vulnerabilities were assigned severity Low.

We also tested the 10 risks initially not selected for testing, in order to have a basis for comparison. This testing revealed only 2 vulnerabilities of severity Low.

**RQ2 To what extent are relevant risks identified compared to previous penetration tests?** The party commissioning the case study had previously executed commercial penetration tests. We did not get access to the reports or results from these penetration tests due to confidentiality reasons. However, it was confirmed by the party that we had identified 16 security risks

which had also been identified by the previous penetration tests. Additionally, we had identified 5 new security risks which had not been identified by the previous penetration tests. Moreover, the party also confirmed that we had not left out any risks of relevance for the features considered in the case study.

**RQ3 To what extent are the threat scenarios that causes the identified risks directly testable?** The identified 21 security risks were caused by 31 threat scenarios. The 11 risks initially selected for testing were caused by 18 threat scenarios, while the remaining 10 risks were caused by 13 threat scenarios. We identified 18 security test cases based on the 18 threat scenarios causing the 11 risks. Similarly, we identified 13 security test cases based on the 13 threat scenarios causing the remaining 10 risks.

**RQ4 To what extent is the CORAL approach useful for black-box testing and white-box testing, respectively?** Table 1 gives an overview of the results obtained during black-box testing and white-box testing. The row “risks tested” represents the number of risks initially selected for testing, as well as those not initially selected for testing (in parentheses). The row “vulnerabilities identified” represents the number of vulnerabilities identified by testing the risks initially selected for testing, as well as the number of vulnerabilities identified by testing the risks not initially selected (in parentheses). The four rows at the bottom of Table 1 provide statistics on the use of the various modeling constructs of CORAL to express the threat scenarios.

**Table 1.** Results obtained during black-box testing and white-box testing.

	<b>Black-box</b>	<b>White-box</b>	<b>Total</b>
<b>SUT diagrams analyzed</b>	11	2	13
<b>Threat scenarios identified</b>	27	4	31
<b>Risks identified</b>	19	2	21
<b>Test cases identified</b>	27	4	31
<b>Risks testes</b>	10 (plus 9)	1 (plus 1)	11 (plus 10)
<b>Vulnerabilities identified</b>	4 medium and 5 low (plus 0)	2 low (plus 2 low)	4 medium and 7 low (plus 2 low)
<b>New messages</b>	144	17	161
<b>Altered messages</b>	52	8	60
<b>Deleted messages</b>	10	7	17
<b>Unwanted incidents</b>	30	4	34

## 6 Discussion

The two variables that determine the risk level of a risk, that is, the frequency value and the consequence value, are estimates based on data gathered during the security risk assessment. In other words, these estimates tell us to what degree the identified risks exist. Thus, in principle, the higher the risk level, the more likely it is to reveal vulnerabilities causing the risk. The same applies the other way around. That is, the lower the risk level, the less likely it is to reveal vulnerabilities causing the risk.

The results obtained for RQ1 show that 11 vulnerabilities were revealed by testing the risks considered as most severe, while only 2 vulnerabilities were revealed by testing the risks considered as low risks. Additionally, the 2 vulnerabilities identified by testing the low risks were assigned low severity (see Table 1). These findings indicate that the risk levels of identified risks were quite accurate. In contrast, if we had found 2 vulnerabilities by testing the most severe risks, and 11 vulnerabilities by testing the low risks, then that would have indicated inaccurate risk levels, and thus a risk assessment of low quality. The results obtained for RQ2 show that we identified all relevant security risks compared to previous penetration tests. In addition, we identified five new security risks and did not leave out any risks of relevance for the features considered. In summary, the results obtained for RQ1 and RQ2 indicate that the produced risk models were valid and of high quality, and thus that the CORAL approach is effective in terms of producing valid risk models.

The results obtained for RQ3 point out that all threat scenarios were directly testable. We believe this result is generalizable because, in the CORAL approach, risks are identified at the level of abstraction testers commonly work when designing test cases [3]. This is also backed up by the fact that we made direct use of all threat scenarios as security test cases. Thus, the CORAL approach is effective in terms of producing threat scenarios that are directly testable. However, it is important to note that the CORAL approach is designed to be used by individual security testers, or by a group of security testers collaborating within the same testing project. The risk models produced by a tester, or a group of testers working together, will most likely be used by the same tester(s) to design test cases, and consequently execute the test cases.

In general, the CORAL approach seems to work equally well for black-box testing and white-box testing. Based on the results obtained for RQ4, we see that it is possible to carry out the complete CORAL approach both in black-box testing and white-box testing. The reason why Table 1 shows lower numbers in the white-box column compared to the numbers in the black-box column, is because we had fewer white-box models to analyze compared to black-box models.

Table 1 also shows that the threat scenarios mostly consisted of *new messages* and *altered messages*. Only 17 out of 272 messages were *deleted messages*. This may be an indication that threat scenarios can be sufficiently expressed without the usage of deleted messages. Nevertheless, they are important to document that an interaction is deleted by a threat. Note that the number of unwanted

incidents (34) is greater than the number of identified risks (21). This is because some of the risks reoccurred in several threat scenarios, and thus had to be repeated in every threat scenario in which they reoccurred.

Another important observation is that the threat scenarios identified during white-box testing helped us locate where in the source code risks occurred, although the threat scenarios were initiated at the application level.

## 7 Related Work and Conclusions

In order to get a holistic picture, we relate the CORAL approach to other risk-driven testing approaches at a general level and not only to approaches focusing on security. In addition, because the CORAL approach is a model-based approach, we will relate our approach to other model-based approaches. The reader is referred to our systematic literature review for an overview of state of the art risk-driven testing approaches, where we also review approaches that are not model-based [4].

Most model-based approaches make use of fault tree analysis (attack tree is a variant of a fault tree) for the purpose of safety risk assessment [2, 8, 12, 15, 21, 27]. While fault tree analysis in these approaches are useful for identifying specific safety risks, they do not include information such as the threat profile initiating the chain of events causing the risks, the likelihood of an event occurring, and the consequence of risks. These constructs are necessary in a risk assessment [10]. Moreover, these approaches do not provide any guidelines for estimating or evaluating risks. The approaches leave it to the tester to identify the most severe risks in an ad hoc manner, and plan the testing process accordingly. In these approaches, test cases are designed by first analyzing the fault tree diagrams, and then modeling state machine diagrams (similar to UML state machines [16]) that represent test cases exploring the faults. However, there are some existing gaps between fault trees and state machines that need to be taken into consideration when modeling state machine diagrams based on fault tree diagrams [11]. In addition, while these approaches focus on modeling state-based test cases, our approach focuses on modeling interaction-based test cases.

The risk-driven security testing approaches provided by Botella et al. [1], Großmann et al. [9], and Seehusen [23] make use of the CORAS risk analysis language [14] for the purpose of security risk assessment. The graphical notation of the CORAL risk analysis language is based on the CORAS language. The CORAL approach is therefore closely related to these approaches. However, there are some fundamental differences. First, CORAS risk models represent threat scenarios and risks at a high-level of abstraction, while we represent these at a low-level of abstraction. Second, CORAS risk models are represented as directed acyclic graphs, while we represent risk models as sequence diagrams, which are better suited for model-based testing [3]. Third, these approaches use the risk estimates assigned on a CORAS risk model to make a prioritized list of threat scenarios which in turn represent a prioritized list of high-level test procedures [23]. The high-level test procedures are then used as a starting point

for identifying/designing test cases either manually or by instantiating certain test patterns. In the CORAL approach, we map the risks to a risk evaluation matrix based on the risk estimates, and then we make a prioritized list of risks. We then select the most severe risks that the system under test is exposed to, and design test cases by making use of the CORAL risk models in which the selected risks occur.

The approach provided by Wendland et al. [25] makes use of behavior trees for the purpose of identifying risks at the level of requirements engineering. This approach does not identify risks by modeling threat scenarios, but rather by carrying out high-level qualitative risk assessment, and annotate the behavior trees with qualitative risk exposure values. The approach does not explicitly model test cases, but instead provides guidelines (referred to as test design strategy) testers may use to model test cases.

Zech et al. [26] identify security risks solely by matching attack patterns on the public interfaces of a system under test. The pattern matching is carried out automatically, which in turn produces risk models in terms of UML class diagrams [16]. The produced risk models contain information about possible attack scenarios that may be carried out on the public interfaces. However, the risk models do not contain information regarding the threat initiating the attacks, and the chain of events causing the security risks. The approach transforms the risk models into misuse case models, represented as class diagrams, from which test cases are generated.

What is common for all the approaches discussed above is that they model risks and the system under test in separate models using separate modeling languages. This makes it difficult to get an intuitive understanding with respect to exactly how and where the risks affect the system under test. The risk models in the CORAL approach represent specific threat scenarios, security risks caused by the threat scenarios, and the relevant aspects of the system affected by the risks, within the same model. This enables testers to identify exactly how and where certain security risks may occur.

In this report, we have presented an evaluation of CORAL based on our experiences from applying the approach in an industrial case study. The SUT in the case study was a web application designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans. The objective of the case study was to evaluate to what extent the CORAL approach helps security testers in selecting and designing test cases. In the CORAL approach, we base the test selection and the test design on the risk models produced during the security risk assessment. Our hypothesis was that the produced risk models are valid, and that the threat scenarios represented by the risk models are directly testable.

The case study results indicate that the CORAL approach is effective in terms of producing valid risk models. This is backed up by two observations. First, we identified in total 21 risks, and 11 of these risks were considered as most severe, while the remaining 10 risks were considered as low risks. By testing these 11 risks we identified 11 vulnerabilities, while by testing the remaining

10 risks we identified only 2 vulnerabilities. Second, we identified all relevant security risks compared to previous penetration tests. In addition, we identified five new security risks and did not leave out any risks of relevance for the features considered.

The CORAL approach seems to work equally well for black-box and white-box testing. One point worth noting for white-box testing is that the threat scenarios help locating risks at the source code level although they are initiated at the application level.

Finally, one of the most important findings we did in the case study is that the CORAL approach is very useful for identifying security test cases. We used all threat scenarios identified in the case study for the purpose of security test case design and execution.

**Acknowledgments.** This work has been conducted as a part of the DIAMONDS project (201579/S10) and the AGRA project (236657) funded by the Research Council of Norway, as well as the RASEN project (316853) funded by the European Commission within the 7th Framework Programme.

## References

1. J. Botella, B. Legeard, F. Peureux, and A. Vernotte. Risk-Based Vulnerability Testing Using Security Test Patterns. In *Proc. 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, pages 337–352. Springer, 2014.
2. R. Casado, J. Tuyá, and M. Younas. Testing long-lived web services transactions using a risk-based approach. In *Proc. 10th International Conference on Quality Software (QSIC'10)*, pages 337–340. IEEE Computer Society, 2010.
3. A.C. Dias Neto, R. Subramanyan, M. Vieira, and G.H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *Proc. 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASEL'07)*, pages 31–36. ACM, 2007.
4. G. Erdogan, Y. Li, R.K. Runde, F. Seehusen, and K. Stølen. Approaches for the Combined Use of Risk Analysis and Testing: A Systematic Literature Review. *International Journal on Software Tools for Technology Transfer*, 16(5):627–642, 2014.
5. G. Erdogan, A. Refsdal, and K. Stølen. A Systematic Method for Risk-driven Test Case Design Using Annotated Sequence Diagrams. In *Proc. 1st International Workshop on Risk Assessment and Risk-driven Testing (RISK'13)*, pages 93–108. Springer, 2014.
6. G. Erdogan, A. Refsdal, and K. Stølen. Schematic Generation of English-prose Semantics for a Risk Analysis Language Based on UML Interactions. In *Proc. 2nd International Workshop on Risk Assessment and Risk-driven Testing (RISK'14)*, pages 205–310. IEEE Computer Society, 2014.
7. Find Security Bugs V1.2.1. <http://h3xstream.github.io/find-sec-bugs/>. Accessed April 30, 2015.
8. M. Gleirscher. Hazard-based selection of test cases. In *Proc. 6th International Workshop on Automation of Software Test (AST'11)*, pages 64–70. ACM, 2011.

9. J. Großmann, M. Schneider, J. Viehmann, and M.-F. Wendland. Combining Risk Analysis and Security Testing. In *Proc. 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, pages 322–336. Springer, 2014.
10. International Organization for Standardization. *ISO 31000:2009(E), Risk management – Principles and guidelines*, 2009.
11. H. Kim, W.E. Wong, V. Debroy, and D. Bae. Bridging the Gap Between Fault Trees and UML State Machine Diagrams for Safety Analysis. In *Proc. 17th Asia Pacific Software Engineering Conference (APSEC'10)*, pages 196–205. IEEE Computer Society, 2010.
12. J. Kloos, T. Hussain, and R. Eschbach. Risk-based testing of safety-critical embedded systems driven by Fault Tree Analysis. In *Proc. 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 26–33. IEEE Computer Society, 2011.
13. Lapse Plus Console V2.8.1. <https://code.google.com/p/lapse-plus/>. Accessed April 30, 2015.
14. M.S. Lund, B. Solhaug, and K. Stølen. *Model-Driven Risk Analysis: The CORAS Approach*. Springer, 2011.
15. R. Nazier and T. Bauer. Automated risk-based testing by integrating safety analysis information into system behavior models. In *Proc. 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW'12)*, pages 213–218. IEEE Computer Society, 2012.
16. Object Management Group. *Unified Modeling Language (UML), superstructure, version 2.4.1*, 2011. OMG Document Number: formal/2011-08-06.
17. Object Management Group. *UML Testing Profile (UTP), version 1.2*, 2013. OMG Document Number: formal/2013-04-03.
18. Open Web Application Security Project. [https://www.owasp.org/index.php/Main\\_Page](https://www.owasp.org/index.php/Main_Page). Accessed April 30, 2015.
19. OWASP Zed Attack Proxy. [https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project). Accessed April 30, 2015.
20. B. Potter and G. McGraw. Software Security Testing. *Security & Privacy, IEEE*, 2(5):81–85, 2004.
21. M. Ray and D.P. Mohapatra. Risk analysis: A guiding force in the improvement of testing. *IET Software*, 7:29–46, 2013.
22. P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 2012.
23. F. Seehusen. A Technique for Risk-Based Test Procedure Identification, Prioritization and Selection. In *Proc. 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, pages 277–291. Springer, 2014.
24. Visual Code Grepper V2.0.0. <http://sourceforge.net/projects/visualcodegrepp/>. Accessed April 30, 2015.
25. M.-F. Wendland, M. Kranz, and I. Schieferdecker. A systematic approach to risk-based testing using risk-annotated requirements models. In *Proc. 7th International Conference on Software Engineering Advances (ICSEA'12)*, pages 636–642. IARA, 2012.
26. P. Zech, M. Felderer, and R. Breu. Towards a Model Based Security Testing Approach of Cloud Computing Environments. In *Proc. 6th International Conference on Software Security and Reliability Companion (SERE-C'12)*, pages 47–56. IEEE Computer Society, 2012.



27. F. Zimmermann, R. Eschbach, J. Kloos, and T. Bauer. Risk-based statistical testing: A refinement-based approach to the reliability analysis of safety-critical systems. In *Proc. 12th European Workshop on Dependable Computing (EWDC'09)*, pages 1–8. The Open Archive HAL, 2009.



Technology for a better society

[www.sintef.no](http://www.sintef.no)

# Chapter 13

Paper 5: Assessing the usefulness of testing for validating and correcting security risk models based on two industrial case studies

---