UNIVERSITÄT KOBLENZ-LANDAU

DISSERTATION THESIS

Recovering Security in Model-Based Software Engineering by Context-Driven Co-Evolution

---

by
Jens BÜRGER

Approved Dissertation thesis for the partial fulfilment of the requirements for a Doctor of Natural Sciences (Dr. rer. nat.)

Fachbereich 4: Informatik
Universität Koblenz-Landau

Chair of PhD Board: Prof. Dr. Maria A. Wimmer
Chair of PhD Commission: JProf. Dr. Mario Schaarschmidt
Examiner and Supervisor: Prof. Dr. Jan Jürjens
Further Examiner: Prof. Dr. Timo Kehrer

Date of the doctoral viva: Oct 31th, 2019

Abstract

Software systems have an increasing impact on our daily lives. Many systems process sensitive data or control critical infrastructure. Providing secure software is therefore inevitable. Such systems are rarely being renewed regularly due to the high costs and effort. Oftentimes, systems that were planned and implemented to be secure, become insecure because their context evolves. These systems are connected to the Internet and therefore also constantly subject to new types of attacks. The security requirements of these systems remain unchanged, while, for example, discovery of a vulnerability of an encryption algorithm previously assumed to be secure requires a change of the system design. Some security requirements cannot be checked by the system's design but only at run time. Furthermore, the sudden discovery of a security violation requires an immediate reaction to prevent a system shutdown. Knowledge regarding security best practices, attacks, and mitigations is generally available, yet rarely integrated part of software development or covering evolution.

This thesis examines how the security of long-living software systems can be preserved taking into account the influence of context evolutions. The goal of the proposed approach, *S^2EC^2O*, is to recover the security of model-based software systems using co-evolution.

An ontology-based knowledge base is introduced, capable of managing common, as well as system-specific knowledge relevant to security. A transformation achieves the connection of the knowledge base to the UML system model. By using semantic differences, knowledge inference, and the detection of inconsistencies in the knowledge base, context knowledge evolutions are detected.

A catalog containing rules to manage and recover security requirements uses detected context evolutions to propose potential co-evolutions to the system model which reestablish the compliance with security requirements.

$\text{S}^2\text{EC}^2\text{O}$ uses security annotations to link models and executable code and provides support for run-time monitoring. The adaptation of running systems is being considered as is round-trip engineering, which integrates insights from the run time into the system model.

$\text{S}^2\text{EC}^2\text{O}$ is amended by prototypical tool support. This tool is used to show $\text{S}^2\text{EC}^2\text{O}$'s applicability based on a case study targeting the medical information system *iTrust*.

This thesis at hand contributes to the development and maintenance of long-living software systems, regarding their security. The proposed approach will aid security experts: It detects security-relevant changes to the system context, determines the impact on the system's security and facilitates co-evolutions to recover the compliance with the security requirements.

Zusammenfassung

Softwaresysteme haben einen zunehmenden Einfluss auf unser tägliches Leben. Viele Systeme verarbeiten sensitive Daten oder steuern wichtige Infrastruktur, was die Bereitstellung sicherer Software unabdingbar macht. Derartige Systeme werden aus Aufwands- und Kostengründen selten erneuert. Oftmals werden Systeme, die zu ihrem Entwurfszeitpunkt als sicheres System geplant und implementiert wurden, deswegen unsicher, weil sich die Umgebung dieser Systeme ändert. Dadurch, dass verschiedenste Systeme über das Internet kommunizieren, sind diese auch neuen Angriffsarten stetig ausgesetzt. Die Sicherheitsanforderungen an ein System bleiben unberührt, aber neue Erkenntnisse wie die Verwundbarkeit eines zum Entwurfszeitpunkt als sicher geltenden Verschlüsselungsalgorithmus erzwingen Änderungen am System. Manche Sicherheitsanforderungen können dabei nicht anhand des Designs sondern nur zur Laufzeit geprüft werden. Darüber hinaus erfordern plötzlich auftretende Sicherheitsverletzungen eine unverzügliche Reaktion, um eine Systemabschaltung vermeiden zu können. Wissen über geeignete Sicherheitsverfahren, Angriffe und Abwehrmechanismen ist grundsätzlich verfügbar, aber es ist selten in die Softwareentwicklung integriert und geht auf Evolutionen ein.

In dieser Arbeit wird untersucht, wie die Sicherheit langlebiger Software unter dem Einfluss von Kontext-Evolutionen bewahrt werden kann. Der vorgestellte Ansatz S^2EC^2O hat zum Ziel, die Sicherheit von Software, die modellbasiert entwickelt wird, mithilfe von Ko-Evolutionen wiederherzustellen.

Eine Ontologie-basierende Wissensbasis wird eingeführt, die sowohl allgemeines wie auch system-spezifisches, sicherheitsrelevantes Wissen verwaltet. Mittels einer Transformation wird die Verbindung der Wissensbasis zu UML-Systemmodellen hergestellt. Mit semantischen Differenzen, Inferenz von Wissen und der Erkennung von Inkonsistenzen in der Wissensbasis werden Kontext-Evolutionen erkannt.

Ein Katalog mit Regeln zur Verwaltung und Wiederherstellung von Sicherheitsanforderungen nutzt erkannte Kontext-Evolutionen, um mögliche Ko-Evolutionen für das Systemmodell vorzuschlagen, welche die Einhaltung von Sicherheitsanforderungen wiederherstellen.

$\mathrm{S^2EC^2O}$ unterstützt Sicherheitsannotationen, um Modelle und Code zum Zwecke einer Laufzeitüberwachung zu koppeln. Die Adaption laufender Systeme gegen Bedrohungen wird ebenso betrachtet wie Roundtrip-Engineering, um Erkenntnisse aus der Laufzeit in das System-Modell zu integrieren.

$\mathrm{S^2EC^2O}$ wird ergänzt um eine prototypische Implementierung. Diese wird genutzt, um die Anwendbarkeit von $\mathrm{S^2EC^2O}$ im Rahmen einer Fallstudie an dem medizinischen Informationssystem *iTrust* zu zeigen.

Die vorliegende Arbeit leistet einen Beitrag, um die Entwicklung und Wartung langlebiger Softwaresysteme in Bezug auf ihre Sicherheit zu begleiten. Der vorgestellte Ansatz entlastet Sicherheitsexperten bei ihrer Arbeit, indem er sicherheitsrelevante Änderungen des Systemkontextes erfasst, den Einfluss auf die Sicherheit der Software prüft und Ko-Evolutionen zur Bewahrung der Sicherheitsanforderungen ermöglicht.

Acknowledgements

During my work in academia, I had the opportunity to work with people doing outstanding work in their fields. Countless conversations, workshops, meetings, debates, and results provide an indescribable amount of contributions to how I see and work with software engineering.

Firstly, I would like to express my sincere gratitude to my supervisor Prof. Dr. Jan Jürjens. Jan supported me in all relevant concerns. His relentless endeavor of finding a solution to a problem that arises has driven me forth many times. I appreciate the amount of knowledge, time, and patience he supported my PhD with.

I would also like to thank Prof. Dr. Timo Kehrer for immediately agreeing to be the second correspondent for my thesis. Timo was also available whenever I needed feedback.

I would like to thank all of my colleagues of the SecVolution project, namely Prof. Dr. Jan Jürjens, Prof. Dr. Kurt Schneider, Dr.-Ing. Stefan Gärtner, and Dr. Thomas Ruhroth. I have benefited greatly from your experience in this intense time.

I would like to thank all of the people responsible for the SiLift approach for providing me support.

Furthermore, I would like to thank Dr. Rainer Buchty. Throughout the years, he shared his experience and expertise with me in various aspects and provided me continuous moral support.

I also express my gratitude to Dr. Sven Wenzel, who encouraged me to become a PhD student. He also provided me helpful recommendations and insights through my entire time as PhD student.

I would like to thank Dr. Marco Konersmann, Dr. Volker Riediger, Dr. Daniel Strüber, Shayan Ahmadian, Katharina Großer, Sven Peldszus, and Qusai Ramadan for proofreading various chapters of my thesis and providing me valuable feedback.

My sincere thanks also go to the PhD board of the University of Koblenz-Landau.

I would like to thank my parents, my parents-in-law, and all of my friends for their support, and sympathy for my situation.

Last but not least, I would like to thank Beate, who did everything conceivable for me to be free of any obligations and who cared for me in every successful as well as unpleasant situation.

I would like to express my gratitude towards the community that has founded and continuously maintains LaTeX, with all of its packages and tooling. Especially when combined with numerous tools coming from the UNIX and Linux community, these tools of the trade have been an indispensable part of my daily work at the university and all the more as a researcher.

Finally, my sincere gratitude goes to all scientists and people who strive to make our world a better place. Science is not a matter of opinion. Science matters. Do not let you be pushed aside and continue using evidence-based methods.

Note: This document was built using the LaTeX template *Masters/Doctoral Thesis*[1]. The template is licensed under the creative commons license CC BY-NC-SA 3.0[2]. Version 2.5 (27/8/17) of the template was used.
Version 2.x major modifications were made by *Vel* (vel@latextemplates.com). The template is based on a template by Steve Gunn[3] and Sunil Patel[4].

For this thesis, compared to the template as provided, the title page had to be adapted to meet the requirements of the local promotion regulations. Starting after the heading, the text elements in the centered block where reordered accordingly. The remaining parts of the `titlepage` block were commented out, the additional necessary information in the lower left was inserted using a `large` block.

Regarding the list of abbreviations, the `longtable` in the `.cls` file was commented out to use content provided by the `acronym` package.

In the abstract page design of the `.cls` file, insertion of `Author name` and `Thesis title` was commented out.

The template modified as described on this page has the same license as the unmodified version.

[1]available for download at http://www.LaTeXTemplates.com
[2] http://creativecommons.org/licenses/by-nc-sa/3.0/
[3]http://users.ecs.soton.ac.uk/srg/softwaretools/document/templates/
[4]http://www.sunilpatel.co.uk/thesis-template/

Contents

Abstract iii

Zusammenfassung v

Acknowledgements vii

List of Abbreviations xvii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Challenges . . . 3
- 1.3 Research Method . . . 4
- 1.4 Preliminary Publications . . . 5

2 Research Roadmap 7
- 2.1 Thesis Structure . . . 7
- 2.2 S^2EC^2O Introduction . . . 10
  - 2.2.1 Assumptions . . . 11
  - 2.2.2 S^2EC^2O Components . . . 11
  - 2.2.3 S^2EC^2O Process . . . 13

3 Background 19
- 3.1 Long-Living Systems . . . 19
- 3.2 Model-based Security Engineering . . . 19
- 3.3 Vulnerability Databases . . . 20
- 3.4 Ontologies . . . 20
- 3.5 Model Queries . . . 20
- 3.6 Graph Transformations . . . 21
- 3.7 Self-Adaptive Systems . . . 22
- 3.8 PhD Context: Research Project SecVolution . . . 22

4 Context Knowledge in Model-Based Security Engineering 27
- 4.1 Security Context Knowledge . . . 28
- 4.2 Modeling Security Context Knowledge . . . 30
  - 4.2.1 Defining the Security Upper Ontology . . . 30
  - 4.2.2 Ontology Layering as Modularized Knowledge Base . . . 33
- 4.3 Building up the Security Context Knowledge . . . 35
  - 4.3.1 Laws and Regulations . . . 35
  - 4.3.2 Standards and Guidelines . . . 36
  - 4.3.3 Attack Scenarios . . . 36
  - 4.3.4 Vulnerability Databases . . . 36
  - 4.3.5 Incorporate Security Knowledge into the Security Context Knowledge . . . 37
  - 4.3.6 System Level Knowledge . . . 38

4.4 Managing the Knowledge Base . . . 40
4.4.1 Ontology Queries . . . 40
4.4.2 Ontology Updating . . . 41
4.5 Related Work . . . 42
4.5.1 Compliance Checking of Ontologies . . . 42
4.5.2 Knowledge Elicitation . . . 42
4.5.3 Security Requirements Elicitation . . . 43

5 Leverage Changes in the System Context for Secure System Design 45
5.1 Detecting and Assessing Knowledge Changes . . . 46
5.2 Semantic Differencing . . . 47
5.2.1 Security Context Knowledge (SCK) Evolution Example . . . 48
5.2.2 Semantic vs. Atomic Changes . . . 48
5.2.3 Using SiLift to detect SCK Evolutions . . . 50
5.2.4 Henshin Rule Layout Conventions . . . 51
5.2.5 Complex Edit Rule Example . . . 53
5.3 Ontology Reasoning . . . 53
5.3.1 The Challenge of Closed-World Assumption vs. Open-World Assumption . . . 54
Narrowing Degrees of Freedom to Increase Expressivity . . . 55
5.3.2 Choosing a Reasoner . . . 56
5.3.3 Inconsistency with Explanation . . . 57
5.3.4 Inference of Ontology Elements . . . 59
5.4 Related Work . . . 61

6 Co-Evolve Design-Time Models by Assessing Context Evolution 63
6.1 Leveraging Context Evolution for System Co-Evolution . . . 64
6.2 Initial Compliance to Security Properties . . . 65
6.2.1 Security Context Catalog Meta Model . . . 65
6.2.2 Example of the S^2EC^2O Initialization Process . . . 68
6.2.3 Check System's Security prior to Context Evolution . . . 69
6.3 Coordinate Context Evolutions . . . 70
6.4 Co-Evolution at Design Time . . . 71
6.5 Semi-automatic Co-Evolution of Models . . . 73
6.6 Related Work . . . 74
6.6.1 Analyze the Impact of Changes with respect to Co-Evolution . . . 74
6.6.2 Vulnerability and Attack Management . . . 75

7 Assess Security Compliance During Run Time 77
7.1 Run-Time Monitoring with Run-Time Insights . . . 78
7.2 Specifying Security Properties . . . 80
7.2.1 Specification of Security Requirements at Model level . . . 80
7.2.2 Security Requirements at Source-Code Level . . . 82
7.2.3 Mapping of Model Level and Source Level Annotations . . . 82
7.2.4 Synchronizing Model and Code . . . 83
7.3 Round-Trip Engineering Approach for Security Monitoring . . . 84
7.3.1 Verification at Run Time . . . 85
7.3.2 Countermeasures . . . 89
7.4 Support Security Fixing with Run-Time Insights . . . 90
7.4.1 Run-time Protocol for Subsequent Analysis . . . 91
7.4.2 Addition of Missing Elements . . . 92

7.4.3 Documentation of Security Violations . . . 93
7.5 Related Work . . . 94
7.5.1 Taking System Context into Account at Run Time . . . 94
7.5.2 Undiscovered Program Activities . . . 95
7.5.3 Security Monitoring . . . 96

8 Co-Evolve Run-Time Components of Systems 99
8.1 Run-Time Adaptation Approach . . . 100
8.1.1 Motivating Examples . . . 100
8.1.2 Adaptation Support at Design Time . . . 102
8.1.3 Adaptation at Run Time . . . 103
Startup: Pull all Essential Security Requirement (ESR) States . . . 103
Push Newly Threatened ESRs . . . 104
Reset ESR State after Threat is repealed . . . 104
8.1.4 Example of Adaptation Interaction . . . 105
8.1.5 Traceability of Adaptations . . . 106
8.2 Related Work . . . 106
8.2.1 Application Behavior Adaptation . . . 106
8.2.2 Development of Adaptive Systems . . . 107
8.2.3 Security-Aware Systems at Run Time . . . 108

9 Prototypical Implementation 109
9.1 Architecture Overview . . . 109
9.1.1 System Model Design . . . 113
9.1.2 Management of Security Context Knowledge . . . 113
9.1.3 Employing Reasoners via API . . . 113
9.1.4 Realizing specific Implementations for Ecore Objects . . . 114
9.2 Relation of S^2EC^2O's Initialization Process and Realized Prototype Artifacts . . . 115
9.2.1 Initialization Process: Make System S^2EC^2O aware . . . 115
9.2.2 Initialization Process: Initial Compliance . . . 119
9.2.3 Initialization Process: Run-Time Monitoring . . . 119
9.3 Relation of S^2EC^2O's Delta Process and Realized Prototype Artifacts . 121
9.3.1 Delta Process: Determine Context Evolution from SCK . . . 121
9.3.2 Delta Process: Apply Co-Evolution Steps . . . 122
9.3.3 Delta Process: Generating Run-time Findings . . . 124
Detecting System Evolution Automatically . . . 126
9.3.4 Support for Run-Time Adaptation . . . 126
9.3.5 Review of the Prototype's Unique Features . . . 127

10 Case Study: Applying S^2EC^2O to iTrust 129
10.1 Introduction to iTrust . . . 129
10.1.1 Architecture of iTrust . . . 130
10.1.2 Metrics of iTrust . . . 132
10.2 Introduction to Security-Related Context evolutions . . . 132
10.3 Classification and Relevance of Vulnerabilities . . . 134
10.4 S^2EC^2O Application Examples . . . 135
10.5 Example 1: Access Control . . . 136
10.5.1 Initial Compliance . . . 136
10.5.2 Context Evolution and Vulnerability . . . 139
10.5.3 Security Maintenance and Co-Evolution . . . 140

10.6 Example 2: Privacy by Encryption . . . 143
10.6.1 Initial Compliance . . . 143
10.6.2 Context Evolution and Vulnerability . . . 146
10.6.3 Security Maintenance and Co-Evolution . . . 146
10.7 Example 3: Data Protection by Locking . . . 149
10.7.1 Initial Compliance . . . 149
10.7.2 Context Evolution and Vulnerability . . . 150
10.7.3 Security Maintenance and Co-Evolution . . . 151
10.8 Example 4: Communication using Insecure Encryption . . . 153
10.8.1 Initial Compliance . . . 154
10.8.2 Context Evolution and Vulnerability . . . 156
10.8.3 Security Maintenance and Co-Evolution . . . 157
10.9 Example 5: Secure Dependencies . . . 161
10.9.1 Initial Compliance . . . 161
10.9.2 Context Evolution and Vulnerability . . . 162
10.9.3 Security Maintenance and Co-Evolution . . . 163
10.10 Performance Observations . . . 163

11 Contribution to Research **165**
11.1 Review of Research Question 1 . . . 165
11.2 Review of Research Questions 2 and 3 . . . 167
11.3 Review of Research Question 4 . . . 168
11.4 Review of Research Question 5 . . . 168

12 Conclusion **169**
12.1 Contributions . . . 170
12.2 Assumptions and Limitations . . . 173
12.2.1 Generalizability Considerations . . . 173
12.2.2 Scalability Considerations . . . 173
12.2.3 Limitations of the Case-Study Results . . . 173
12.2.4 Support for Additional Security Properties . . . 174
12.2.5 Support for Further Programming Languages . . . 175
12.2.6 Support for Further Software Engineering Methods . . . 175
12.3 Future Work . . . 175
12.3.1 Regard Evolution of ESRs and SMRs . . . 175
12.3.2 Share security and co-evolution knowledge of S^2EC^2O publicly among projects . . . 176
12.3.3 Consider Implementation-Specific Security Vulnerabilities . . . 176
12.3.4 Investigate Influence of Security Co-Evolutions on Functional Requirements . . . 177

A Preliminary Publications **179**

Bibliography **187**

List of Figures

2.1 This thesis' research questions related to typical artifacts in secure software engineering . . . 9
2.2 Relationship of typical artifacts in secure software engineering to evolution and co-evolution . . . 10
2.3 Overview of S^2EC^2Os structure . . . 12
2.4 The initialization process of S^2EC^2O . . . 14
2.5 The delta handling process of S^2EC^2O . . . 16

3.1 Henshin model query to search a state in a UML state chart by its name 21
3.2 Overview of the *SecVolution* approach . . . 23
3.3 Overview of the *SecVolution@Run-time* envisioned approach . . . 24

4.1 Artifacts and activities typically used in model-based software engineering . . . 27
4.2 Example of an ontology to model a secure communication . . . 29
4.3 Ontology of security concepts and their relationships . . . 32
4.4 Example of Security Context Knowledge to provide an encryption algorithm . . . 35
4.5 Overview of active learning using pool-based sampling . . . 38
4.6 UML profile to annotate UMLsec models for the SCK . . . 39
4.7 Example of UML annotations to support bridging to the SCK . . . 39

5.1 Relationship of evolution and co-evolution with regard to model-based security engineering . . . 46
5.2 Overview of different approaches accessing the Security Context Knowledge in S^2EC^2O . . . 47
5.3 Example of evolving Security Context Knowledge: An encryption algorithm is discovered to be vulnerable. . . . 48
5.4 Comparison of atomic change log and pattern-based change log: Atomic change operations can be presented by semantic change operations. . . . 49
5.5 The SiLift process . . . 50
5.6 Example of complex edit rule describing addition of a *Threat* to an existing *Encryption* of the Security Context Knowledge . . . 52
5.7 Example of an ontology modeling access restriction . . . 58
5.8 Screenshot of Protégé explaining the ontology inconsistency . . . 58
5.9 Example of ontology used to infer knowledge . . . 59
5.10 Protégé screenshots showing inferred knowledge . . . 60

6.1 Concepts used in the co-evolution approach and their relation . . . 64
6.2 Meta model of the Security Context Catalog . . . 66
6.3 Example of an Essential Security Requirement . . . 69
6.4 Meta model of delta information in S^2EC^2O . . . 70

7.1 Approach to realize run-time monitoring of security properties specified at the model level . . . 78
7.2 Example of UMLsec secure dependency application . . . 81
7.3 Structure of the run-time monitoring approach regarding software development abstraction layers . . . 84
7.4 Events monitored at run time . . . 86
7.5 Ecore meta model for run-time protocols in S^{2}EC^{2}O . . . 91
7.6 Deployment and manifestation of classes with evolution . . . 93
7.7 sequence diagram generated by S^{2}EC^{2}O run time . . . 94

8.1 Overview of the S^{2}EC^{2}O run-time adaptation approach . . . 100
8.2 UML profile to support *safe mode* adaptations . . . 103
8.3 Sequence diagram showing adaptation interaction of the example . . . 105

9.1 Overview of the S^{2}EC^{2}O tool's architecture . . . 111
9.2 Excerpt from Figure 6.2 to illustrate relation between `SMR` and `SMRExecutable` classes . . . 114
9.3 S^{2}EC^{2}O initialization process: involved tasks 1-5 . . . 116
9.4 Components of the S^{2}EC^{2}O tool relevant for initialization of a S^{2}EC^{2}O aware system . . . 117
9.5 Begin of initialization wizard: Choice of ESRs . . . 117
9.6 S^{2}EC^{2}O initialization wizard: choose details from SCK . . . 118
9.7 S^{2}EC^{2}O initialization process: involved tasks 6 and 9 . . . 119
9.8 S^{2}EC^{2}O initialization process: involved tasks 7-11 excluding 9 . . . 119
9.9 S^{2}EC^{2}O initialization wizard: provide information of the system to instantiate run-time monitor . . . 120
9.10 Components of the S^{2}EC^{2}O tool relevant for delta handling process . . . 121
9.11 S^{2}EC^{2}O delta handling: involved tasks 1, 2, 6, and 7 . . . 122
9.12 S^{2}EC^{2}O Delta Wizard: choice of alternatives . . . 123
9.13 S^{2}EC^{2}O delta handling process: involved tasks 8 and 9 . . . 123
9.14 S^{2}EC^{2}O delta handling process: involved tasks 3, 4, and 5 . . . 124
9.15 Components of the S^{2}EC^{2}O tool relevant for run-time monitoring . . . 125

10.1 Package tree of iTrust Java classes . . . 131
10.2 Security Context Catalog entry for Access Control . . . 136
10.3 SCK excerpt to model access control . . . 137
10.4 Excerpt of iTrust class `PatientDAO` . . . 138
10.5 Excerpt of iTrust class `PatientBean` . . . 138
10.6 SCK excerpt: relations of `getReligion` to other SCK elements . . . 139
10.7 Reasoner explanation for SCK inconsistency . . . 140
10.8 Henshin rule to search «SCK» annotated operations . . . 141
10.9 Henshin rule to alter «SCK» annotations . . . 141
10.10 `The class PatientBean` in its evolved state . . . 142
10.11 Security Context Catalog entry for Privacy by Encryption . . . 142
10.12 SCK excerpt to model privacy requirements . . . 144
10.13 iTrust class `ViewExerciseEntryAction` to provide patient exercise data 145
10.14 SCK excerpt: result of transformation from the system model to the SCK . . . 145
10.15 Inferred knowledge (yellow background) after evolution of SCK . . . 147
10.16 Henshin rule to search classes annotated with «encryptedPersistence» . . . 148
10.17 Henshin rule for adding «encryptedPersistence» to a specific class . . . 148

10.18 Co-evolved class `ViewExerciseEntryAction` with added annotation . 148
10.19 Security Context Catalog entry for Data Protection 149
10.20 SCK excerpt showing a lockable data entity 150
10.21 SCK excerpt showing a new lockable data entity according to the law change 150
10.22 SiLift rule to detect addition of `Lockable` requirement in the SCK . . . 151
10.23 Henshin model query to search for «lockable» annotated classes 152
10.24 Henshin rule to add «lockable» to a given class 153
10.25 Co-evolved class `PrescriptionsDAO` with added «lockable» 153
10.26 Security Context Catalog entry for secure communication 154
10.27 SCK excerpt describing available encryption algorithms 155
10.28 Deployment diagram for iTrust with «secure links» 156
10.29 SCK evolution incorporating threatened RC4 algorithm 157
10.30 SiLift complex edit rule to detect addition of a new threat targeting an encryption algorithm 158
10.31 Henshin rule to query all «encrypted enc» annotated communication paths 159
10.32 Henshin rule to alter «encrypted enc» annotations 159
10.33 Evolved deployment diagram of iTrust with «secure links» 160
10.34 Security Context Catalog entry for secure dependencies 161
10.35 Classes `EditPrescriptionsAction` and `PrescriptionsDAO` annotated according to «secure dependency» 162

List of Abbreviations

API Application Programming Interface

BPMN Business Process Model and Notation

CAPEC Common Attack Pattern Enumeration and Classification

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

CWA closed world assumption

CWE Common Weakness Enumeration

DAO Database Access Object

DL Description Logic

EMF Eclipse Modeling Framework

ESR Essential Security Requirement

GDPR General Data Protection Regulation

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

IRI Internationalized Resource Identifier

JavaEE Java Enterprise Edition

JSON JavaScript Object Notation

JSP JavaServer Pages

JVM Java Virtual Machine

LCW local closed world

MOF Meta Object Facility

NLP Natural Language Processing

NVD National Vulnerability Database

OCL Object Constraint Language

OWA open world assumption

OWASP Open Web Application Security Project

OWL Web Ontology Language

PDE Plug-in Development Environment

RDF Resource Description Framework

RFC Request for Comments

RBAC Role-based access control

SCK Security Context Knowledge

SLR Systematic Literature Review

SMM Security Maintenance Model

SMR Security Maintenance Rule

SQL Structured Query Language

TGG Triple Graph Grammar

TLS Transport Layer Security

UID unique ID

UML Unified Modeling Language

URI Uniform Resource Identifier

UNA unique name assumption

XMI XML Metadata Interchange

Chapter 1

Introduction

1.1 Motivation

Software and especially software-defined systems play an increasingly important role in our daily lives. This is not least made possible by Internet technologies, which build the core of an ongoing pervasion of networks. For example, even conventional communication networks are replaced by solutions built upon software-driven and Internet technologies: Deutsche Telekom, the largest carrier of Europe, has announced to switch off their current phone network based on ISDN technologies until 2018 and is currently migrating to SIP- and IP-based solutions. At the time of writing, the process is still ongoing. Moreover, mobile phone networks are at present also migrated to IP-based technologies [GSM].

Telephony is a part of critical infrastructure, as it ensures reachability of police, paramedics and firefighters. The phone network is also used for a number of measuring and control tasks. Besides of fire-alarm systems and burglar alarm, it is vital for industry plants as well as pumping stations in areas where the ground-water level needs to be regulated.

Contemporary innovations like *smart cities* and *Internet of Things* [AIM10] also reinforce the pervasion of our daily lives with software systems. IoT devices can be considered embedded systems, in other words computers embedded into a surrounding system. The user only interacts with the surrounding system and not with the computer directly [Mar03].

Thus, humans who have little or even no direct interaction with these systems rely on their security. Especially regarding systems providing (critical) infrastructure, the persons affected have no choice if they want to use a specific system nor are they able to patch affected software components. Many of these systems can have significant harmful impact on people's lives, in case their security is compromised.

Systems typically are designed according to requirements. For security-relevant properties, *security requirements* are elicited. Systems used in critical domains obviously need to be compliant with their security requirements. The compliance needs to be preserved over a long period of time, typically more than 10 years [VJL+14]. During such a timespan, it seems even more likely that a system needs to be adapted to an evolving environment. Changing laws and regulations as well as encryption methods discovered to be insecure are examples of changes to the environment. The longer the timespan, the more likely it becomes that such fundamental changes become necessary. Some security issues in particular cannot be foreseen at system design time and are considered *unknown unknowns* [MH05]).

With regard to smart cities and IoT, it can be assumed that critical systems will increasingly be used in an networked manner and thus will be confronted with a continuously changing environment perpetually. Moreover, critical systems may be required to have virtually no downtime. In contrast to that, security vulnerabilities

can be discovered all of a sudden. Regarding the fact that also critical systems are networked and may be even accessible over the Internet, they are vulnerable to new attacks immediately. Thus, a full-fledged development needs too much time and an immediate reaction to the new threat is required.

Knowledge about security vulnerabilities, mitigations, and best practices, is available and, in many cases, publicly. For example, knowledge in terms of expertise of security white hats and formal recommendations like the *Grundschutzkatalog* [Bun] of the Federal Office for Information Security is available. Moreover, platforms managed by the software engineering community exist. For example, the MITRE corporation hosts the popular Common Vulnerabilities and Exposures (CVE) [MITb] database, containing vulnerabilities for concrete software products, as well as the Common Weakness Enumeration (CWE) [MIT17b], being a database for common weaknesses in software development. However, security knowledge is available but distributed among various sources and abstraction levels, ranging from expert knowledge to semi-formal databases. Technical mechanisms for preserving security must be complemented by procedures and cognitive support for human experts who are willing to share their knowledge; they must be empowered to do so at the least effort possible.

The question arises, how to leverage security knowledge to preserve a system's security over its period of application. According to the 2017 Global Information Security Workforce Study, commissioned by the Information System Security Certification Consortium $(ISC)^2$, Europe will face a shortfall of 350 000 cyber-security professionals by the year 2022 [Int17]. For example, even organizations like the European Telecommunications Standards Institute (ETSI), only employ external security experts for a limited time [HIK+10]. Thus, security experts are few in number and it is reasonable to support them to become as efficient as possible.

Another possibility for dealing with security maintenance is to make use of the programmer community as a whole. For example, distributed systems communicating over the Internet make use of a small number of core technologies on lower communication layers, for example SSL. Moreover, communication interfaces or APIs of services like REST and SOAP often rely on HTTP(S). For all these wide-spread technologies, open source implementations exist. The *principle* behind open source software regarding security is, that the public availability of the source code enables the community to publicly audit the code. This should ease the process to find and fix security issues, which in turn leads to secure software. On closer inspection of numerous security issues in the recent past, for example, the OpenSSL bug *Heartbleed*, it is obvious that the mere availability of source code for audits is no guarantee that it actually will take place. Thus, it is no protection against deployment of open source implementations containing severe security issues [DKA+14].

Ultimately, development and maintenance of secure software often is conducted without support of approaches that facilitate compliance to security requirements already at design phase and thus support a correct implementation. This does not only hold for the communications with other systems regarding attacks to interfaces, but also the direct user interaction, when evolving privacy regulations enforce a re-interpretation of roles and access rights [BGR+15], for example. It means in effect that often not the security requirements of a system itself change, but the knowledge about the environment evolves [RGB+14a].

There is demand for supporting security experts in the maintenance of long-living software systems. Manually gathering knowledge is time-consuming and the manual inspection of systems to assess impact of context changes to the system is error-prone due to the complexity of today's systems.

We want to tackle the above-mentioned topics with an approach we define as *research objective* as follows:

> We want do develop an approach that supports maintaining the security of long-living software. Security-relevant knowledge from various sources, in various forms, and at various abstraction levels shall be captured and managed. The knowledge shall be used by the approach to assess, if the security of a given system is compromised, given a change of the security context knowledge. If this is the case, the approach shall support revising the system, so that it fulfills its security requirements again.

In the next section, we will refine the research objective by deriving a number of challenges.

1.2 Challenges

In the previous section, we motivated the topics this thesis deals with and formulated an overall research objective. In this section, we refine this objective. We thus enumerate the following challenges:

C1: Leverage evolution of context knowledge. The context of a software system changes continuously. Examples of knowledge sources that are relevant for the security design of a software system are laws, ordinary requirements and vulnerability databases like CWE, CVE, and the *Open Web Application Security Project* (OWASP) [MIT17b, MITb, Thed]. It is vital having access to as much of this knowledge as possible to raise the probability noticing a change that has negative impact to the security compliance of the software system under consideration. Thus, knowledge is principally available but needs to be handled. While approaches exist to build a secure software system from scratch regarding the current security knowledge, repeating a full security analysis after every change of the security context knowledge that may have no impact on the system at all, wastes resources. With having evolution information at hand, a security analysis can concentrate on the knowledge that changed, avoiding repeating the whole security analysis.

C2: Infer concrete security requirements. A multitude of regulations can be sources for security requirements. Apart from explicit security requirements as part of the ordinary system requirements, there may be regulations coming from the domain the system is to be used in. For example, for systems used in the medical sector, a number of laws need to be followed. For systems processing personal data, the (national) privacy laws need to be followed. Based on the multitude of regulations, vulnerabilities, and mitigations, it needs to be decided which concrete security requirements are necessary to be fulfilled regarding the current security knowledge.

C3: Assess impact of context evolution to a given system. Evolution of the context knowledge can have impact on the system's security. The current system design needs to be inspected with regard to the evolved knowledge. The context evolution information is many-faceted and may be described on different abstraction levels. For example, a rather abstract context evolution information may also be relevant for a rather concrete part of the system, for example a

part of the source code. Thus, information given in various levels of abstraction needs to be bridged.

C4: Co-evolve system design to preserve its security. In case the system is insecure regarding the evolved knowledge, it needs to be co-evolved so that the security requirements are preserved. The functionality of the system needs to be untouched.

C5: Assess system's security during run time. There are security requirements which typically are checked or can only be monitored during run time. This especially is relevant when the behavior of the code depends on data stored in a database. Another reason which makes run-time monitoring necessary is when a full static code analysis cannot be done because the source code of used frameworks is not available. Moreover, run-time monitoring can gain insights regarding suspicious behavior of the system. This information needs to be linked with the security context knowledge at run time to judge, if the system fulfills its requirements regarding the current knowledge.

C6: Adapt the system to preserve security during run time Software is often engineered using model-based techniques, but in many cases, the running code is the result of using subsequent manual steps after automatic transformations. Thus, the executable system often cannot be built automatically from the system model. A pure co-evolution of the model, making a whole implementation cycle necessary, whilst the production system is shut down may not be appropriate. Instead, a mechanism is necessary that is able to adapt the system's security behavior during run time and without need for extensive manual code adaptations.

1.3 Research Method

The presumably most widespread research method in computer science is called constructive design, also known as exploratory research [HMPR04, Crn10]. Application of the method begins with defining a relevant problem. A solution is constructed using theoretical foundations. The solution is then evaluated in dimensions that are appropriate for the problem or the domain the solution shall be applied respectively. Evaluation aspects are, for example, performance of an approach as well as accuracy or completeness of the produced results.

The motivation of this thesis puts emphasis on the relevance of problems occurring with long-living systems. Regarding the aspects of security in long-living systems, S^2EC^2O is related to the research project SecVolution and its successor. We give details in Section 3.8. Moreover, as argued in Section 1.4, S^2EC^2O is related to numerous peer-reviewed publications worked out in context of the SecVolution project.

S^2EC^2O was constructed in an exploratory way. We first considered related work of the relevant domains to determine the state of the art and identify a gap in research as shown by the challenges we presented in the preceding section. After that, we developed an approach to close the research gap and investigated its applicability by means of well-defined scenarios. For this step, we inferred concrete research questions from the challenges (see Section 2.1). We generalized S^2EC^2O and evaluated it in a case study targeting *iTrust*, a system with reasonable complexity. As we argue in Chapter 10, iTrust can be considered a long-living system.

1.4 Preliminary Publications

This thesis builds upon preliminary work, mostly in conjunction with project members of the SecVolution project. This section enumerates them. All publications are joint effort. A detailed discussion of the publications, emphasizing the contributions of this thesis' author, follows in Appendix A. We give an overview of the approach presented in this thesis, S^2EC^2O, in Section 2.2.

- Jens Bürger, Jan Jürjens, and Sven Wenzel: Restoring security of evolving software models using graph transformation.
  In: *STTT*, 17(3): 267–289, 2015. [BJW15]
- Thomas Ruhroth, Stefan Gärtner, Jens Bürger, Jan Jürjens, and Kurt Schneider: Versioning and evolution requirements for model-based system development.
  In: *CVSM 2014*,
  volume 34/2 of *Softwaretechnik-Trends*, pages 20–24, 2014. [RGB+14a]
- Stefan Gärtner, Thomas Ruhroth, Jens Bürger, Kurt Schneider, and Jan Jürjens: Maintaining requirements for long-living software systems by incorporating security knowledge.
  In: *RE 2014*. IEEE, 2014. [GRB+14]
- Jens Bürger, Jan Jürjens, Thomas Ruhroth, Stefan Gärtner, and Kurt Schneider: Model-based security engineering with UML: Managed co-evolution of security knowledge and software models.
  In: *FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *LNCS*, pages 34–53, 2014. [BJR+14]
- Thomas Ruhroth, Stefan Gärtner, Jens Bürger, Jan Jürjens, and Kurt Schneider: Towards adaptation and evolution of domain-specific knowledge for maintaining secure systems.
  In: *PROFES 2014*, volume 8892 of *LNCS*, pages 239–253. Springer, 2014. [RGB+14b]
- Stefan Gärtner, Jens Bürger, Kurt Schneider, and Jan Jürjens: Zielgerichtete Anpassung von Software nach der Evolution von kontextspezifischem Wissen.
  In: *1st Collaborative Workshop on Evolution and Maintenance of Long-Living Systems (EMLS)*, 2014. [GBSJ14]
- Jens Bürger, Stefan Gärtner, Thomas Ruhroth, Johannes Zweihoff, Jan Jürjens, and Kurt Schneider: Restoring security of long-living systems by co-evolution.
  In: *COMPSAC 2015*. IEEE Computer Soc., 2015. [BGR+15]
- Jens Bürger, Daniel Strüber, Stefan Gärtner, Thomas Ruhroth, Jan Jürjens, and Kurt Schneider: A framework for semi-automated co-evolution of security knowledge and system models.
  In: *Journal of Systems and Software*, Elsevier, 2018 [BSG+18].
- Cyntia Montserrat Vargas Martinez, Jens Bürger, Fabien Viertel, Birgit Vogel-Heuser, and Jan Jürjens: System evolution through semi-automatic elicitation of security requirements: A Position Paper.
  In: *3rd IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control* (CESCIT), 2018 [VBV+18].

- Jan Jürjens, Kurt Schneider, Jens Bürger, Fabien Patrick Viertel, Daniel Strüber, Michael Goedicke, Ralf Reussner, Robert Heinrich, Emre Taşpolatoğlu, Marco Konersmann, Alexander Fay, Winfried Lamersdorf, Jan Ladiges, Christopher Haubeck: Maintaining Security in Software Evolution.
  In: *Managed Software Evolution* (*to appear; Springer Open*).
  Editors: Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin.

- Sven Peldszus, Jens Bürger, Jan Jürjens: Reactive Security Monitoring of Java Applications with Round-Trip Engineering.
  Under review [PBJ19].

Chapter 2

Research Roadmap

The approach developed as part of this thesis is called S^2EC^2O (pronounced *Secco*), an acronym for **S**ecure **S**oftware in **E**volving **C**ontexts via **CO**-evolution. The goal of this approach is to contribute to the research questions we will define hereinafter.

2.1 Thesis Structure

Considering the challenges that we discussed in Section 1.2, we determine that an approach is needed that monitors its environment, analyzes possible impact to the system, and carries out necessary adaptation actions. This basically resembles the MAPE-K [ARS15] loop (Monitor, Analyze, Plan, Execute, plus Knowledge), we will also take up again at the end of this section. Moreover, MAPE-K systems make their decisions based on algorithms and/or rules [ARS15]. We thus identify the following research questions regarding a rule-based system. We briefly take up the challenges these research questions relate to:

RQ1: *How can changes in security-relevant context knowledge be used to assess the impact on the system?*

The security-relevant knowledge of the system's context can change in various ways and at various abstraction levels. The question focuses on how these diverse and non system-specific information can be used to assess the impact of a context evolution to the system. This research question tackles the challenges C1 and C3.

RQ2: *How can rules be formalized that are able to preserve the system's security given knowledge evolution?*

Besides security-relevant knowledge that describes vulnerabilities and mitigations, there is also information regarding best practices. Especially when a system needs to be evolved to fix a vulnerability, the question arises what is the current state-of-the-art with no known threats to fulfill the security requirement in question. This research question tackles the challenges C1 and C2.

RQ3: *How can these rules be used to carry out a semi-automatic co-evolution given a context evolution?*

In case a context evolution has been discovered and it has been determined that the system under consideration needs to be fixed, the question arises, how this can be accomplished. A set of co-evolution rules needs to be universal so that it is not tailored against a specific system, but, of course, it needs to be applied to a specific system. Eventually, user support is necessary. It needs to

be decided when to carry out operations automatically and when to rely on the user. This research question directly relates to challenge C4.

RQ4: *How can information coming from the system execution be used, to assess the quality of the security requirements compliance during run time?*

There are security requirements that cannot be checked at design time. Moreover, information coming from the system execution needs to be associated with the current security knowledge to assess, if the current behavior is suspicious. This research question directly relates to challenge C5.

RQ5: *How can information gathered at run time be used to adapt the system when the context evolves, avoiding shutdown or additional design cycle?*

The design documents of a software system do not represent the running code. For example, in model-based software engineering, the system model is the core design artifact. Often, the code is not fully generated from the model but adapted manually before it is deployed. In this case, a co-evolution carried out at the system design does not come into effect until the code itself is altered. Especially regarding systems in critical domains, a shutdown may be unwanted. Instead, an immediate reaction altering the behavior of the running system is required. The adapted running system and co-evolved system design need to be put in sync again later on. This research question directly relates to challenge C6.

Research Q.	Challenges	Chapters
RQ1	C1, C2	Chapter 4 & 5
RQ2	C1, C3	Chapter 6
RQ3	C4	Chapter 6
RQ4	C5	Chapter 7
RQ5	C6	Chapter 8

TABLE 2.1: Relation between research questions and challenges as well as chapters covering them

Table 2.1 lists all research questions and provides an overview, which challenge is addressed by which research question and is handled in which chapter.

We put the research questions in relation to a software engineering approach. Figure 2.1 relates the research questions realizes this using model-based software engineering as reference model, being the focused engineering approach in this thesis.

The gray, filled arrows sketch the typical process flow. On the vertical axis, the typical flow of artifacts is recognizable: The first artifact to be build are requirements. From the requirements, models are created. As a subsequent step, source code is build (either by generation, manual creation or a mixture of both). Ultimately, the system can be executed, which is the run-time phase. Regarding security knowledge, in this simplified view, it contributes to requirements elicitation and finds its way into the system design (i.e. models).

In the following, an overview of S^2EC^2O is given, referring in detail which part of the approach treats which research question.

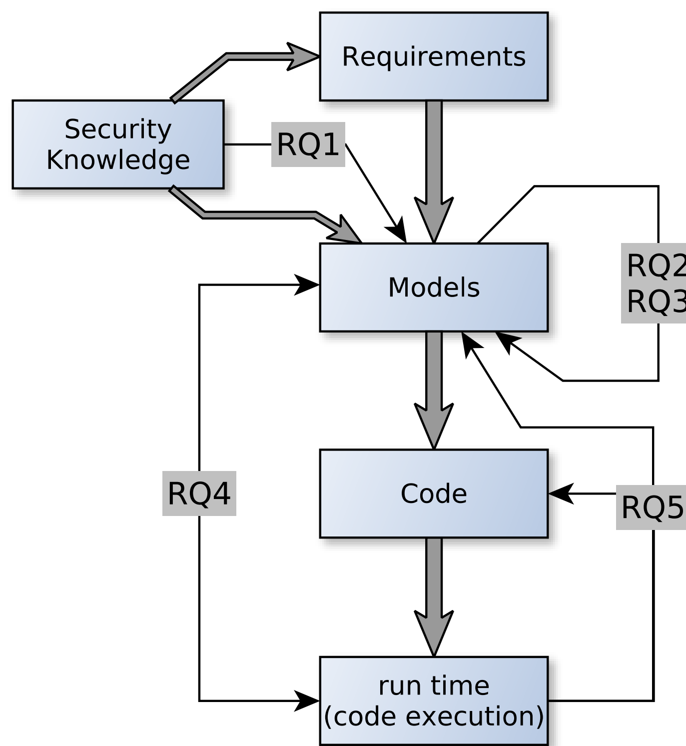


FIGURE 2.1: This thesis' research questions related to typical artifacts in secure software engineering

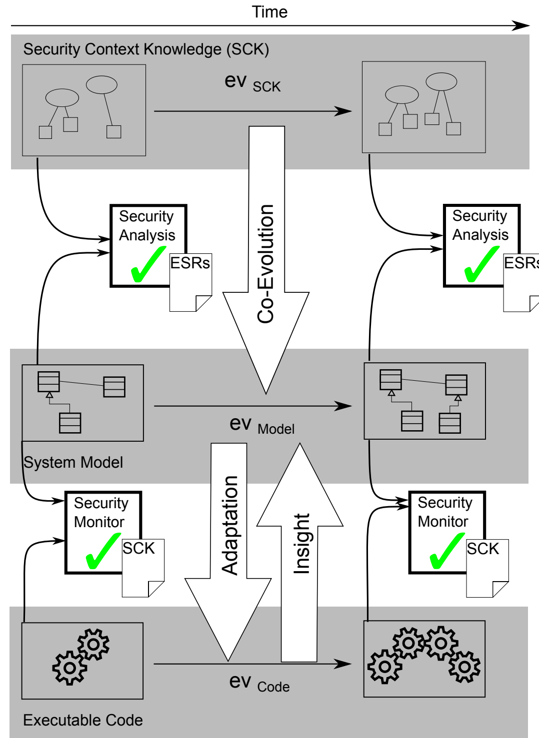


FIGURE 2.2: Relationship of typical artifacts in secure software engineering to evolution and co-evolution

2.2 S²EC²O Introduction

Figure 2.2 shows the relation of evolution and co-evolution regarding the design time and also the system in execution.

S²EC²O's goal is to make a software system aware of its context and react to context evolutions that put the security of a software system at stake. The *Security Context Knowledge* (SCK) guides the development of a secure software system. Using mature *security analysis* techniques, it is possible to assure that a system design meets all security requirements (shown by the green ticks). We call the core of a typical security requirement, detached from technical details, *Essential Security Requirement* (ESR) in this work.

Whenever an evolution of the context occurs (ev_{SCK}), this means that the concrete implementation of security requirements in the given system may not be sufficient regarding the altered context anymore. Thus, a co-evolution of the system design may be necessary (ev_{Model}), so that it passes the security analysis again afterwards.

S²EC²O acknowledges the fact that the system design may be too abstract to generate executable code directly from it. Instead, S²EC²O supports proposing security monitors, by utilizing Security Context Knowledge that couple the executable code and model loosely and in a lightweight manner.

As soon as the system context evolves (ev_{SCK}) or the system design needs to be adapted (ev_{Model}), the running system may still be vulnerable. Many factors can influence the time needed until a security-fixed release can be deployed. To avoid a system shutdown, it may be reasonable to adapt the behavior of the running system as immediate reaction. Moreover, certain security properties cannot be checked statically. For both factors, run-time adaptation is appropriate, thus altering the behavior

of the running code (ev_{Code}). Run-time monitors, using their findings, can provide valuable insights about the current compliance with the security requirements.

In the succeeding sections, we will state the assumptions S^2EC^2O makes and we will present the components S^2EC^2O consists of.

2.2.1 Assumptions

S^2EC^2O is intended to be used by software engineers who are responsible for the management of a software system's security. Thus, we assume background knowledge in the domain of software security and thus call the user of the approach *security expert*. We use the terms *security expert*, *S^2EC^2O user* and *developer* synonymously.

S^2EC^2O's knowledge base is built upon an OWL ontology. Thus, the user should have experience in managing a knowledge base using this semi-formal representation.

As we discussed in Section 2.1, S^2EC^2O targets model-based security engineering. It currently can be used for systems modeled using the *Unified Modeling Language* (UML).

The programming language specific parts of S^2EC^2O currently support Java. The prototypical implementation, the S^2EC^2O tool, is also written in Java.

S^2EC^2O's goal is to assess evolutions of a system's context that has been initially secure, regarding the question if co-evolution operations need to be issued. Respective approaches that are used to design secure systems are often referred to follow the *security by design* principle. How to design a system that is initially secure is not focus of this work. We will refer to preliminary and related work as well as tool support for that to realize.

2.2.2 S^2EC^2O Components

Figure 2.3 depicts the structure of S^2EC^2O. It tackles all phases of a typical model-based software engineering process. In the remainder, we introduce the components, relate them to the research questions and indicate which research question is covered by which chapter.

The light gray rectangle comprises the components of S^2EC^2O, while all remaining components resemble sources for security knowledge (*vulnerability databases*) and the accompanied software system (*UMLsec model*, *System@Run-Time*).

S^2EC^2O incorporates security requirements as given by the developer on one hand, called Essential Security Requirements (ESRs), and external knowledge sources like vulnerability databases on the other hand. ESRs provide terms for basic, abstract, technology-independent security needs. Finally, knowledge about the given software system is also considered. These three sources of knowledge are gathered and managed in the Security Context Knowledge (SCK). The knowledge is represented by an *ontology*. For example, it captures current security-relevant knowledge like which encryption algorithms and key lengths are considered to be secure. The knowledge can be queried using standardized querying languages. Differences in the knowledge can be discovered using differencing approaches, for example.

Chapter 4 elaborates on how to build a knowledge base for security knowledge to capture various aspects, covering common security knowledge, threats and domain- as well as system-specific knowledge to bring these together.

Whenever a change to the knowledge occurs (*Delta SCK*), this triggers an investigation of the system model. The security impact is assessed. The exploitation as well as analysis of knowledge differences relate to **RQ1** and are handled in Chapter 5.

FIGURE 2.3: Overview of S^2EC^2O's structure

Using the Security Context Knowledge, annotations to further support the software engineering process can be inferred. S^2EC^2O focuses on but is not limited to UMLsec, a security extension to the Unified Modeling Language (UML).

To check if the knowledge evolution led to the fact that the model of the system is now insecure and needs to be co-evolved, the delta of SCK is used as a trigger for a set of rules, the *Security Maintenance Rules (SMRs)*. In case a model co-evolution is necessary, it is carried out. This, for instance, can be realized using model and/or graph transformation techniques.

The concept of Security Maintenance Rules and their application relates to **RQ2** and **RQ3**, which are covered in Chapter 6.

S^2EC^2O also covers the operation phase of a software system by taking run time aspects into account. In many cases, executable code is not fully generated out of models, but manual adaptations to the source code are made. S^2EC^2O is designed in a way that acknowledges this fact, so that model and code can be traced using annotations that are related to ESRs.

There are security properties that cannot be fully checked statically, particularly when considering the execution context of a system. For example, code can be loaded dynamically during run time, or the system's behavior can highly depend on a user's input. Thus, S^2EC^2O also proposes run-time monitoring to observe a system during its execution so that this specific run-time data can be taken into account at run time.

Monitoring is realized by a run-time component (*S^2EC^2O run time*). This component monitors the system and reports findings during run time to the S^2EC^2O service component (*S^2EC^2O service*), which then uses the information to eventually trigger design-time or run-time co-evolutions. This is accomplished by providing additional run-time aware Security Maintenance Rules (SMRs).

It seems reasonable to not shut the whole system down as soon as a security breach is detected, but rather *adapt* the system behavior, i.e. disable specific functionalities temporarily, or not taking actions at all but just trigger a warning and detailed logging.

Monitors, as well as an adaptation of system behavior, can be realized using source code annotations as well as manipulating the running code externally using communication interfaces.

Round-trip engineering is supported by feeding run-time insights back into design-time artifacts.

Assessing run-time findings tackles **RQ4** and is described in detail in Chapter 7. How the system is adapted during run time is concerned by **RQ5** and discussed in Chapter 8.

As a consequence, S^2EC^2O instantiates the MAPE-K loop for self-adaptive systems [ARS15]. Monitoring is carried out by the S^2EC^2O run-time component, analysis and planning is handled using Security Maintenance Rules and they also trigger execution of eventual co-evolutions, while the knowledge is managed as part of the Security Context Knowledge and Essential Security Requirements.

2.2.3 S^2EC^2O Process

The S^2EC^2O approach spans a wide range of abstraction levels, starting with probably natural language requirements, affecting model-based development and ending with running code on an actual execution context.

A number of steps require the security expert to take action, while S^2EC^2O generally is semi-automated and tool support for crucial steps is provided.

In the remainder, we introduce the *S^2EC^2O process* to precisely define which action shall take place in what order by which role. The process consists of two phases, both of them are presented using the Business Process Model and Notation (BPMN) syntax. In both phases, the process is driven by two roles: the (human) security expert and S^2EC^2O.

The goal of the *initialization phase* as depicted in Figure 2.4 is intended to make an existing software system accessible for the S^2EC^2O approach. This is possible not only for greenfield development but also for legacy systems (i.e. long-living systems). The phase is described according to the numbering in the figure.

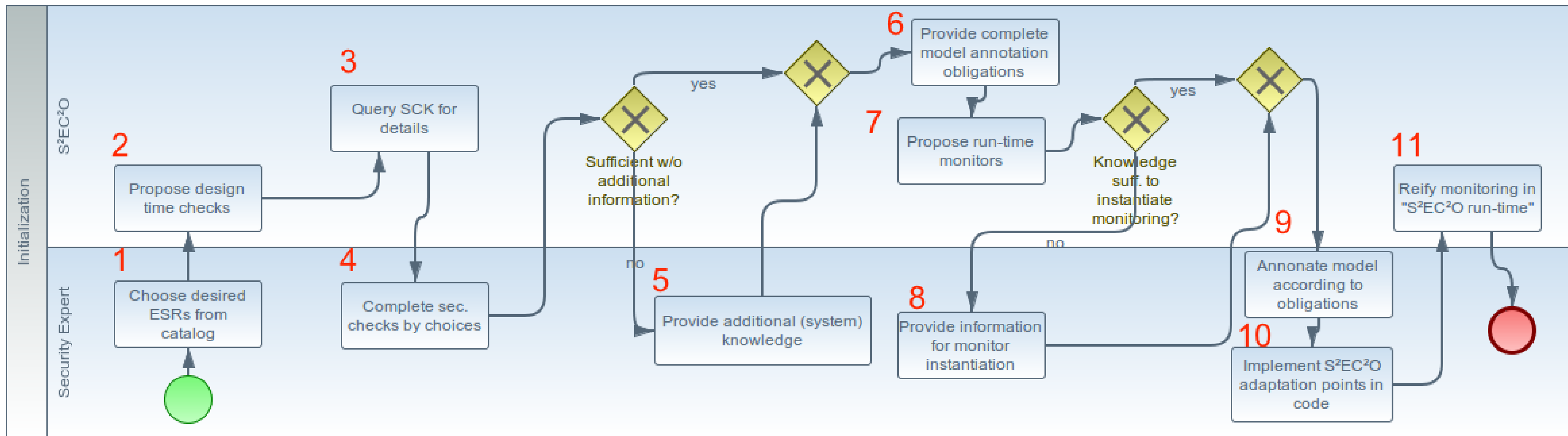


FIGURE 2.4: The initialization process of S^{2}EC^{2}O

Initialization Phase

1. The process is initiated by the security expert who chooses a set of Essential Security Requirements (ESRs) he wants to take care of in development or maintenance of the system.
2. S^2EC^2O proposes a number of security properties to observe, for instance by UMLsec checks or ontology-based checks.
3. S^2EC^2O queries the Security Context Knowledge to collate chosen ESRs with the SCK. This can be realized using standard querying languages. As the SCK is provided as an ontology, this can be, for example, done using the SPARQL query language.
4. The security expert makes choices regarding the queries that have been issued by S^2EC^2O. For example, he has to choose a specific encryption algorithm when more than one algorithm matches the conditions.
5. In case the information provided by the security expert is not sufficient, additional information is requested by S^2EC^2O. For instance, the security expert can be requested to provide system-specific knowledge to populate an access control mechanism.
6. Eventually using additional input provided by the security expert, S^2EC^2O completes the annotation obligations necessary for the design-time checks.
7. S^2EC^2O proposes run-time monitors to the security expert, including instructions which steps need to be followed to make the system adaptable through S^2EC^2O.
8. S^2EC^2O checks if the desired run-time monitors can be instantiated. Eventually, the security expert needs to provide information necessary to instantiate monitoring. For example, consider an application running on an application server, it may be necessary to gain control over the application server to a certain extent. In this case, it is necessary for S^2EC^2O to get required interfaces to these external points configured.
9. The security expert annotates the model according to the issued obligations.
10. The security expert implements adaptation points in the code of the system as necessary so that the system can be monitored and adapted at run time.
11. The monitoring can be instantiated and the software system under consideration is now monitored.

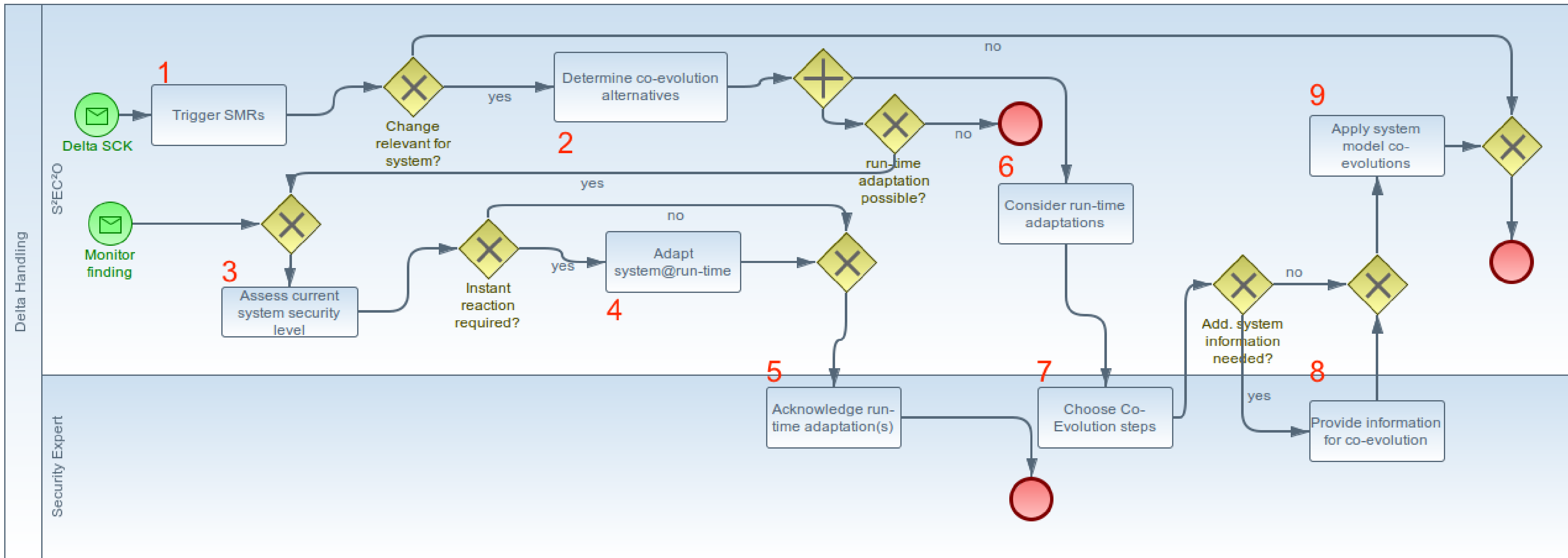


FIGURE 2.5: The delta handling process of S^2EC^2O

Delta Handling Phase

After initialization has been completed, the *delta handling phase* shown in Figure 2.5 becomes relevant. This phase again is described according to the numbering in the figure.

1. One of the entry points to the delta handling is that an evolution of the Security Context Knowledge (SCK) has been detected (*Delta SCK*). This, for instance, can be accomplished using a semantic differencing approach. The delta is investigated by S^2EC^2O to assess a security impact on the system. This is realized by triggering relevant SMRs. If the change is completely irrelevant, nothing is done and the process instance terminates.

2. In case the change is relevant for the system, it is determined which Essential Security Requirements are endangered by the change. If there are no ESRs endangered, (for example, an encryption algorithm is now considered insecure but more secure algorithms are already used throughout the system) the process instance terminates. Security Maintenance Rules then further investigate the system and determine co-evolution alternatives for the system to mitigate the security vulnerability.

3. In case S^2EC^2O has detected a threat to manage, it checks if a run-time adaptation is possible. This also is the same first action taken in case of the second entry point to the process, a reported monitor finding, is triggered. Note that both paths, run-time adaptations as well as assessing the system to determine if instant reactions are necessary are executed in parallel.

   The current system security level is assessed and it is checked if an instant reaction, adaption, for example, is appropriate. An instant reaction may be necessary in case a breach is detected which may have impact on the overall system's functionality.

4. In case an instant reaction is advised by S^2EC^2O, it is realized.

5. In case an instant reaction has been taken place, a report has been generated the security expert has to read to be briefed about altered system behavior and the need for a permanent solution.

6. In case a design-time adaption is worked out with the security expert, run-time adaptations can become available additionally.

7. The security expert chooses the co-evolution steps to be applied to the system.

8. S^2EC^2O checks if additional information is necessary to apply co-evolutions (i.e. provide names for new classes/methods to introduce or acknowledge refactoring proposal). If this is the case, the security expert is asked to provide this information.

9. After collection of co-evolution associated data is completed, S^2EC^2O carries the co-evolutions out and generates a report.

Chapter 3

Background

3.1 Long-Living Systems

Regarding software engineering, a long-living system is in service a considerably long timespan. While there does not seem to be a precise definition, the typical lifespan of a software intensive system is said to be 10 - 15 years [VJL+14]. According to Lehman [LR03], software evolution is the ongoing *progressive* change of software artifacts in one or more of their attributes over time. *Progressive* in this context means that the change results in improvement of the corresponding software. Each change preserves most properties (functionality and security, for example) of the former system and is justified by a rationale. But changes may also lead to the emergence of new properties. Thus, evolution is caused by a wide variety of environmental changes such as technological changes, new stakeholders' needs, modified requirements and assumptions, changes in laws and rules as well as regulations, corrections of discovered problems, and many others. Maintaining security of information systems by taking into account a continuously changing environment is therefore a challenging task in software engineering. Thus, a long-living system demands continuous maintenance *for more than 10 - 15 years*. An investment upfront to wrap a software system into a context monitoring layer can be compensated by the support S^2EC^2O can accomplish in the long run. This thesis focuses on long-living software systems.

3.2 Model-based Security Engineering

S^2EC^2O focuses on supporting model-based security engineering. Model-based software engineering, for example using the Unified Modeling Language (UML), can be used in a process of gradually refining requirements into executable code. In this work, model-*based* and model-*driven* software engineering are distinguished as follows: model-*driven* engineering means that the code is mostly built using models, whereas in model-*based* engineering, models are considered key assets but manual code adaptation steps are acknowledged. Model-based security engineering is supporting model-based software engineering processes to capture security requirements and design secure software. During the requirements elicitation phase, it is necessary not only to consider regular system requirements but also security-related requirements, to ensure secrecy of certain data, for example. A secure design is difficult to get right and many designs contain security weaknesses which are often nontrivial to find and then to fix. Especially when considering evolving contexts, it is necessary to constantly validate the security requirements of design models. To annotate UML design models with security requirements in S^2EC^2O, the UMLsec approach is used [JJ05]. Recurring security requirements and security assumptions on

the system environment can be specified as part of UML models. Thus, knowledge on careful security engineering is encapsulated as annotations in models or code and made available to developers who may not be security experts. The UMLsec extension is given as a UML profile using the standard UML extension mechanisms. Stereotypes are used together with tags to formulate security requirements and assumptions. In [JJ05], the core approach is presented. UMLsec can and has already been extended by a number of additional security requirement annotations.

Tool support for UMLsec has been introduced together with the original approach. Moreover, an extensible version built upon the Eclipse Modeling Framework (EMF), called CARiSMA, also exists [APRJ17]. It also has been used and made further process as part of the EU project *VisiOn* (Visual Privacy Management in User Centric Open Environments; 2015-2017, H2020-DS-2014-1) [EU].

3.3 Vulnerability Databases

Community knowledge regarding security best practices, known vulnerabilities, and mitigations is available. One example is the Common Weakness Enumeration (CWE) database. A CWE entry [MITb] contains the following aspects, among others: (1) a description of the security issue, (2) consequences, (3) applicability to specific programming languages and development process phases, and (4) possible mitigations. In our setting, the system becomes subject to a certain CWE entry after the environment has changed: either a change of the domain knowledge (an encryption algorithm becomes insecure, for example) or a law change (introduction and handling of the additional notion for personal data).

3.4 Ontologies

This work makes use of a knowledge representation that needs to be storable, updateable and flexible enough to support different levels of abstraction and uncertainty. Specifically, security issues cannot be foreseen at system design time and are considered *unknown unknowns* [MH05]. Thus, a suitable knowledge representation which can be adapted to entirely new fields of knowledge is required. To this end, we use the knowledge representation concept of *ontologies* [Gru93]. Ontologies exist in many different forms and flavors. In this work, we build upon the understanding that an ontology contains the key concepts of a domain and the relationships between them. Our technical realization of ontologies is based on the Web Ontology Language (OWL), standardized by the W3C [OWL09]. Thus, for discussing ontologies in this work, we especially use the terminology as is used for the OWL.

3.5 Model Queries

To retrieve specific information from a model, model queries can be used. Since system models can be interpreted as graphs [BJW15], various techniques based on graph algorithms can be incorporated to investigate properties of a given model. Model queries are a widely used concept (see [HKSS08, UBAH+15]). A model query is carried out by firstly providing a query string based on a query language and a model as input to an evaluation algorithm. Execution of this algorithm determines a set of model elements matching the query, which also can be empty.

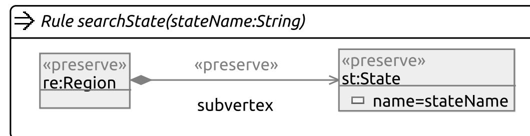


FIGURE 3.1: Henshin model query to search a state in a UML state chart by its name

To check if a model violates certain properties, one can proceed as follows: the non-compliance to this properties is modeled as query. If the query result is empty, it means that the model does not contain non-compliant elements and thus is compliant with the property in question. By now, S^2EC^2O focuses on the following possibilities to query models. First, we can formulate queries by a graph transformation and its underlying matching algorithm [BJW15]. Second, we can make use of our tool platform for risk and compliance checks, CARiSMA [APRJ17] which supports analyzing models, for example in UML using approaches such as the Object Constraint Language (OCL) to formalize the security properties under investigation.

3.6 Graph Transformations

Graph transformation is a well established paradigm to describe change operations of graphs in a formal way, mainly used for describing changes, synchronizing two graphs, for example coming from different meta models or generating code out of a software system model. Graph transformations typically consist of rules that are specified by a left-hand side (LHS) and a right-hand side (RHS). Typically, wherever the LHS can be matched as part of a model, the rule is applied to transform the LHS into the RHS. Graph transformations can also be used to conduct model queries by designing graph transformation rules where the LHS equals the RHS [BJW15]. To model and execute graph transformations, we make use of the graph transformation framework Henshin [ABJ+10, SBG+17]. Henshin has a unified view on the LHS and RHS of a transformation rule. In a nutshell, mappings between LHS and RHS are represented as so-called *actions*. These actions are annotated as stereotypes in a graphical representation. We shortly introduce the fundamental action types. ⟪preserve⟫ means that an element is member of the LHS as well as the RHS. ⟪create⟫ means that an element is only member of the RHS. ⟪delete⟫ means that an element is only member of the LHS. Henshin features a powerful API that provides access to resulting model elements of a match and thus gives a flexible way to interpose Java code between rule executions. For example, *partial match* allows us to use transformation rules with fewer nodes. Here, usage of partial match as well as the Henshin API allows us to design more abstract transformation rules that are parameterized using program code with information gathered from model queries and the knowledge delta. We make use of Henshin in two ways.

First, we use Henshin to carry out model queries. This is accomplished by modeling graph transformation rules solely using ⟪preserve⟫ type nodes. This way, we achieve that there is no real modification of the underlying model carried out, but the matcher as part of Henshin is used as a pattern matcher for the given model. Figure 3.1 presents a model query which is used to search a state in a state chart by a given name. Thus, every match of such a *query* rule means that the respective pattern has been found in the model.

Second, we also use Henshin to carry out co-evolutions. Additional details on Henshin and how we use it to find model flaws is in our previous work [BGR+15, BJW15].

3.7 Self-Adaptive Systems

Referring to [OGT+99, MEHDH13], a self-adaptive system monitors itself and is able to adapt its behavior accordingly. Regarding levels of automation, realizing adaptations using human-in-the-loop in a semi-automatic manner is also conceivable. S^2EC^2O makes use of two core concepts commonly known among self-adaptive systems. First, S^2EC^2O makes use of run-time monitoring to check during run-time if certain requirements (i.e. security requirements) requested during design time, hold during run time. For example, monitors can be generated according to the security requirements as annotated at the system specification. Monitoring can supervise system variables which provide information about the internal state. Additionally, monitoring using call traces helps to get information about the system state.

Second, S^2EC^2O makes also use of run-time adaptation. To alter the system behavior at run time, annotations are used. Similar to run-time monitoring, variables as well as central method calls in the control can be used as interception points to decide if the original behavior should come into effect or if alternate methods should be called. Regarding self adaptivity, S^2EC^2O is inspired by the core idea of the GRAF [ADET12], which tends to transform existing systems into self-adaptive systems.

3.8 PhD Context: Research Project SecVolution

This work has been carried out as part of the research project *SecVolution – Beyond One-Shot Security: Keeping Information Systems Secure through Environment-Driven Knowledge Evolution*. This project is part of the German Research Foundation (DFG) priority programme 1593 "Design For Future - Managed Software Evolution" [DFG]. SecVolution has been extended to be part of the SPPs second funding phase as *SecVolution@Run-time – Beyond One-Shot Security: Requirements-driven Run-time Security Adaptation to Reduce Code Patching*.

The DFG issued the priority programme SPP1593 with an overall run time of 6 years and an overall funding of approx. 10 million EUR. The priority programme is split into two funding periods. SecVolution is part of both funding periods.

The SecVolution approach is a holistic framework to deal with evolving knowledge in the environment of a software project. The overall goal is to restore security levels of an information system when changes in the environment put security at risk.

It is built upon the *SecReq* approach developed in previous work of the author's working group and project partners [HIK+10, Sch11, JS14]. As a core feature, SecReq supports reusing security engineering experience gained during the development of security-critical software and feeding it back into the model-based development process. To this end, SecReq combines three distinctive techniques to support security requirements elicitation as well as modeling and analysis of the corresponding system model: (1) Common Criteria [Int07] and its underlying security requirements elicitation and refinement process, (2) the HeRA tool [KLM09] with its security-related heuristic rules, and (3) the UMLsec tool set [JJ05] (predecessor of CARiSMA [APRJ17]) for secure system modeling and security analysis. This

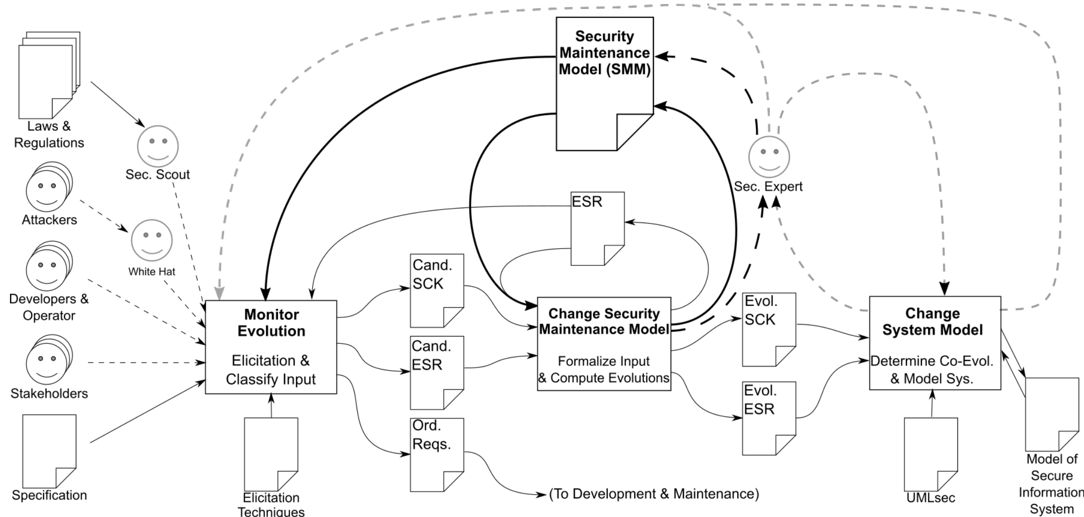


FIGURE 3.2: Overview of the *SecVolution* approach. It uses the FLOW notation [SSK08].

bridges the gap between security best practices and the lack of security experience among developers. However, a significant limitation of SecReq is that it cannot cope with evolution of the required security knowledge and, thus, has to be regarded as a *one-shot* security approach.

SecVolution overcomes this limitation of SecReq by monitoring the system's environment to infer appropriate adaption operations.

Figure 3.2 depicts the information flow developed during the first project phase. It uses the FLOW notation, which has been used to describe explicit and implicit information flows as well as the overall project process in the SecVolution project. Basically, nodes represent persons, documents, and activities; solid arrows denote solid (i.e., long-term accessible and repeatable) information flow, dashed arrows indicate fluid (non-solid) information flow. A non-solid flow of information is characterized by information that gets into the indirectly, passively, or not accountable. For example, a security white hat is not part of the software engineering team, but the information he provides is used in the process. Bold arrows indicate experience, an information flow type that often acts as catalyst on other information flows. We refer to the full publication for more details on FLOW [SSK08].

SecVolution's reaction to evolution is a systematic co-evolution of corresponding software models with respect to environmental changes as triggering events. An assumption of the project is that the system fulfills all security requirements regarding the current knowledge during design time. The system specification is assumed to be a UMLsec model. There exist a number of security requirements which can be modeled using UMLsec. Their compliance can be checked using checking plug-in implementations for the UMLsec tool suite and its successor CARiSMA [JJ05, APRJ17].

SecVolution monitors and prepares the change to the system by providing a knowledge base (*Security Maintenance Model*). The knowledge base contains information on how to react adequately to changes to the environment (*Change Security Maintenance Model*). This happens always with security in mind. The knowledge supporting long-living software evolution is elicited from different sources. This includes not only natural-language documents like *laws* and *regulations* as well as semi-formal assets like system specification documents, but also people that are

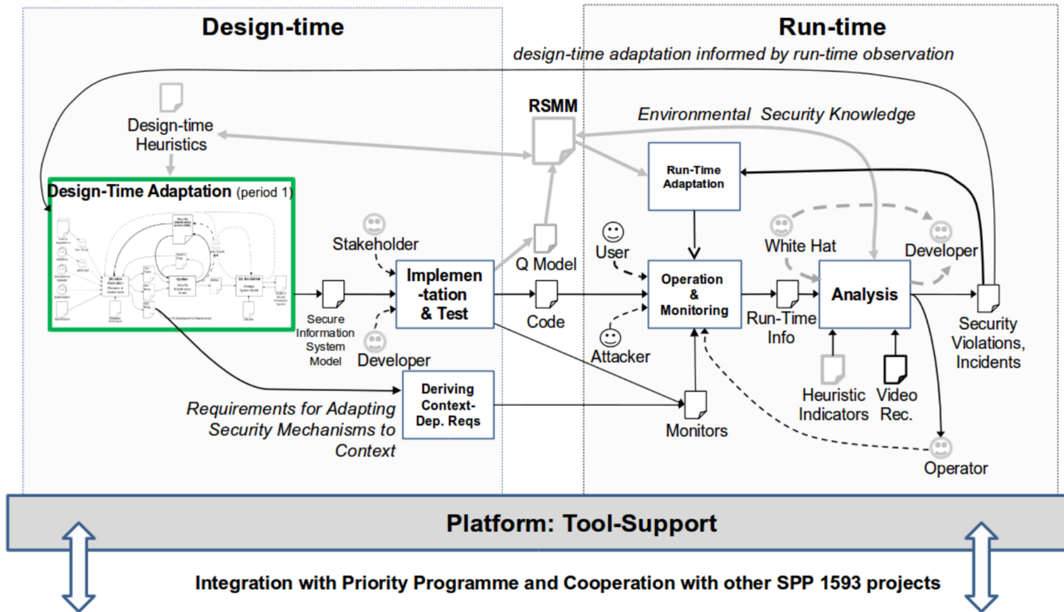


FIGURE 3.3: Overview of the *SecVolution@Run-time* envisioned approach

relevant to the development process like *attackers*, *white hats*, security experts (*sec. scouts*), and *stakeholders*. With this updated knowledge base (*Monitor Evolution*) the system models are evolved by a semi-automatic mechanism to get back the initial fulfilled security of the software system. Monitored changes can trigger reactions which lead to a co-evolution of security precautions and the corresponding system model. To maintain an achieved security level, these models must be adapted to deal with environmental changes affecting security properties of the system. To support the elicitation of the security knowledge, a heuristic mechanism based on natural language processing is available [GRB+14] (*Elicitation Techniques*). As argued in previous work (see [HIK+10, Sch11]), freeing valuable time of security experts is an essential benefit, because they can then deal with new threats that are not covered by automated reaction yet.

The PhD thesis of the SecVolution project member Stefan Gärtner focuses on the elicitation of security and attack knowledge. Knowledge on how to perform attacks is mined from vulnerability databases. From this, attack sequences are generated and used to check the existing requirements specification (especially use case descriptions) if any vulnerabilities exist [Gär16].

The overall security knowledge, capturing the context knowledge as well as actions how to react to evolution, resides in the central element called *Security Maintenance Model (SMM)*. Security knowledge itself is split into *Essential Security Requirements (ESRs)* and *Security Context Knowledge (SCK)*.

ESRs resemble the bare security requirements elicited during the requirements engineering phase, while SCK contains knowledge with technical details and context knowledge about vulnerabilities and mitigations.

After all, the system model is co-evolved (*Change System Model*). The target artifact is the altered system model, now compliant with all security requirements (*Model of Secure Information System*).

The goal of *SecVolution@Run-time*, the second project phase of SecVolution, is to extend the core ideas of SecVolution, which reside in the design-time phases of the

software life-cycle, into the operation phase. The output of applying the SecVolution approach is a model of a secure information system, as Figure 3.3 shows.

First of all, SecVolution@Run-time acknowledges that model-based development often relies on manual code adaptations until it goes into deployment. To bridge the code and system model, the SMM introduced in the first project phase is extended by information relevant to the run time, as trace links, to a *Run-time SMM* (*RSMM*).

The system's security is not assessed on a binary scale (i.e. all security requirements are fulfilled or not) but on a more fluid scale, to acknowledge the fact that systems are more and more complex and a single vulnerability does not mean that a full system shutdown is the only possibility to react to a security breach. This is supported by a quality model (*Q model*) of the system. Findings that are detected during run time are analyzed immediately (*Analysis*). The approach decides in a semi-automatic manner if a permanent run-time adaption is sufficient (*Run-time adaptation*), a temporary adaptation followed by a design-time cycle is necessary or a full shutdown is inevitable . To make this decision, a security expert may be involved.

To get information about the running system, (security) monitors as well as methods from the domain of (self-)adaptive systems are used. The analysis of the system is additionally supported by *heuristic indicators*, as well as screen video recordings (*Video rec.*). Video recordings can be used for demonstration and guided explanation purposes of inner workings. Call traces of demonstrations can be recorded as a byproduct. Thus, intended behavior can be demonstrated, as well as the realization of vulnerability mitigation or demonstration of an attack. Moreover, video snippets can be reused when an issue occurs which is related to a system part or call trace that has been documented this way.

Regarding the first project phase of SecVolution, this thesis focuses on the process of reacting to an evolution of the SCK. It is also considered how necessary co-evolutions can be inferred and applied.

Regarding *SecVolution@Run-time*, research incorporated by this thesis focuses on monitoring code at run time by providing monitoring probes and adaptation points realized by source code annotations. Design time and run time, i.e. model and code level, are bridged by providing methods to let monitor findings contribute to design-time artifacts.

Chapter 4

Context Knowledge in Model-Based Security Engineering

Among the numerous approaches for software development, model-based software engineering is well known and provides a structured way of developing software. According to Höhn et al. [HLJA10], the traditional strategy for security assurance has been *penetrate and patch*. Figure 4.1 shows, what steps are typically undertaken in model-based software engineering. Starting with *requirements*, a system is built. The *models* can be analyzed against the requirements to check if all requirements are met. The models can then be used to generate *configurations* for an execution context, the *code* to be executed later on, and tests. The *code* may be adapted manually. Tests can be used to check if the code still corresponds to the model, as well as models can be *reverse engineer*ed from existing code to realize a difference inspection on the model level. Moreover, systems that have not been built using model-based engineering in the first place can be reversely converted into such a project.

S^{2}EC^{2}O especially focuses on security by design, where security requirements are captured accompanying the ordinary requirements elicitation phase of the regular software engineering process [JJ05]. Typically, a software engineering process begins with high-level stakeholder interaction eliciting requirements. As argued in [Sch06], requirements elicitation should be as non-intrusive as possible, so that using natural language for this task is appropriate. Thus, security requirements can

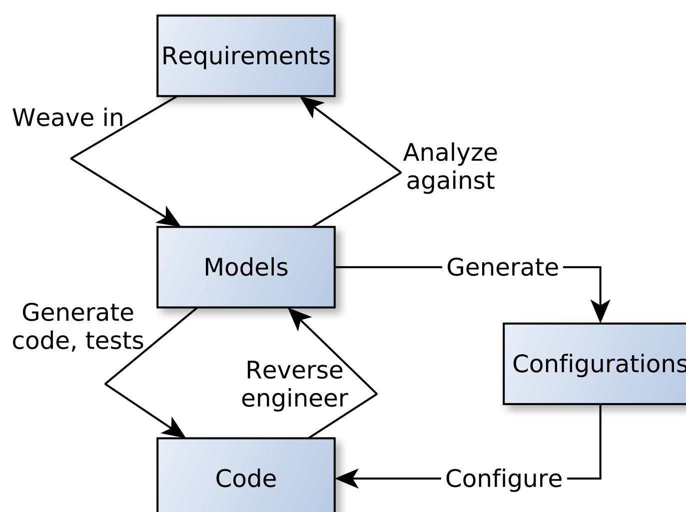


FIGURE 4.1: Artifacts and activities typically used in model-based software engineering (adapted from [HLJA10])

come up during the requirements elicitation phase. Moreover, it is reasonable to capture security-relevant requirements as early in the development process as possible [Gär16].

Nowadays, the majority of software systems are distributed systems one way or the other. The security of a software system is in large measure affected by the context it is used in. There is the need to bring the various dimensions of engineering complex systems and their context under control. While S^2EC^2O's focus is to analyze context evolution, a formalization of the context prior to evolution is needed in the first place for having a well-founded baseline to compare to. In Section 4.1, we introduce the notion of *Security Context Knowledge* to capture context knowledge that is relevant to build secure software systems. Section 4.2 then shows, how to model and build a knowledge base according to this notion. We accomplish this by using an ontology. We present a number of knowledge sources which can serve to populate the ontology with security knowledge. Apart from that, we introduce a modularization in terms of layering the knowledge base, to support separation of concerns in knowledge modeling. For example, system-specific aspects can remain in a separate ontology, keeping other parts of the knowledge model clean and reusable.

We also elaborate on approaches which can be used to incorporate knowledge sources, only available in natural language, into the Security Context Knowledge (SCK).

After all, we show how to bridge the gap between the knowledge base with its layers ranging from coarse-grained global knowledge down to fine-grained system-specific aspects, and model-based secure software design.

Section 4.4 covers the question of how to manage the knowledge of SCK after it is built initially.

4.1 Security Context Knowledge

A software system is accompanied by a number of security requirements, but the question if the requirements are met correctly, may depend on the system's context. On the one hand, security requirements need more or less effort regarding the domain the system under consideration is to be used. Given a messaging system, realizing security requirements may differ on a technical level based on the application domain, for example a messaging system as part of a public web forum compared to a messaging system an insurant can use to communicate with his insured people and doctors.

As another example, we consider a system processing private data and connected to a database. Consider a security requirement like *The database server must use RC4 as secure encryption algorithm.* There are at least two facts that hold for this requirement:

1. It may not be system specific. What is considered a secure encryption algorithm is a general question made out by domain experts or is even part of a more common security knowledge.

2. In every case it is not time invariant. What is considered a secure encryption algorithm at one day, can be known to be insecure all of a sudden when a zero day exploit is disclosed.

Thus, the security of a system under consideration is affected by common, security-related knowledge that is not part of the actual system. Further knowledge to be taken into account concerns existing vulnerabilities and possible mitigations,

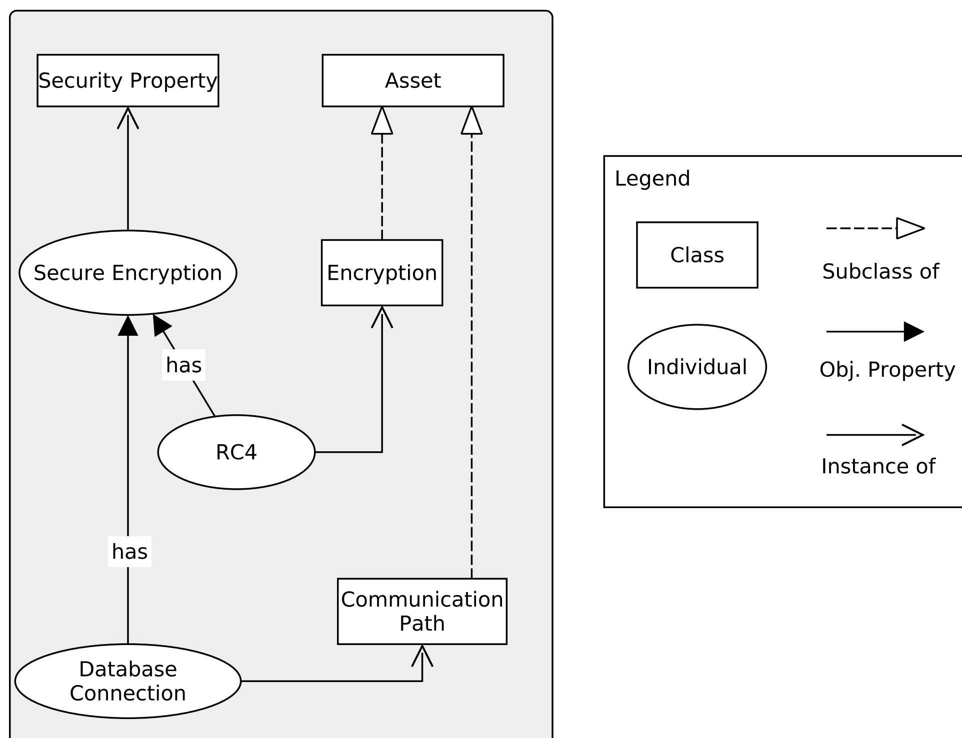


FIGURE 4.2: Example of an ontology to model a secure communication

eventually provided by the community. In S^2EC^2O, we call this kind of knowledge *Security Context Knowledge (SCK)*.

Figure 4.2 shows an excerpt of an ontology with a security requirement (`Secure Encryption`), system components (`Database Connection`, `Communication Path`) and an actual encryption algorithm (`RC4`).

More precisely, what is considered to be an ordinary security requirement (i.e. *The database has to be connected using the secure encryption algorithm RC4.*) is broken down into two aspects. First, *Essential Security Requirements (ESRs)* cover the abstract, technology-independent security needs (i.e. *The database server must use a secure encryption algorithm.*). SCK then provides the security knowledge required to bring ESRs into action for a specific system. ESRs will be discussed in detail in Section 6.2.1.

To sum it up, SCK includes, but is not limited to (especially new) attacker types and their abilities, encryption protocols, and their robustness against various attacks. SCK is usually gathered from natural language documents of various kind, for example the security knowledge as part of the IT baseline protection guidelines proposed by the German Federal Office for Information Security [Bun], or attack and vulnerability reports as provided by the MITRE Corporation in the CVE database [MITb]. Moreover, individual persons such as white hats or developers can also contribute to the SCK. It is supposed to evolve more often than ESRs, as knowledge about security techniques may change quickly and has direct impact on the security of the system.

The following section gives details on how the Security Context Knowledge (SCK) is represented and managed in S^2EC^2O.

4.2 Modeling Security Context Knowledge

As argued in the preceding section, the SCK needs to incorporate knowledge from various sources. The knowledge base needs to be managed and used in S^2EC^2O in a semi-automatic manner (see Section 2.2, [BGR+15, BSG+18]), so that a formal representation of knowledge is inevitable. Since collecting, formalizing, and maintaining required knowledge is a laborious task, a formal representation of knowledge should be shared among various projects and adapted to the specific needs and requirements.

For example, privacy should be fulfilled by an organization according to given security standards and regulations. Before introduction of the GDPR [EU 16], the European Union defined privacy rules [EU 95] which had to be refined by each member state. Germany accomplished this with the *Bundesdatenschutzgesetz* BDSG [Bun05], while the United Kingdom had the *Data Protection Act* DPA98 [Bri]. Additionally, each organization had its own privacy guidelines extending and re-defining the respective country regulation.

To model such knowledge, ontologies are a commonly used technique [HS06, NCLM06]. They represent knowledge in a formal manner by using a set of types, properties, relationships, individuals, and axioms. The knowledge base also needs to be flexible and easily extensible, because security issues cannot be foreseen at system design time and are considered *unknown unknowns* [MH05]. Ontologies, by design, provide high flexibility in knowledge modeling, because they are based on the open-world assumption.

Moreover, knowledge management activities (for example the use of ontologies) need to fit into the developers' work flow since rigid integration typically results in retarding instead of supporting the development as well as the maintenance process [MARS12].

In the succeeding sections, we will use the ontology shown in Figure 4.2 as example and gradually develop it into SCK. We will introduce the *security upper ontology* to build the core of the SCK. In this process, we will introduce an *upper* ontology providing a basic taxonomy for security-relevant notions. We will introduce it in Section 4.2.1. Section 4.2.2 shows how layering ontologies supports modularization and thus foster project-spanning reusability of knowledge. In Section 4.3, we put emphasis on how to populate the upper ontology with actual security knowledge. Section 4.3.6 puts emphasis on how to connect the system model with the SCK.

4.2.1 Defining the Security Upper Ontology

In this section, we introduce the *security upper ontology*: It provides a number of relations and notions which are used to build up *instances* of the knowledge base respectively. The security upper ontology can thus roughly be compared to a meta model in the modeling domain. The ontology introduced here and used for the Security Context Knowledge (SCK) has been introduced firstly in [Gär16, BSG+18]. An earlier, substantially smaller security ontology of security-relevant methods, threats and mitigations is presented in [GRB+14].

The *security upper ontology* as presented in this section is created after carrying out a systematic literature review. We refer to [Gär16] for full details of the systematic literature review, while we only briefly summarize the review process as presented in [BSG+18]. The focus of this thesis is on how S^2EC^2O extends the ontology as presented in [Gär16] by adding (semi-)automatic mechanisms to query and manipulate the knowledge base using OWL 2 [OWL09] and accompanying technologies.

To have the security upper ontology in a well supported format, it was modeled in OWL 2 using Protégé [Sta]. It can thus be accessed or processed using widespread formats like Resource Description Framework (RDF) triples or the OWL/XML format [OWL09].

Systematic Literature Review

To define the security upper ontology, we conducted a Systematic Literature Review (SLR). SLR is an empirical method used to aggregate, summarize, and critically assess all available knowledge on a specific topic [KC07]. In our case, we searched for scientific publications related to the modeling of security-specific knowledge in the context of security management or software and systems modeling. In particular, we addressed publications in which concrete ontologies for the modeling of knowledge related to IT security are described.

To find the relevant publications, we used the literature databases *ACM Digital Library*, *IEEE Xplore*, *ScienceDirect*, and *SpringerLink*, since they cover the largest part of scientific journals, conferences, and workshops in the domains of software engineering and knowledge management [BKB+07]. The search query for a automated search was obtained by combining the search terms *Security* (in title), *Information System, Software* (in title or abstract), and *Ontology, Metamodel* (in title, abstract or text). This search yielded a total population of 287 publications which we subsequently filtered to retain only those that (i) were available in English, (ii) describe an existing, practically applicable approach (rather than a position statement, for example), (iii) address the modeling, application, or acquisition of security knowledge in software engineering, and (iv) were not specific to a particular domain. The remaining 46 publications contained 29 security-related ontologies that we analyzed in detail to aggregate the minimal set of concepts, required to enable the modeling of security knowledge.

Ontology Analysis

In the analysis of the 29 ontologies, the objective was to answer the following research question: *What is the minimum set of terms required for modeling security knowledge to enable a heuristic security analysis of development artifacts?*

None of the analyzed ontologies matched all requirements fully. By focusing on the minimum set of terms, we address a key prerequisite for the design of our ontology, related to *ontological commitment*.

Ontological commitment [Gru95] is an important principle in ontology engineering. To support flexible use of a created ontology, it proposes to impose as few restrictions as possible. A minimal ontological commitment is achieved if the ontology contains only the essential and most general terms in the considered domain. In his PhD thesis, Willem Nico Borst [Bor97] puts emphasis on the purpose of sharing knowledge and comes to the following conclusion:

> "Ontologies are formal specifications of shared conceptualizations. They can be the instruments for knowledge sharing and reuse. Ontologies can support information systems design because they specify the knowledge an information system must capture to perform its tasks. To be (re)usable, ontologies should not capture facts about instances in the domain, but make explicit tacit and meta-level knowledge."

This allows others to use the ontology in their particular applications, specializing it where necessary. Based on the Systematic Literature Review, we achieved a

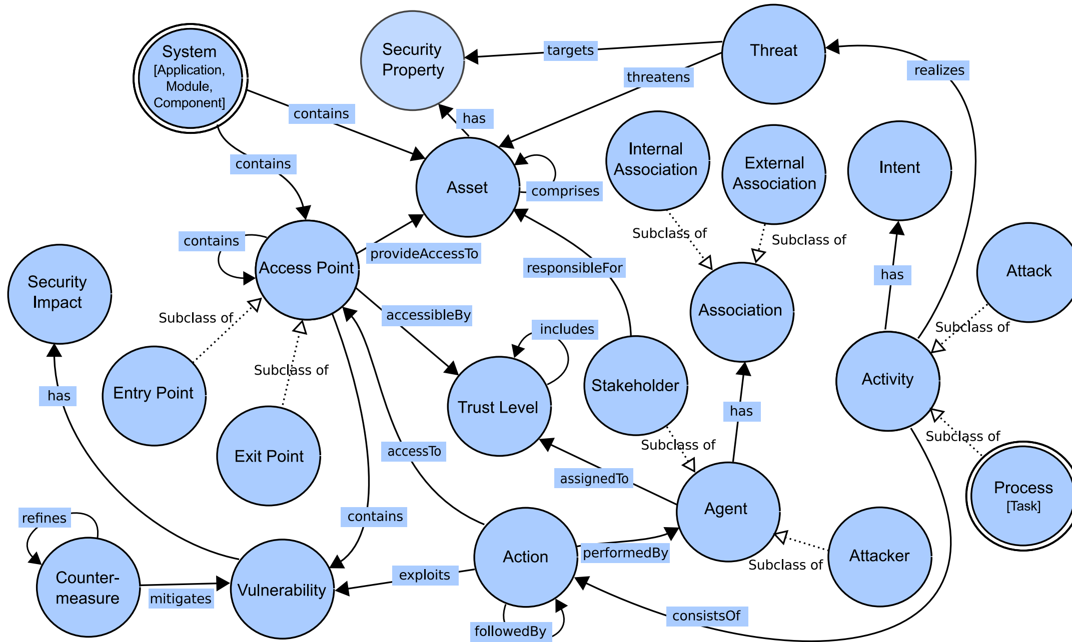


FIGURE 4.3: Ontology of security concepts and their relationships in VOWL 2 syntax

minimal ontological commitment by extracting the essential and most general terms. A term fulfills this criterion if it occurs in multiple ontologies, or if it was pointed out as particularly relevant in the underlying ontology's description.

We developed a classification of all terms included in all ontologies. To this end, the terms were assigned to classes, based on the descriptions in the publications from which they were obtained. Various concepts occur under different terms across publications, mostly due to the different research backgrounds of the authors and different problems they want to solve. For instance, in most considered ontologies, the term *asset* is used to refer to a protection-worthy object; yet, in some cases, the terms *affected element*, *resource*, or *information* were used. Our classification represents all of these terms as one particular class *asset*.

Resulting Security Upper Ontology

From the classification, we obtained the security-relevant classes for our ontology. Figure 4.3 shows the resulting ontology in VOWL 2 [LNHE14] syntax; circles denote classes, and lines between them denote object property associations and subclass relations. In the following, we summarize the classes this thesis focuses on and discuss evidence of their background. For the explanation of the remaining classes, we refer to [Gär16].

The classes *System*, *Access Point*, *Asset*, and *Trust Level* represent system-specific knowledge about the system and its parts. These aspects are addressed by the 10 of the 29 considered ontologies, in which the terminology is subject to some variability. For instance, Elahi et al. [EYZ09] distinguish the system parts *Product*, *Component*, and *Function*. According to Swiderski et al. [SS04], an access point is an integral part of a system, application, module, or component.

The class *Security Property* describes security requirements to an asset of the considered system. 11 out of the 29 considered ontologies specify a corresponding term.

The relationship between a security property and an asset is highlighted in several ontologies, such as the one by Dubios et al. [DHMM10].

The classes *Threat* and *Attack* are dedicated to threat and attack modeling. Corresponding terms are included in 25 out of the 29 considered ontologies; 21 of them also represent vulnerabilities or weaknesses of the considered system explicitly. Usually, a vulnerability is related to a particular attack or threat. Moreover, the proposed ontology assigns an intent to attackers which is required to specify the goal or motivation of attackers. Comparable terms are included in the ontologies of, for instance, Elahi et al. [EYZ09] and Miede et al. [MNG+10].

The class *Action* is used to specify concrete steps performed by an attacker to exploit a given security weakness. Modeling these concrete steps is necessary to facilitate the detection of weaknesses where the order of steps performed by an attacker matters [JKM+15]. 9 out of 29 ontologies considered this aspect; the most detailed description is found in the ontology by Elahi et al. [EYZ09], in which an attack consists of a sequence of steps, called *malicious actions* in their work.

The classes *Stakeholder* and *Process* are dedicated to users and business processes. 3 out of the 29 ontologies consider this aspect. *Stakeholder* is a sub-class of *Agent* used in particular for representing the users of the considered system. Based on the ontology of Baras et al. [BOAI14], stakeholders can be assigned responsibility over *Assets*. In addition, agents can refer to *Components* to perform specific actions. Processes represent a type of activity which consists of a number of actions. Related terms are found in the ontologies of Launders and Polovina [LP13] and Mouratidis et al. [MGM03].

4.2.2 Ontology Layering as Modularized Knowledge Base

On the one hand, maintaining an ontology with security knowledge just like the Security Context Knowledge (SCK) imposes additional burden on the security experts. But, on the other hand, there is room for synergy effects, because ontologies, in particular the OWL, offer a mechanism that supports modularization and reusability of knowledge. OWL features a mechanism to support the integration of knowledge called *import*. Apart from that, an OWL ontology can also be equipped with a *version* ID [OWL09].

While in general structure of ontology imports is not restricted, S^2EC^2O focuses on using imports to layer ontologies, building three types of layers with specific security focus. As we argued above, first, an ontology providing a core taxonomy is inevitable, to provide a base for modeling security-relevant knowledge. Second, as S^2EC^2O's focus is to support evolving concrete software system, it is also necessary to support modeling knowledge that is specific to a concrete software system, like, for instance, a role hierarchy of a system with role-based access control. Lastly, we see a need for a number of intermediate layers, to serve the purpose of using the taxonomy provided by the security upper ontology to model generic aspects of the domain the system is applied to.

Regarding the term *domain*, we focus on the definition of Eric Evans, who defines domain as "[a] sphere of knowledge, influence, or activity" [Eva04]. To be precise, for a banking system, a domain ontology may feature security-relevant measures, threats, and mitigations especially relevant from the banking domain, and, for example, incorporating rules and regulations important for this domain.

The three types of ontologies S^{2}EC^{2}O works with are as follows:

Upper The top or upper ontology is independent of a particular software domain or application. It represents the most general software security concepts, such as *encryption algorithm* and *attack* as introduced in Section 4.2.1. Thus, we call this ontology layer *Upper Ontology*.

Domain *Domain* ontologies allow domain knowledge as well as concrete security issues and measures to be captured.

Domain ontologies have to be created for each domain anew and can be shared by different systems in the same domain. We will emphasize on how to populate domain ontologies in Section 4.3.

System *System* ontologies express the security-relevant knowledge about a concrete system, for example, to specify that a banking system uses an encryption algorithm defined in the domain ontology. These system ontologies can be produced or enriched from existing artifacts, such as an UML-based system model. Section 4.3.6 gives detail on how to couple ontology knowledge with a system model.

In the ontology community, the concept of building ontologies from different other ones is also known. Scherp et al. [SSFS11] define several types of ontologies. We align the ontology types defined by us with the types defined in [SSFS11]. The security upper ontology would then be considered a *core ontology*. According to Scherp et al.,

> "[...] core ontologies provide a detailed abstract definition of structured knowledge in one of these fields, e.g., medicine, law, software services, personal information management, multimedia annotations and others."

Our definition of a domain ontology matches the one given also for a *domain ontology*, given in [SSFS11]:

> "[...] [W]ith domain ontologies we find representation of knowledge that is specific for a particular domain [...]. Domain ontologies use terms in a sense that is relevant only to the considered domain and which is not related to similar concepts in other domains."

In this work, we extend this hierarchy by system ontologies, to bridge the gap between universal and shareable ontologies and secure system design. We will discuss this aspect in detail in Section 4.3.6.

Thus, the abstraction level decreases from upper to system ontology within the Security Context Knowledge (SCK). We come back to the example Figure 4.2 shows by adding layers and import relationships. Figure 4.4 shows the result.

In the upper layer, two classes from the taxonomy of the security upper ontology are used: `Security Property` and `Asset`. The former is used to refine the notion of a security property into a `Secure Encryption` property. As an individual, encryption algorithm `RC4` is modeled.

The concept `Asset` is refined as part of the system-specific ontology as `Communication Path`, resembling the ability of at least two system components to communicate with each other.

An individual `Database Connection` indicates to be a communication path and, using the object property `has`, shows that this connection uses the RC4 encryption algorithm.

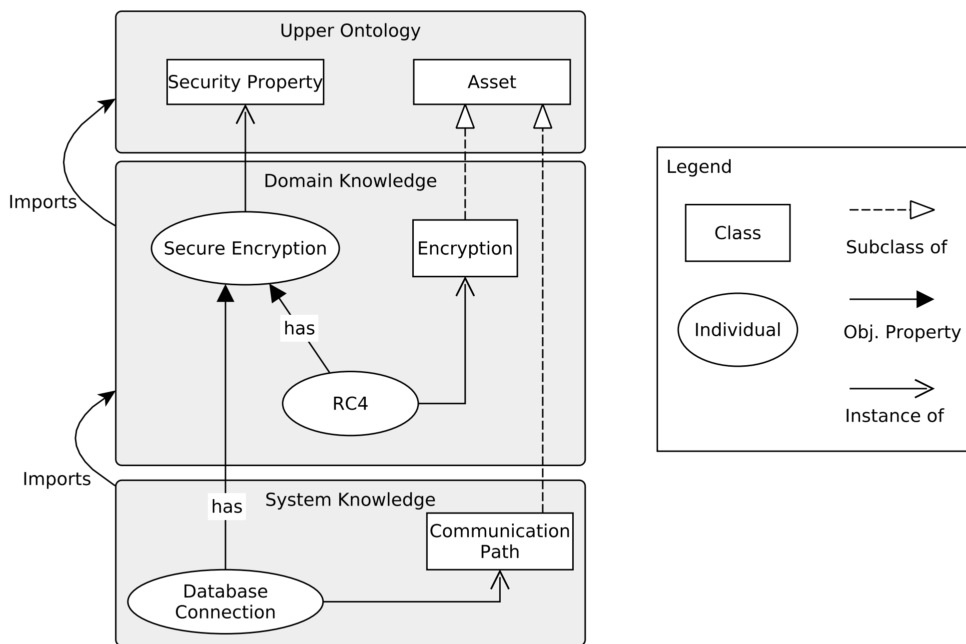


FIGURE 4.4: Example of Security Context Knowledge to provide an encryption algorithm

Sharing the knowledge to other systems is possible with regard to system- or domain-independent parts of the security knowledge. Reuse and exchange of Security Context Knowledge between different projects and abstraction levels is supported, reducing the additional effort of modeling SCK.

Naming convention Regarding S^{2}EC^{2}O, the Security Context Knowledge is composed of the security upper ontology and the layers of respective ontologies of different abstraction levels. Throughout this thesis, we use the terms *Security Context Knowledge*, *layered ontologies*, and *ontology* (as far as the context is SCK) synonymously.

4.3 Building up the Security Context Knowledge

In the preceding section, we introduced a security upper ontology, providing the base structure for the SCK. To be able to operate, the SCK needs to be populated with actual security knowledge in terms of additional ontology classes (potential, concrete encryption algorithms, for example). In this section, we argue which sources of knowledge for security guidelines, incidents, attacks, and mitigations are available publicly. After that, we sketch an approach of how this knowledge can be incorporated into the SCK to make it usable.

We begin with enumerating various sources for security knowledge available in non-formal (i.e. natural language) and semi-formal (i.e natural language parts but provided with meta data) manner.

4.3.1 Laws and Regulations

While work on this thesis began, on EU level, the directive 95/46/EG [EU 95] and its refinement in Germany, the Bundesdatenschutzgesetz BDSG [Bun05] were still

valid. In May 2018, the General Data Protection Regulation [EU 16] came into effect, replacing both laws in Germany. These laws build a foundation that most software systems running in Germany, or the EU respectively, need to comply with.

There also have been changes to the German BDSG during its legal force [BfD14] which also needed existing systems to be altered to remain compliant with the new version of the law. We investigated such changes needed by privacy regulation changes in preliminary work [BSG+18, BGR+15].

A major change introduced by the General Data Protection Regulation (GDPR), is for example that the user of a data processing system has to be informed what data is collected, how it is processed, and under what circumstances it is handed over to third parties. The user needs to give consent upfront and also should have the right to get access to all of the personal data a system has stored and also has the right to get all of it deleted irrevocably. These laws may have fundamental influence on a software system's data processing design.

4.3.2 Standards and Guidelines

The German Federal Office for Information Security (BSI) issues a series of Technical Guidelines (Technische Richtlinie), for instance giving recommendations for the use of cryptographic algorithms [Bun18]. Using this, it is possible to keep track of security algorithms used in a software system that are considered to be secure.

Regarding guidelines for the development of secure software systems, there is also the *Open Web Application Security Project* (OWASP) [Thed]. The maintainers provide an open web presence documenting exemplified attacks and vulnerabilities and also guidelines of how to design secure software. Moreover, they also issue a top 10 of security risks. Since the Open Web Application Security Project (OWASP) is structured and aligned to a taxonomy, incorporating the knowledge into an ontology is facilitated. Apart from that, the OWASP also provides an ontology called *Automated Threats to Web Applications*. The term ontology may be misleading here, because it is not available in the form of, for example, an OWL ontology or RDF triples, but rather as wiki pages. Nevertheless, entries do contain cross links to other knowledge bases like CWE, which we will introduce in a subsequent section.

4.3.3 Attack Scenarios

One possibility to gain insights in possible attacks is the *Common Attack Pattern Enumeration and Classification* (CAPEC) [MITa]. It provides numerous attack scenarios a software system may need to be secured against. The catalog is also structured after domains and mechanisms used for an attack. For example, *input data manipulation* is one attack mechanism. The entries are described by natural language and feature both references to other CAPEC entries and further attack pattern databases.

4.3.4 Vulnerability Databases

Regarding vulnerabilities, two well-known catalogs related to each other are *Common Vulnerabilities and Exposures* (CVE) [MITb] and *Common Weakness Enumeration* (CWE) [MIT17b]. The CVE is a database of vulnerabilities a concrete software product in a specific version was affected. The CWE is a structured catalog of typical weakness types in software.

Regarding this chapter's example considering secure communication, the entry *CWE-326 Inadequate Encryption Strength*, describes the weakness introduced by an

encryption algorithm which performs insufficiently in terms of information security. An encryption algorithm especially is inadequate as soon as an attack becomes disclosed [BSG+18]. For each CWE entry, possible consequences, an example, cross references to other catalogs like the OWASP Top 10, and possible mitigations are eventually provided.

While the CWE is focused on providing general information and advice on how to build secure software, *Common Vulnerabilities and Exposures* (CVE) is a catalog of actual security incidents taking place in existing products. For example, CVE-ID CVE-2009-1284 describes a vulnerability in BibTeX:

> "Buffer overflow in BibTeX 0.99 allows context-dependent attackers to cause a denial of service (memory corruption and crash) via a long .bib bibliography file."

Cross-references to, for example, CWE entries are also provided. Information available via CVE and others is given in a directly processable form.

The *National Vulnerability Database* (NVD) [Nat] is a service run by the U.S. government. Security experts scan newly discovered security issues, make cross references and incorporate the knowledge into a database that is available as a XML file.

4.3.5 Incorporate Security Knowledge into the Security Context Knowledge

In the preceding sections, we introduced a number of sources for security knowledge to populate the SCK. However, as these knowledge sources are not complying with the security upper ontology, the question arises how to incorporate that knowledge into the SCK.

Proposing an approach on how to incorporate security knowledge from the various sources, semi-formal as well as natural language texts into the ontology is beyond the scope of this thesis. We will give an overview of existing approaches to give suggestions of how to accomplish this instead.

Active Learning

An approach adequate for natural language input is presented in the PhD thesis of Stefan Gärtner [Gär16]. The main idea is to examine development artifacts for security vulnerabilities. To accomplish this, Natural Language Processing (NLP) techniques are used. Using heuristics, process-oriented development artifacts such as use case descriptions are compared to descriptions of attacks and can thus be linked to security properties at stake and possible mitigations.

The ontology in the approach is used to link the security-relevant as well as development-related descriptions to concepts at a higher level.

The proposed approach makes use of a learning technique called *Active Learning*, we will sketch shortly here. Figure 4.5 shows the principle of active learning and pool-based sampling [Set12].

The initial learning set needs to be provided manually. Using this, an instance-based classifier is trained (1). The classifier then classifies the elements in the pool (2). From the elements in the pool, the element with the biggest uncertainty is selected (3) and the expert is asked to classify it (4). This learning cycle is iterated until the maximum uncertainty remains under a given threshold or a maximum number of cycles has been reached.

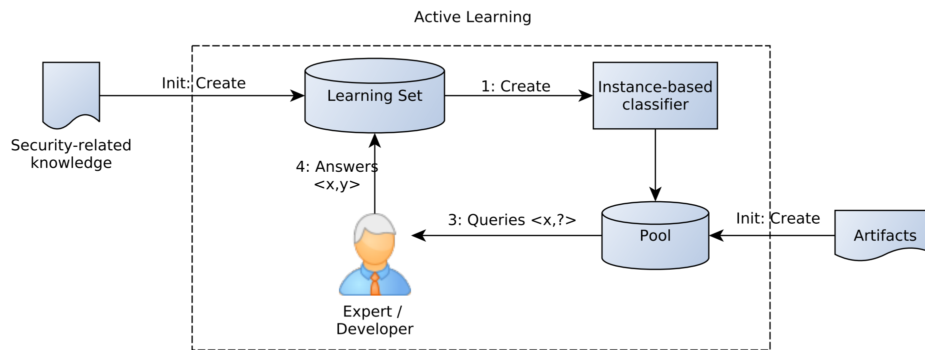


FIGURE 4.5: Overview of active learning using pool-based sampling (adapted from [Gär16])

Approaches specific to Knowledge Sources

Apart from this universal approach, there is also work tailored to specific knowledge sources. While being only usable for the knowledge sources designed for, this work can profit by using structure and/or meta data as provided by the specific sources and thus provide more automatism and needing less user interaction.

As we mentioned previously in this section, the National Vulnerability Database (NVD) is provided by security experts and incorporates preprocessed security knowledge related to CVE, Common Vulnerability Scoring System (CVSS), and others. The data is available as XML- and as JSON-feed. Due to the structure and internal links already provided, an approach making use of this information to improve transformation of the knowledge into an ontology is conceivable.

Joshi et. al [JLFJ13] propose an approach where knowledge from the NVD is translated directly into RDF triples which in turn can be directly incorporated into their ontology. The natural language description parts of the database entries are processed using an algorithm based on a *conditional random field* (CRF) implementation provided by a Stanford named entity recognizer.

Du et al. [DRW+18] propose another approach. They also use the data feed and transform the knowledge rather directly into an ontology using scripts written for the graph framework Neo4j [Neo].

We will cover further in the related work section of this chapter.

4.3.6 System Level Knowledge

The SCK needs security-relevant elements from the UMLsec system model to be present in the ontology so that relations (object properties, for example) between system elements and the security knowledge can be defined.

This work is up to the security expert, but a manual approach would result in a vast amount of additional and yet error-prone work, since every security-relevant elements would have to be modeled both in the UMLsec model and the SCK.

We provide support for coupling these two model types by proposing a UML profile, offering a UML stereotype. UML profiles provide a native an lightweight method of extending UML models with additional elements [Obj17]. The advantage of extending the UMLsec model with SCK information is that relevant elements of the system model can be tagged with their security-relevant notions and relations. The other way around, every SCK class and individual would have to be annotated with all of its applications in the UMLsec model which would lead to less clarity.

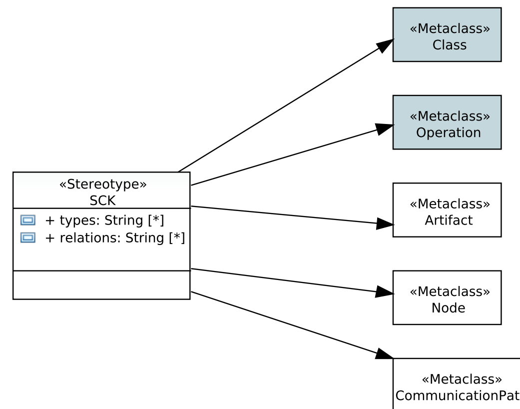


FIGURE 4.6: UML profile to annotate UMLsec models for the SCK

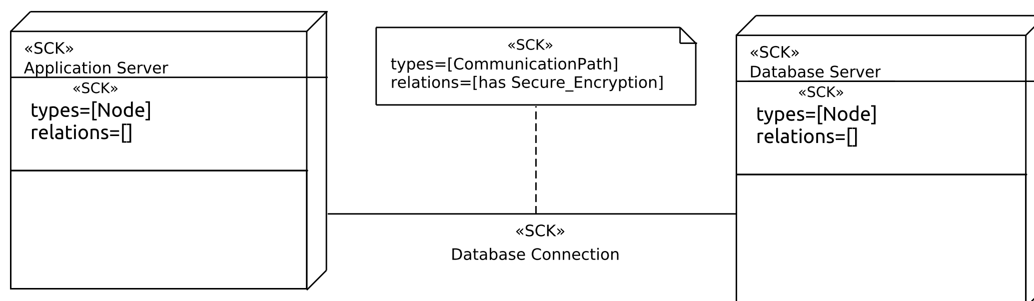


FIGURE 4.7: Example of UML annotations to support bridging to the SCK

Figure 4.6 shows the profile. The stereotype ⟪SCK⟫ can be attached to all UML elements used for S^{2}EC^{2}O so far and can be extended to support additional classifiers. It provides two attributes: `types` can be used to express which (ontology-)classes the annotated UML element should have in the SCK, while `relations` does the same for OWL object properties.

Figure 4.7 shows an excerpt from an UML model regarding the database connection example in Figure 4.4. Here, an excerpt of a deployment diagram is given, showing two nodes which communicate over a communication path. The path is annotated with ⟪SCK⟫. It is defined that, in the SCK, the path should be present and have the class assertion `Communication Path`, and it shall have a relation to the ontology individual `Secure_Encryption` using the object property `has`. Please note that due to technical reasons, spaces in OWL ontologies can be given by an underscore (_).

Thus, using a (programmatic) transformation, providing individuals for a *System* level ontology containing the relevant elements is possible which just imports the lowest domain ontology in the layering to complete the SCK.

Another approach is to employ graph transformation techniques to carry the elements from the UMLsec model over to the ontology respectively. Models can be interpreted as graphs which especially holds for UML models as well as for OWL ontologies [W3Ca]. Henshin [ABJ+10, SBG+17] and eMoflon [eDT] are two candidates for this. Another related approach is presented by Walter et al. [WSR10]. They present an approach based on the work on TGraphs by Ebert et al. [ERW08]. TGraphs are typed and attributed graphs, for which UML support, querying and

transformation support exists. The approach of Walter et al. targets translating UML models into OWL ontologies. A transformation from TGraphs to OWL is presented, too.

4.4 Managing the Knowledge Base

In the preceding sections, we showed how to build the SCK. The security upper ontology provides a foundation, we introduced various sources for security knowledge, and we also presented a way to couple system model design and the SCK.

S^{2}EC^{2}O requires that the Security Context Knowledge (SCK) can be accessed to acquire knowledge as shown with the activity *Query SCK for details* in Section 2.2.3 (on page 15). In the following, we will show potential interfaces to request knowledge from the SCK and also to alter the knowledge. To cope with this challenges, ontology query languages are a widespread concept [BBFS05].

As an example, we want to query the SCK for a set of encryption algorithms that is currently not threatened by attacks. This can be used to provide an appropriate encryption algorithm to be integrated in the example model shown in Figure 4.7.

4.4.1 Ontology Queries

The Security Context Knowledge (SCK) should be accessible automatically, using a query language that is widespread, i.e. is supported by a number of tools and there should also be a possibility for the S^{2}EC^{2}O user to check and test new queries.

Protégé [Sta] as the de-facto standard in the ontology domain, supports mainly three query languages we investigated:

1. DL Query
2. SQWRL
3. SPARQL

DL Queries (DL is short for Description Logic) are based on *Manchester Syntax* class expressions [W3Cb]. DL Query lacks support for variable binding and also prefixes (i.e. different ontologies in the layering) cannot be addressed explicitly.

SQWRL [OD09] is a query language based on the SWRL rule language (SWRL is short for Semantic Web Rule Language). At the time of writing, the syntax supported inside Protégé does not support negation for object properties. This is a drawback, because, regarding the example, to specify a set of encryption algorithms that is NOT threatened by, for example, a *Threat* is not possible directly.

In contrast to that, SPARQL [W3Cc] (pronounced *sparkle*, a recursive acronym) is a W3C standard. Using a graph framework like Apache Jena [Thea], it is possible to answer SPARQL queries via an API. Reasoners like Pellet [PS04] do also support SPARQL queries. We will cover reasoners in detail in Section 5.3.2.

SPARQL also does support variable binding of multiple variables and, with constructs like `MINUS`, `NOT EXISTS`, `FILTER`, results can be explicitly narrowed. The syntax of SPARQL is similar to the *Structured Query Language* (SQL) used for relational databases, for example [ISO16].

Hence, SPARQL is the only query language to provide essential features needed by S^{2}EC^{2}O and is selected as standard query language.

```
1 SELECT ?algorithm
2 WHERE {
3 ?algorithm a domain:Encryption;
4 upper:has domain:Secure_Encryption.
5 FILTER NOT EXISTS {?t a upper:Threat.
6 ?t upper:threatens ?algorithm}.
7 }
```

LISTING 4.1: SPARQL query for encryption algorithms for which no threat is known

Listing 4.1 shows an example of a SPARQL query to determine encryption algorithms for which no threats are known. First, a variable `algorithm` is defined using `?` as prefix (line 1). The `WHERE` clause defines the properties that have to be fulfilled by `?algorithm`, comprised by the curly braces (lines 2 - 7). Firstly, `?algorithm` is required to be of the type `Encryption`. As this type in turn is an individual of a ontology, in this case a domain ontology, it is required to be defined using its complete *Internationalized Resource Identifier* (IRI), for example `http://rgse.uni-koblenz.de/domain#Encryption`. As using complete Internationalized Resource Identifiers (IRIs) over and over, this would bloat SPARQL queries, this can be shortened by defining so-called prefixes. Prefixes need to be provided additionally to a SPARQL query and precede it. For `domain`, the prefix is specified as follows:

```
PREFIX domain: <http://rgse.uni-koblenz.de/domain#>
```

Prefixes for the other ontologies are defined analogously. The keyword `a` specifies the variable to be of the respective type (line 3). As next step, the algorithm is required to have the `Secure_Encryption` property. For this, the object property `has` from the upper ontology is used as well as the individual `Secure_Encryption` as defined in the domain ontology. Using `;` as delimiter in line 3, the statement in line 4 is also applied to `?algorithm`. The full stop `.` ends the statement.

The WHERE clause so far (lines 3 - 4) queries all encryption algorithms from the knowledge base. We now need to filter the results (i.e. algorithms) that are currently threatened. We realize this using the `FILTER NOT EXISTS` clause and describe that we want every result removed where an individual `?t` of type `Threat` exists that `threatens` a given `?algorithm` (lines 5 - 6).

To conclude, we showed that, using SPARQL, we are able to query the SCK.

4.4.2 Ontology Updating

As we argued above, it may be desirable to also provide an interface to alter the knowledge stored in the SCK using a well-defined interface. While a detailed treatment of this issue is out of this thesis' scope, we shortly sketch how this could be realized. In addition to the SPARQL query language, there is also a specification provided by the W3C called *SPARQL Update* [W3Cd]. SPARQL Update defines a syntax that can be used to specify updates on the knowledge base, being a superset of the SPARQL syntax.

This would facilitate automated updates of the knowledge base. For instance, Apache Jena currently supports SPARQL Update queries [Thea]. However, updating the knowledge base without any user interaction is non-trivial when knowledge is to be added that contradicts already existing knowledge, being a current research objective [ACPS16]. At the time of writing, the user is requested to update and check the SCK manually, for example using Protégé.

4.5 Related Work

In this section, we discuss related work of this chapter. As the publications that we took into account only cover a some aspects of this chapter, we categorized them.

4.5.1 Compliance Checking of Ontologies

Humberg et al. [HWP+14] present an approach using ontologies to check compliance of cloud-based processes. The authors present an analysis approach that uses an OWL ontology as knowledge base. A small number of classes and object relations is given as base which is to be extended when various sources for compliance requirements like the German IT-Grundschutz Catalogues is parsed. These build a number of *rules*. A number of rules again constitute a *situation* which can be compared to recognition patterns. The cloud-based process is assumed to be modeled in BPMN. The process is examined regarding eventually existing situations. This is done analyzing the activity labels and utilizing word databases which consider co-occurrent words and synonyms. As soon as a situation is found in the model, it can be checked whether the process is compliant with all rules. For example, the constraint *Separation of Duties* can be checked in the model by assuring that activities are performed by different actors.

The approach does not tackle evolution explicitly, while work towards using imports and adaptation operations in ontologies is suggested. Progression of this work as part of the SecVolution project is published later by Ruhroth et al. [RGB+14b]. While there is a minimal structure comparable to an upper ontology provided, this has only little structure to rely on. Furthermore, sophisticated features of ontologies like reasoning are not used. All system specific information is handled outside the ontology to keep it reusable. In contrast to that, S^2EC^2O leverages an ontology layering to modularize the knowledge base.

Da Silva et al. [DSRdV+07] present an ontology for information security. The ontology is built by extracting knowledge from natural language text like information security standards, security policies, and security control descriptions. The approach is based on an OWL DL ontology. Experiences from a project context showed that the information system experts preferred to work focused on text, for example standards, human-readable protocols, and security control descriptions. As part of the approach, a Protégé plug-in was implemented. The plug-in is a customized editor where the specialist can mark text parts and thus categorize actions, security controls, and objectives. The plug-in internally populates the ontology with the respective individuals and creates relations; additionally, Description Logic (DL) formulas are generated to allow semi-automatic checking. Afterwards, Description Logic (DL) reasoners are used to work on the generated formulas.

4.5.2 Knowledge Elicitation

Harmain and Gaizauskas [HG03] utilize NLP for eliciting software requirements. With their technique they are able to detect entities related to classes and attributes to build an initial UML class model.

In contrast to S^2EC^2O they are not considering requirements in context of security issues.

Compagna et al. [CEK+08] integrate legal patterns into a requirements engineering methodology for the development of security and privacy patterns using NLP. This description is parsed by a natural language processor on the basis of a semantic

template. The pattern design and validation process requires legal experts to describe patterns in natural language. While providing a possibility to model entities and their security needs in a SI* model, there is no process provided of how to hand the knowledge over to subsequent software development steps.

4.5.3 Security Requirements Elicitation

Gegick and Williams [GW07] developed a methodology for early identification of system vulnerabilities for existing threats based on regular expressions. Patterns of possible vulnerabilities are used to identify threats. The method is called Security Analysis for Existing Threats (SAFE-T) and thus is not tailored to deal with newly discovered threats and a system that is deployed.

Tsoumas and Gritzalis [TG06] provide a security ontology based framework for enterprises linking high level policy statements and deployable security controls. The security ontology is build by extending the DMTF Common Information Model standard. It is used as a container for the security related information that concerns the information system. This approach targets organizational security controls (for example securing server hardware, recommending using a firewall) and not developing secure software systems, in contrast to S^2EC^2O.

Sindre et al. [SO05] argue that use cases offer limited support for eliciting security requirements and therefore regard use cases enriched with textual description about misuse. Based on this, guidelines are presented how to describe misuse cases in detail, so that, in a next step, method guidelines for eliciting security requirements with misuse cases can be established.

While providing a general approach on how to elicit security requirements based on use case descriptions, in contrast to our approach there is no focus on overall security goals/requirements and relations to the later system design.

Kaiya et al. [KSOK13] use information about the underlying architecture to elicit security requirements. Use-case descriptions are converted into data-flow diagrams called asset-flow diagrams. Security requirements are then defined as countermeasures for an attack if detected by the asset-flow diagrams and as design and implementation constraints if not detected. In contrast to S^2EC^2O, there is no systematic approach for evolving stakeholder needs.

Haley et al. [HLMN08] present a framework not only for security requirements elicitation, but also for security analysis. Their method is based on constructing a context for the regarded system. Describing this context with a problem oriented notation makes it possible to validate the system against the security requirements. The approach is very powerful but needs a lot of security expertise to build the context and understand the results of the analysis. Evolution of the context is not supported here.

Huan et al. [NVLG14] propose a *framework for knowledge-based requirements engineering*. Description Logic is used as base to analyze the requirements and realize reasoning. Tool support is provided. The framework features a process beginning with a requirements engineer specifying requirements. Requirements are processed into an internal ontology. The Manchester OWL syntax is used to specify requirements and the approach is tailored to Goal-oriented requirements engineering. Using a provided ontology editor, the requirements engineer can refine and link the specified requirements and also define rules over them. A Manchester syntax parser updates the internal ontology. Analysis based on reasoning is used to detect requirements problems, giving answers to requirements queries and generating explanations.

Chapter 5

Leverage Changes in the System Context for Secure System Design

In the preceding chapter, we introduced the Security Context Knowledge (SCK), how a taxonomy is provided (the upper ontology), how the knowledge base can be populated with security knowledge, and how the knowledge base can be modularized. We also showed how the knowledge base can be managed.

In this chapter, we put emphasis on the question how changes to the knowledge base can be detected. A knowledge base built from different sources, as the SCK is subject to change in manifold ways. We thus focus on detecting evolutions on a sophisticated manner, to foster directed analyses and reactions later on. We will investigate ontologies from a model point of view and calculate differences on a *semantic* manner. We will explore possibilities that ontologies and accompanying techniques offer to detect changes as well.

Hence, this chapter contributes to the following research question:

RQ1: *How can changes in security-relevant context knowledge be used to assess the impact on the system?*

S^{2}EC^{2}O's focus is to support the system co-evolution when the context, i.e. the SCK, evolves. Section 5.1 first introduces the underlying problem of co-evolution and provides an overview of the techniques incorporated by S^{2}EC^{2}O to leverage changes of the knowledge base. Section 5.2 shows in detail how differences in the knowledge base can be recognized using *semantic differencing*. Section 5.3 introduces a concept using ontology reasoners to infer additional knowledge or gain insights after an evolution has taken place.

The concluding section of this chapter, Section 5.4, presents related work. Review of the tackled research question will be carried out in Section 11.1.

5.1 Detecting and Assessing Knowledge Changes

Figure 5.1 illustrates the relation between evolution and co-evolution of the system model and its context as we consider it in this thesis.

The lower layer shows the *system model*. Using S²EC²O, the system development is accompanied by the SCK, shown in the upper layer.

When a system is initially developed, it is ideally compliant with all of its security requirements regarding the security knowledge, i.e. it passes a *security analysis* (shown in the middle left of the figure).

As soon as an evolution of the SCK has taken place (shown as ev_{SCK}), an appropriate co-evolution needs to be determined so that the evolved SCK and the co-evolved system model lead to passing the security analysis again.

The following sections focus on ev_{SCK}, called *Delta SCK*, and investigate techniques how changes to the knowledge base can be determined.

Regarding the question of how the SCK can be altered externally: there is a plethora of possibilities how this can be accomplished. Besides the fact that different tools for the end user exist to view and alter ontologies, OWL and underlying serialization types like OWL/XML and RDF triples are supported by a wide range of other frameworks and tools, for instance graph frameworks like Neo4j and Apache Jena [Neo, Thea]. Additionally, it is possible to access and manipulate the SCK via arbitrary program code. This can be realized using APIs like OWL API, on which for example Protégé is based.

Hence, for S²EC²O we decide the way how to gather evolution of the knowledge is not by making use of internal versioning approaches or by tracing access to the SCK by a specific technology, but rather by analyzing an ontology directly as given, or two versions of the same ontology. Going this way, we can handle all sources of change that can lead to ontology evolution.

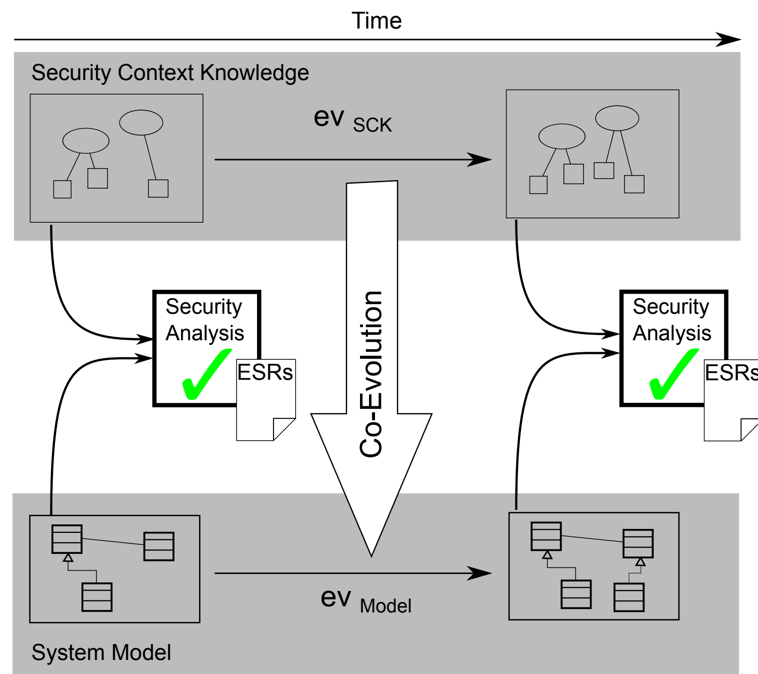


FIGURE 5.1: Relationship of evolution and co-evolution with regard to model-based security engineering

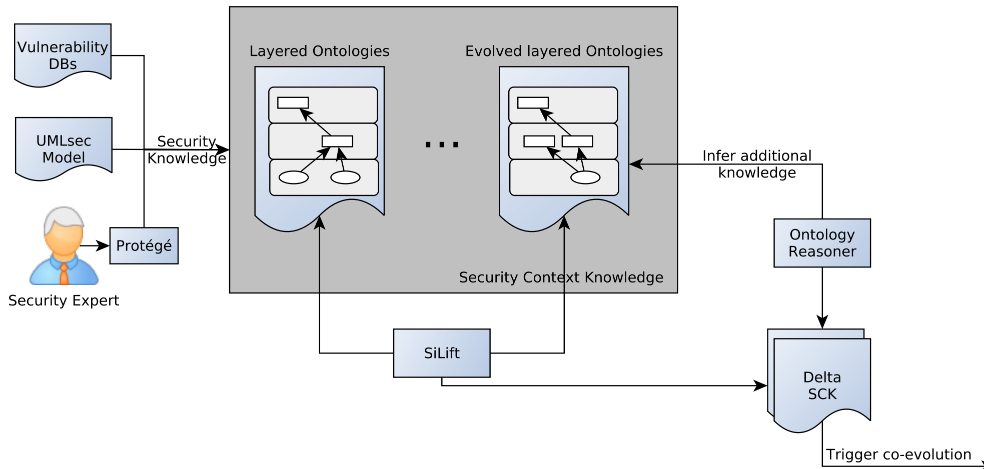


FIGURE 5.2: Overview of different approaches accessing the Security Context Knowledge in S^{2}EC^{2}O

Techniques to Identify Delta SCK

Figure 5.2 shows an overview of the techniques to provide (updated) security knowledge to the SCK. As we introduced in Section 4.3, various knowledge bases like *vulnerability databases* can contribute to the SCK, as the figure's left side shows. Moreover, the system model can also contribute in terms of security relevant elements, like, nodes of a distributed system as well as their communication paths. For details, we refer to Section 4.3.6.

The *security expert* is able, for example using *Protégé* as tool, to edit, maintain and refine the ontology. This leads to a number of evolving ontologies. In this thesis, we focus on two techniques to investigate the SCK for SCK evolution information (Delta SCK).

Semantic Differences between two given versions of the same ontology can be detected using the *SiLift* approach which we will discuss in Section 5.2.

The current given ontology can be analyzed using an *ontology reasoner*. Inconsistencies can be detected and, using justifications, explained to S^{2}EC^{2}O's user to take further action. Moreover, implicit knowledge can be made explicit. We introduce details on this in Section 5.3.

Both techniques we discuss in this chapter are used to investigate the SCK to generate SCK evolution information (Delta SCK) in a (semi-)formal way, being the trigger for potential co-evolution steps in S^{2}EC^{2}O's delta process.

5.2 Semantic Differencing

In this section, we take up again the example of modeling secure communication in the SCK as introduced in the preceding chapter.

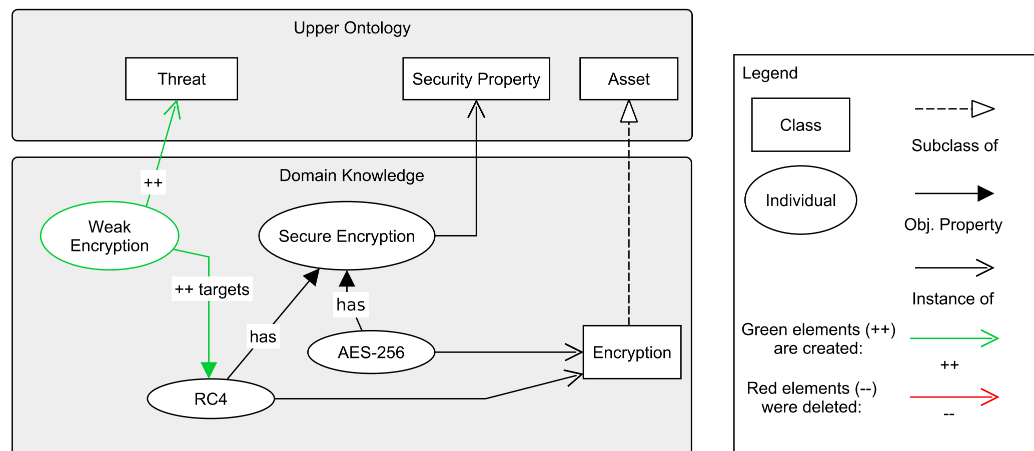


FIGURE 5.3: Example of evolving Security Context Knowledge: An encryption algorithm is discovered to be vulnerable.

5.2.1 SCK Evolution Example

Figure 5.3 shows an excerpt of the SCK, being involved in an evolution. In the preceding chapter, we already showed how a system model can be connected to the SCK and how the knowledge base can be queried to determine potential encryption algorithms for which no attack is currently known.

The figure shows an evolution of the knowledge base in a unified view. Elements in green are added and elements in red are removed, while the remaining elements are part of both versions.

In this example, the encryption algorithm `RC4` (as introduced in Section 4.2.2 on page 33) is modeled as an *encryption* algorithm providing the security property `Secure Encryption`. A second algorithm, `AES-256` is also modeled (both in the center of the lower gray rectangle). The knowledge evolution shows that `Weak Encryption` is added as a new `Threat`, targeting `RC4`. In the following sections, we will show how to detect an evolution like this using semantic differencing.

5.2.2 Semantic vs. Atomic Changes

Information systems in model-based development consist of different software models such as system model and security model. Moreover, different development groups are involved in developing the system and at the most, they work on different parts of the system and of course have different viewpoints on the system. Every entity participating in the development process may cause changes to the corresponding models without notifying the other participants. Thus, if there are commonly shared models among one information system, no development participant can assume the models to be unchanged when trying to apply changes on its own. This means that every model-based approach used to develop software can change models and some other participants have to be informed about these changes. For a sophisticated reaction on evolution, the semantic meaning of changes is inevitable. As we argued in Section 4.2, ontologies provide a flexible approach to manage the knowledge base. Three challenges arise from the purpose of extracting evolutions from edit operations:

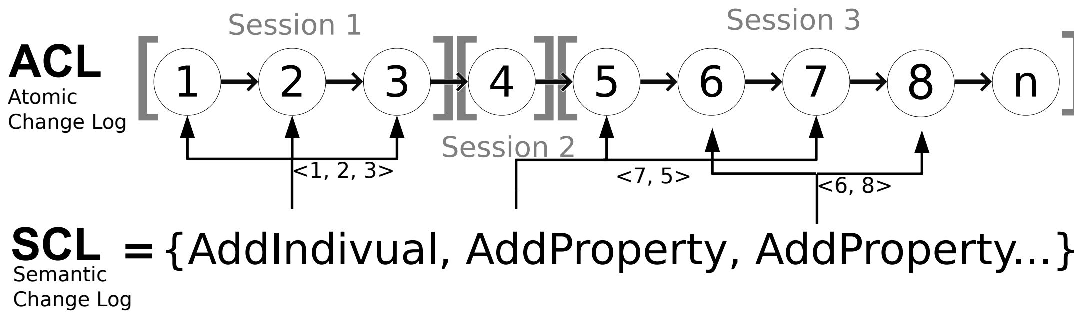


FIGURE 5.4: Comparison of atomic change log and pattern-based change log: Atomic change operations can be presented by semantic change operations.

The first one arises by diversity of the change operations' granularity. Many tools allow pre-defined edit operations on models, but as edit steps can be of varying granularity, they do not necessarily describe evolution steps semantically. This especially applies to ontologies. As the knowledge base can be extended arbitrarily, edit operations provided by an ontology tool like Protégé [Sta] always are restricted to the basic terms of ontologies.

The second challenge is that different semantic evolutions can be performed in an interleaved manner.

The third challenge arises from the fact that models like the ontologies representing the Security Context Knowledge may be also changed semi-automatically by external processes, for example when knowledge about attacks and mitigations evolves.

We illustrate these challenges by discussing an example that Figure 5.4 depicts.

Consider a set of changes to be carried out on a model, in this case the SCK. Each change is a separate and disjunct addition of information which is applied within the scope of a maintenance task and, thus, can be described semantically. An example for a task is: *Add AES-4096 as encryption algorithm.* However, each task may consist of several edit operations which ultimately alters the model. Consider that the above mentioned task has to be carried out on the ontology shown in Figure 5.3. First, a new ontology individual has to be created. After that, a type assertion can be added, and then object property assertions can be added.

Imagine that a security expert is assigned the task of incorporating two changes into the SCK. In *session 1*, he adds the individuals (see the upper half of Figure 5.4). After that, the SCK is changed externally (shown as *session 2*). Finally, the security expert completes his task by adding the necessary object properties (shown as *session 3*).

Assuming that the tool used to alter the SCK logged all editing operations, which can only take place on the atomic level, the result would be a stream of fine-grained edit operations as the upper part of Figure 5.4 shows. Given these differences, different steps of the evolution are executed in an arbitrary and interleaved manner. Thus, we cannot simply regard a sequence of atomic operations as evolutions, and it is desired to create a log of semantic changes based on pre-defined change patterns as the lower half of Figure 5.4 exemplifies.

Consequently, *semantic differencing* is necessary to express evolution of the SCK in a suitable way, more coarse-grained and also domain or even system-specific. To accomplish this, we make use of the *SiLift* approach [KKT11].

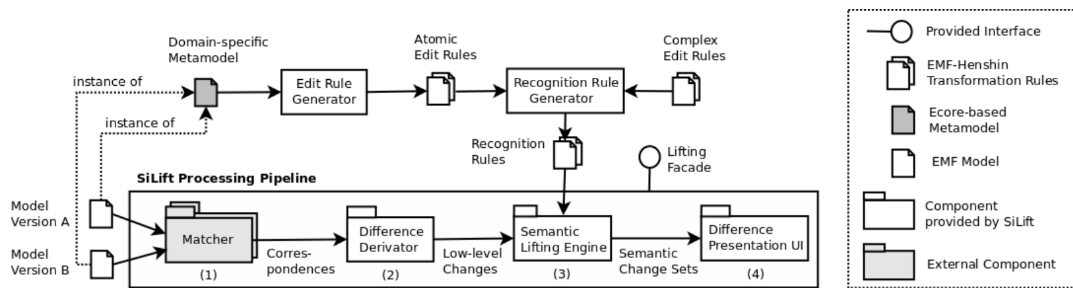


FIGURE 5.5: The SiLift process (from [KKOS12])

5.2.3 Using SiLift to detect SCK Evolutions

SiLift is built upon Eclipse EMF [SBMP08]. The models are interpreted as graphs and analyzed as well as altered using the graph transformation language and tool Henshin [ABJ+10, SBG+17]. Graph transformations and especially Henshin are introduced in Chapter 3. Henshin is also used in our preliminary work [BGR+15, BJW15].

SiLift can be used to realize the semantic change log as shown in Figure 5.4. Figure 5.5 shows an overview of the SiLift process to calculate differences. SiLift calculates semantic differences between two versions A and B of a model which corresponds to an EMF-based meta model. Before SiLift is able to start to work, the following artifacts need do be provided:

- A set of all possible atomic edit actions on the low-level, called *atomic edit rules*, specified as Henshin transformation rules. These can be generated automatically by SiLift using the meta model definition.
- A set of pattern definitions for semantic changes, called *complex edit rules*, which is equivalent to the change operations we show in Figure 5.4. Thus, a complex edit rule can be expressed by a number of atomic edit rule applications.

  Both kinds of edit rules are then translated automatically into a set of recognition rules. This step is necessary for technical reasons, because the complex edit rules specified by the user need to be modified so that there are separate nodes that can be matched to model version A and B separately.
- A *matcher* component that basically needs to provide information about when two objects in question are equal. The matcher needs to be implemented against SiLift's API.

The SiLift process then works as follows (see Figure 5.5). For details we refer to [KKOS12]:

(1) The matcher is used to calculate correspondences, i.e. the set of elements in model version A and B that did not change.

(2) The set of correspondences is used by the difference derivator to compute a set of low-level changes according to the atomic edit rules. This means that occurrences of atomic edit rules are matched against the model versions.

(3) The semantic lifting engine then takes the recognition rules into account to match them against occurrences of atomic edit rules, to infer which complex edit operations have been applied. A compilation of atomic operations, corresponding to a semantic operation is called a *semantic change set*.

(4) The semantic change sets are shown to the user in a UI that provides further operations. Note that SiLift can also be used via an API. S^2EC^2O interacts with the user and processes the information provided by SiLift rather than presenting it to the user.

SiLift is able to calculate the semantic difference in two ways: asymmetric and symmetric. The symmetric difference is adapted from set theory; it consists of a set of correspondences between two model versions, followed by information of how to calculate one or another model version from it [Men02]. We use the asymmetric difference, also known as *edit script* [KKT13]. It is calculated in the following way: a series of edit operations (the edit script) is composed that, when applied in order to version A of the model, version B of the model is obtained.

In what follows, we present a *complex edit rule* able to recognize the SCK change introduced by Figure 5.3 (on page 48), defining a semantic evolution operation. Presentation of the complex edit rule is preceded by remarks regarding Henshin rule layout conventions.

5.2.4 Henshin Rule Layout Conventions

The OWL EMF meta model as provided by the W3C [W3Ca], adhering to the structure of OWL, has a flat design, which results in a relatively high number of classes and relations. This in turn results in a high number of nodes in Henshin rule design. Precisely, in OWL most individuals get their identity by a linked `URI` object. Nearly all other types do not have attributes but only relations to other objects. Figure 5.6 shows an example of a semantic difference specification as used in S^2EC^2O.

Consider the two nodes of the type `Class` in the upper left and upper right part of this rule. While the nodes itself do not have any properties, their *identity* is defined by the other associated elements on the one side and a name-defining `URI` element on the other side.

To raise readability, we arrange Henshin rules for SiLift in this work according to the following layout conventions:

- In associations, the association's type (`entity`, for example) is placed above, while the action type (《preserve》, for example) is placed below the line.
- A `URI` element is placed to the right of the node it specifies.
- Due to technical reasons of the SiLift post-processing, all 《create》 nodes need to be associated to an `Ontology` container node. These elements are placed in the upper right of the respective containing node.

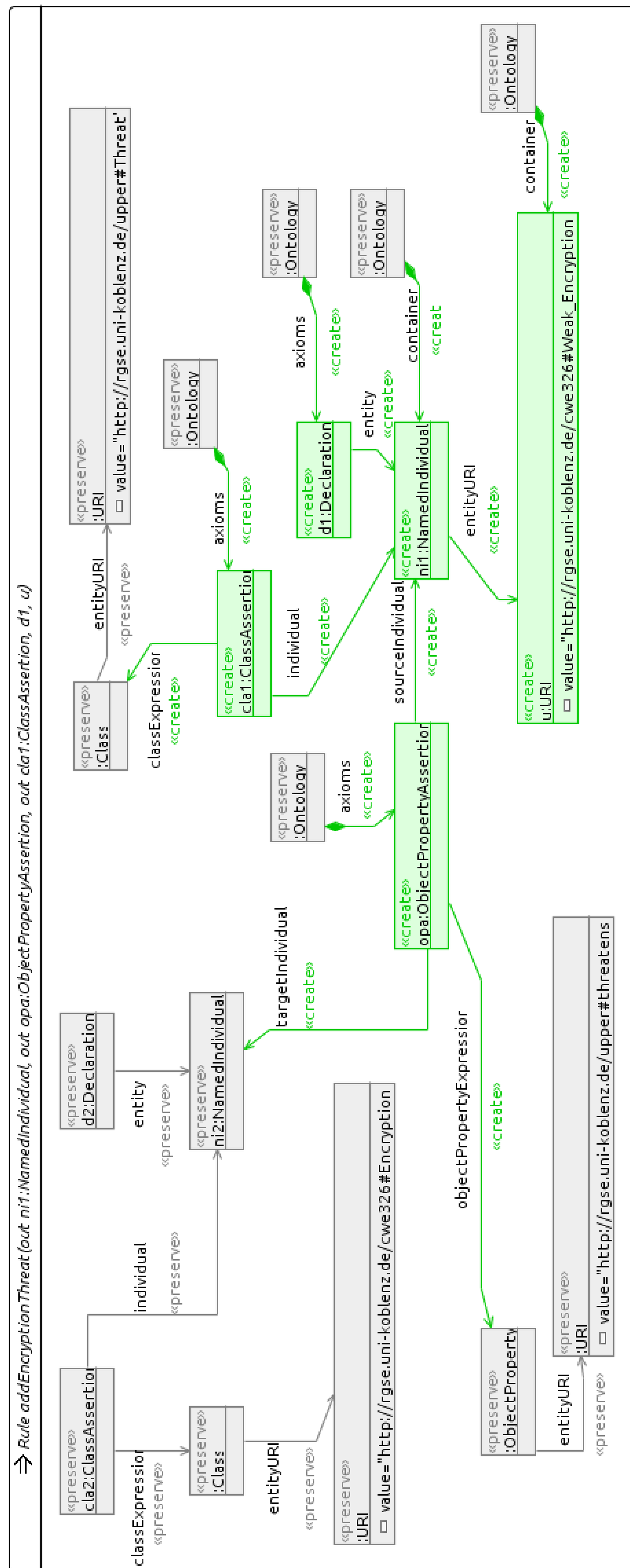


FIGURE 5.6: Example of complex edit rule describing addition of a *Threat* to an existing *Encryption* of the Security Context Knowledge

5.2.5 Complex Edit Rule Example

Figure 5.6 shows a SiLift rule that is able to detect the evolution introduced in Figure 5.3 on page 48. The example pattern *addEncryptionThreat* specifies the situation relevant for security knowledge updates: A threat for a given encryption algorithm is discovered. As we argued above, entities do not have attributes that make them distinguishable directly, but their identity is given by relation to a URI object via an `entityURI` relation. «preserve» nodes specify that there already exists a notion of *Encryption*, and *Threat* accordingly. Both are specified using the same structure: there is a `Class` node getting its identity by association to a `URI` object. The object relation `threatens` is defined in the lower left. These elements come from the structure as given by the Security Context Knowledge and extensions by incorporating security knowledge as introduced in Section 4.3 on page 35.

The actual difference is specified by `create` nodes. An addition of a threat is modeled. The individual in question is compiled by the following nodes: `cla1`, `d1`, `ni1`, and `u`.

The individual constituting the threat has the name `Weak_Encryption` and the type `Threat` accordingly. The association to an existing individual of the type `Encryption` using the object relation `threatens` is defined by the object property assertion in the center of the figure.

Thus, the difference statement modeled by this rule is:

> An individual of the type `Threat`, called `Weak_Encryption` is added. The individual has the object property association `threatens` targeting an already existing individual of the type *Encryption*.

After the SiLift process for a given set of rules and ontology versions has finished, it eventually has found matches of rules like the one shown in Figure 5.6. Basically, all values for parameters defined in the Henshin rule, like attribute values or node targets, can be queried using SiLift's API. Hence, the Delta SCK as determined by SiLift can be processed in subsequent steps of S^2EC^2O.

5.3 Ontology Reasoning

In this section we discuss a further technique to investigate the SCK for the sake of determining delta information to be processed by S^2EC^2O. In contrast to a purely model-based approach, based on the *Meta Object Facility* (MOF) [Obj16], for example, ontologies have the advantage that approaches exist that infer additional knowledge, using the actual knowledge base as starting point. A *Reasoner* investigates the knowledge, can make assumptions and, besides the pure structural information such as type assertions and individual relations defined by object properties, make additional assertions. Reasoners are also able to check an ontology's consistency: It is examined if the modeled knowledge contradicts restrictions in the knowledge. Moreover, a reasoner is able to provide information on how a given fact was inferred, called *justifications* or *explanations*.

The research conducted for this thesis revealed an additional challenge concerning the closed world assumption and open world assumption, having impact on the features of ontology reasoners that can be used for S^2EC^2O. We will discuss related issues using distinct examples.

After that, we elaborate on available ontology reasoners. Finally, we will exemplify how reasoning can be used in S^2EC^2O to provide *Delta SCK* information based on two applications:

Firstly, we will show how an ontology can be brought in an inconsistent state by adding new knowledge, and how evidence can be generated by a reasoner. Secondly, we will show how a reasoner can infer additional knowledge, thus making implicit knowledge explicit.

5.3.1 The Challenge of Closed-World Assumption vs. Open-World Assumption

For reasoners to come into effect, the knowledge base should feature expressions like subclass expressions defining restrictions for certain classes. Ontologies make an assumption regarding the elements modeled and also the ones actually not part of the knowledge base. This base assumption of ontologies is called *open world assumption* (OWA). It basically says that the knowledge that is modeled in a knowledge base is not everything we know, but is just everything we are sure to know by now.

This is in contrast to typical meta-model based structures like the Meta Object Facility (MOF). MOF is the foundation of the Eclipse Modeling Framework (EMF) and UML. These base on the closed world assumption, meaning that what is currently defined in a model is exactly what is known currently.

We demonstrate effects the open world assumption (OWA) has using an example. Imagine two classes *Car* and *Human*, as well as an object property *owns*. Then, assume that a number of *Car*-individuals exists and also a number of *Human*s exist, and some humans have relations to some cars.

Now, we want to build a class *CarlessHuman* with all humans not owning a car in it. Using Manchester Syntax, we could define:

```
CarlessHuman ≡ Human and not (owns Car)
```

This is syntactically correct and also semantically sound. It can be modeled in Protégé and does not lead to an error when using reasoners, but a reasoner will never populate the class *CarlessHuman*. The reason for this is the open world assumption. If, for example, the individual *Sam* has no *owns* relation to a car, it will still not be added to the class by a reasoner because it is possible that Sam in fact owns a car but we do not know this yet.

Thus, if a reasoner would add Sam to the *CarlessHuman* class now, there is the risk that when, at a later point in time, the fact *Sam owns Citroen* can be added, this would lead to an inconsistent knowledge base because of contradicting knowledge.

Enforceability of Cardinality Restrictions

Another challenge emerging with the open world assumption comes with cardinality restrictions. As with negation in class expressions, cardinality restrictions can also not be used as expected, if they require to make statements about knowledge that is not explicitly modeled yet and could be invalidated by adding additional facts.

Regarding the example, imagine that we want all humans to own exactly 1 car. We could define:

```
CarRestriction ≡ Human and owns exactly 1 Car
```

Now, given a human *Sam* not owning a car would not lead to an error with the open world assumption in mind. The reason is that, again, this would require the reasoner to make an assumption about knowledge that is not explicitly modeled. Precisely spoken, the fact that Sam does not own at this point in time does not mean

that he will not own a car in the future, or he actually owns a car but we are not aware of it, yet.

Missing Unique Name Assumption Constraints

Mostly, ontologies, and reasoners respectively, do not work with the unique name assumption (UNA). As the name suggests, the assumption is that two different entities of an ontology having different names, do not necessarily have to be different individuals. When a reasoner is run to infer its knowledge base, it is also *allowed* to infer, for example, equality of classes or individuals when it seems reasonable. Regarding the example we introduced above, imagine a cardinality restriction and facts as follows:

```
CarRestriction ≡ Human and owns max 1 Car
Sam owns Citroen
Sam owns Mercedes
```

One would expect the knowledge base to be inconsistent now. But, in case the unique name assumption is not respected, the reasoner is allowed to infer that two individuals having different names do not necessarily describe different individuals. In this example, the reasoner infers:

```
Citroen sameAs Mercedes
```

The reasoner thus avoids inconsistency of the knowledge base. To prevent this, one could use `owl:differentFrom` to explicitly state that a given individual (or class, for instance) is not equivalent to another one. This in turn may be unwanted because it bloats the knowledge base and also makes knowledge modeling cumbersome because when adding a new fact to the knowledge base, one also has to consider to add these information.

The challenges we identified above impede design of the SCK so that reasoners can be used to infer knowledge. In what follows, we elaborate on approaches to encounter this.

Narrowing Degrees of Freedom to Increase Expressivity

To solve these issues, one possibility would be to prefer the closed world assumption over open world assumption, but this would ultimately turn an ontology in some kind of model with a rigid structure.

The ontology research community approaches this challenge with the concept of *local closed world (LCW) (reasoning)*. The idea is that delimited parts of the knowledge base can be *tagged* in a way so that facts are *surely known to be*. There is various existing work tackling this challenge.

Grimm et al. [GMP06] identified roughly the same challenges as we elaborate on in this section. They bring in an example of an ontology to determine flight services between different cities. A reasoner has the task of answering queries over the knowledge base and determine, if, given two cities, a connecting flight service is possible. Using conventional (i.e. OWA-)reasoning, an *unexpected* behavior is that the reasoner for instance infers Berlin to be the same individual as London so that a requested flight service can be inferred.

Grimm et al. introduce the **K**-operator the can be used to restrict reasoning in a way that concepts can be tagged so that they belong to each possible world [GMP06].

This can be basically read as: **K**-tagged elements have to be as specified in every possible reasoners' interpretation.

The authors describe their approach formally and also started work on a reasoner supporting it. However, the linked implementation is not available anymore and further work has not been carried out.

The authors of the Pellet reasoner also refer to the **K**-operator and state that support for queries using it is future work [PS04].

Anees Ul Mehdi [Ul 14] showed *Epistemic Reasoning* using the **K**-operator in his PhD thesis. However, the proposed approach is only able to be used through queries. Queries must be actively and concretely issued. The draw back is that queries somehow require that one needs to know what possible answers could be like in advance. Moreover, the implementation is not available nor maintained either.

There is another reasoner with support for local closed world reasoning called TrOWL [RPZ10]. A *NBox* (with N for negation) is introduced to allow *negation as failure* for all individuals that are part of the NBox. Technically this is solved using OWL annotations. At the time of writing, the TrOWL reasoner is not actively maintained. Tests in accessing local closed world using an old version of Protégé compatible with the last TrOWL release did not succeed. The search for documentation on how to use NBox reasoning in practice also yielded only sparse results.

The graph framework Apache Jena also comes with an inference API and an OWL reasoner [Thea]. At the time of writing, the documentation mentions that even the ontology support is limited. For example, cardinality restrictions only support 0 and 1 as numbers. Especially, `owl:complementOf` is not supported. Thus, support for local closed world reasoning could not be checked as support for OWL constructs even in modeling is insufficient.

At the time of writing, to the best of the author's knowledge, no working, complete implementation of local closed world (LCW) reasoning in the field of OWL exists. Nevertheless, as soon as such a reasoner exists, it can be leveraged to be used in S^2EC^2O.

In the following, we will elaborate on reasoners available to be used in S^2EC^2O. After that, we will introduce two methods of how the SCK can be enriched with subclass expressions so that a reasoner can be used to infer additional knowledge and thus create information to be used as *Delta SCK* in the delta handling of S^2EC^2O.

5.3.2 Choosing a Reasoner

As we argued in Section 4.2.1, it is meaningful for S^2EC^2O to be flexible regarding the methods the Security Context Knowledge can be manipulated. Moreover, Protégé is available as a tool for the security expert to use. The implementation of S^2EC^2O (see Chapter 9) however is built as a separate tool and thus access on API level is needed. Consequently, requirements for choosing a reasoner were the following:

1. Active development
   As we discussed in the preceding section, reasoners or their features may be described and implementations may be advertised, but it is also important that a reasoner is actively supported, so that it is also compatible with recent frameworks.

2. Available both as part of an ontology workbench as well as standalone/via an API
   For the tool support of S^2EC^2O to be developed, it seems reasonable that the user should not have the need to interact directly and manually with the reasoner. Thus, API support is needed. Apart from that, support as part of an ontology workbench, like Protégé, is needed so that the user can, also during maintenance operations, tell if the ontology is consistent.

3. Support for explanations
   Especially when the SCK grows in size, it is necessary that, in case of an inconsistency is detected, not only the mere information of inconsistency is given, but the reasoner also is able to provide the user (or S^2EC^2O connected via an API) its rationale.

We investigated a number of surveys about reasoners [Abb12, PMG+16, MLH+15]. According to Abburu [Abb12], the Pellet reasoner [PS04] is one of the reasoners matching all requirements. With regard to the additional facts that it is implemented in Java, and available as open source, it is chosen as the best match. To be precise, Pellet became a commercial product in the meantime, but there is an open source version available and actively maintained called *Openllet* [Gal].

5.3.3 Inconsistency with Explanation

In this section we show how the SCK can be enriched so that a reasoner can be employed to detect contradiction in the knowledge base.

For instance, this can happen when changes to the SCK are made to incorporate changes of the system context. A reasoner can then be used to infer that information already modeled (for instance, system-specific information) now leads to inconsistencies of the knowledge base. This fact in turn can be leveraged by S^2EC^2O's delta handling process to eventually co-evolve the system model.

Figure 5.7 shows an excerpt of a layered ontology. We consider a medical information system. This, for instance, provides an `Office Visit` component, incorporating the possibility to view a patient's exercise history. Apart from that, a component for lab procedures features, for example, the possibility to view blood examination data. When the system is initially designed, it is compliant with the laws that not only *Dana*, a doctor is able to access both views but also lab technicians like *Louis*.

We assume a law change that restricts access to data like the exercise view to doctors. We will present how to enrich the ontology so that delta information can be inferred using a reasoner.

To support checking if the ontology is in an inconsistent state which can be investigated, it is necessary to first specify some kind of constraints the ontology should comply with.

In OWL, this mechanism is called restriction (defined by the OWL XML tag `owl:Restriction`). OWL supports definition of restrictions in various ways [OWL09], in this thesis we focus on the value and cardinality restrictions.

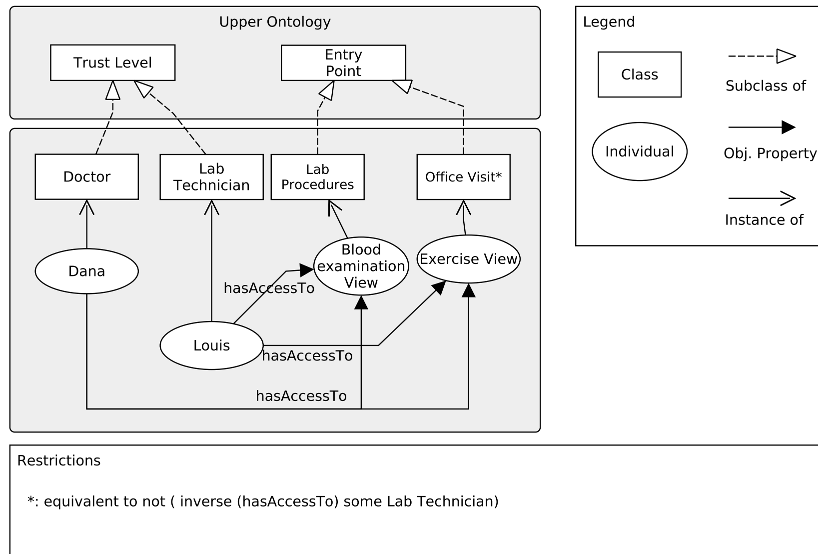


FIGURE 5.7: Example of an ontology modeling access restriction

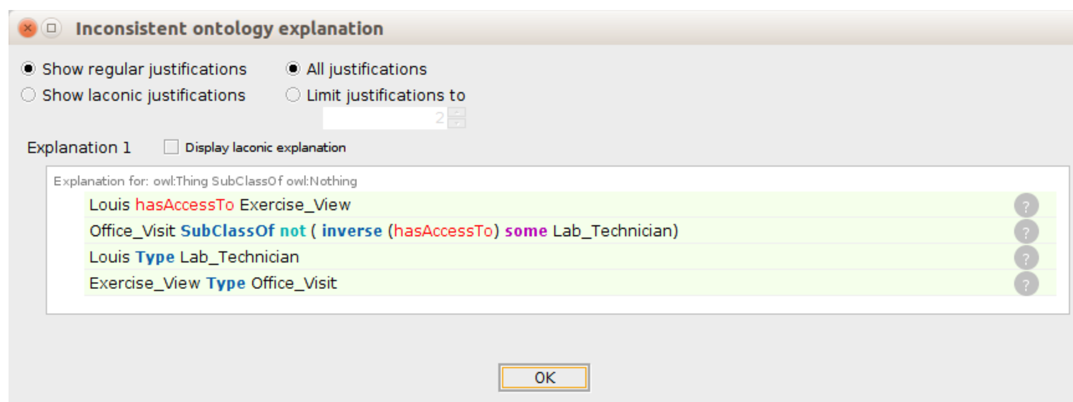


FIGURE 5.8: Screenshot of Protégé explaining the ontology inconsistency (Due to OWL, spaces within individual names are displayed as underlines.)

In the rectangle at the bottom of Figure 5.7, we show a restriction added to `Office Visit`:

```
equivalent to not (inverse (hasAccessTo) some Lab)
```

It specifies that there should not be an individual of this class that has a `hasAccessTo` object property pointing to it which has an individual as source that has the type `Lab Technician`. In other words: The restriction demands that no lab technician shall have access to `Office Visit` entry points. To specify this restriction, we use the object property *accessibleBy* defined in the security upper ontology and use its inverse, called *hasAccessTo*. Now, when a reasoner is started, it determines the inconsistency and is able to produce a justification for this claim.

Figure 5.8 shows a screen shot of Protégé after running the reasoner and calling for justification. As marked in red, the problematic fact (*Louis hasAccessTo Exercise View*) and the conflicting restriction are shown. Louis has access to *Exercise View* that is of type *Office Visit*, but for this class it is modeled that there must not exist an

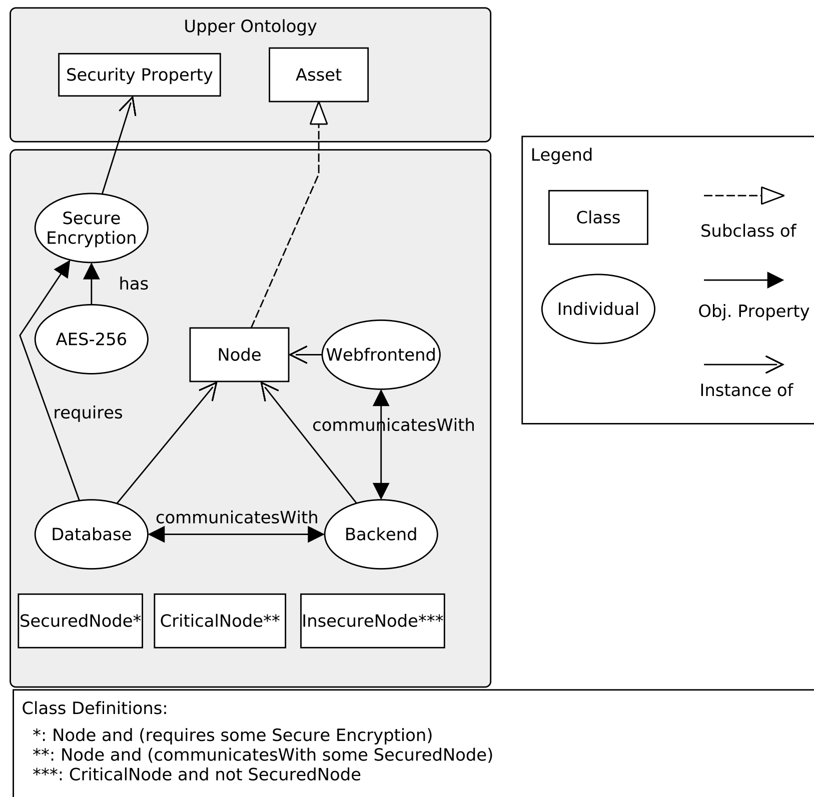


FIGURE 5.9: Example of ontology used to infer knowledge

individual of type *Lab Technician* having the *hasAccessTo* relation to an individual of type *Office Visit*.

5.3.4 Inference of Ontology Elements

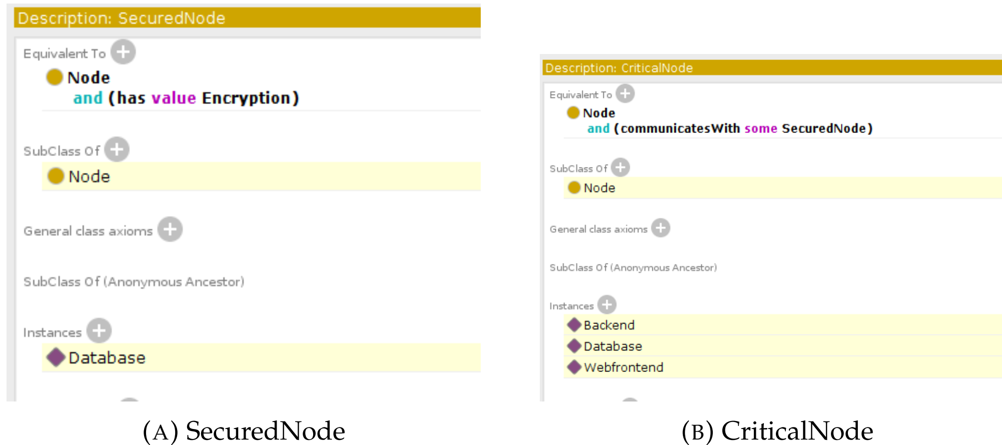
In this section we show how the SCK can be enriched so that a reasoner can be employed to infer members of classes, thus making implicit knowledge explicit. In other words, a reasoner investigates the knowledge modeled in the SCK, starting with simple facts and infers the whole knowledge base, eventually inferring knowledge that has not been stated explicitly so far.

Figure 5.9 shows an example of a layered ontology for a distributed, web-based information system. We take up again the example used throughout the preceding chapter regarding communication encryption between distributed nodes. The figure shows a distributed system in which three nodes *Webfrontend*, *Backend* and *Database* communicate with each other. Note that the web front-end does not communicate with the database node directly.

The modeling as shown in the figure can easily be modeled using a UML deployment diagram. The UMLsec stereotype «secure links» is also defined to support modeling security requirements regarding this scenario. Using «SCK» and a transformation step, the modeling of this distributed system can be generated from a UMLsec-enhanced deployment diagram.

In this example, we use the object property `communicatesWith` as an object property being both transitive as well as symmetric.

We want to use the reasoner to ensure encryption requirements for communication paths to ensure encrypted communication paths regarding all involved nodes.



(A) SecuredNode (B) CriticalNode

FIGURE 5.10: Protégé screenshots showing inferred knowledge

In this case, the *Database* node requires a secure encryption. It also communicates with other nodes, in this case *Backend* and, implicitly, with *Webfrontend*. Note that the only modeled encryption requirement is regarding the *Database* node.

We enrich the SCK by providing three classes with class definitions that can be used by the reasoner to infer the respective knowledge.

The three classes shown at the bottom of the lower layer ontology do not have any relation to the existing elements in the ontology, but they are class definitions which can be processed by the reasoner to infer additional knowledge. Thus, they can be used independent of a specific system. In this case, the meaning of the class definitions is as follows:

- SecuredNode: all nodes that have relation to a *Secure Encryption*
- CriticalNode: all nodes that communicate with a *SecuredNode*
- InsecureNode: all *CriticalNodes* that are no SecuredNodes

Figure 5.10 shows the result after executing the reasoner. As the *communicatesWith* object property is transitive, the reasoner is able to infer that *Webfrontend* can also communicate with the database.

In the figure, knowledge inferred by the reasoner is shown with a yellow background. In detail, Figure 5.10a shows the result for *SecuredNode*. *Database* has not been modeled with any class assertion or other relationships to *SecuredNode*, but, according to the class definition, the reasoner has been able to infer that the database node must have the type *SecuredNode*. Figure 5.10b shows the reasoning result for *CriticalNode*. None of the nodes have been modeled with a type relationship to *CriticalNode*, but as all nodes communicate with a *SecuredNode*, all nodes also are *CriticalNodes*.

Regarding determination of the *InsecureNode*s, which are critical but not secured, the respective class definition is shown in Figure 5.9. To reason the instances of this class, it requires availability of reasoning based on the closed world assumption as we argued in Section 5.3.1.

As a workaround, the sets of individuals of *SecuredNode* and *CriticalNode* just need to be subtracted, which can be done externally, for example by supplementary code.

As a consequence, with this example we showed how a reasoner can be employed to make implicit knowledge explicit. Ontology individuals modeling a

system deployment can be generated from a UMLsec deployment diagram. The system-independent classes *SecuredNode*, *CriticalNode*, and *InsecureNode* can be specified in a system-independent way. After a reasoner has run, the instances for these classes can be provided as delta information to support S^2EC^2O delta handling phase.

5.4 Related Work

In the related work section of the preceding chapter, Section 4.5, we already elaborated on various work in the field of security knowledge management. Thus, work shown in the other chapter's related work section may also be relevant for this chapter. While it is hard to clearly distinguish both classes, in this section, we only refer to work which explicitly also covers evolution.

Yskout et al. [YSJ12] propose a pattern-based framework focusing on co-evolution. The relationship between different artifacts of a software project is specified by a so-called pattern. Evolutions are thus captured and processed systematically. Adaptation of a corresponding software architecture is realized by analyzing a change specification to find appropriate patterns. The main contribution of this publication is the detection of common patterns at requirements level (change scenarios) in concrete change descriptions. Every change scenario is connected to concrete co-evolutions at architecture level and forms one pattern. The approach requires a system model which is complete and formalized. Concrete change scenarios need to be supplied as well as corresponding co-evolutions.

In contrast to this work, S^2EC^2O is built upon, but not limited to UML as modeling language and UMLsec as extension to model security requirements. UML supports a broad bandwidth of abstraction levels. Supporting developers who are not fully into every detail of security details is one of the core goals of SecVolution project, to which S^2EC^2O is related.

Ernst et al. [EBM11] identified changes in requirements specification as triggering event for software evolution. The relationship between requirements and the implementation is described formally. Together with goals derived from software specification, implementation tasks are computed using a constraint solver to reach the goals in accordance with the requirements. The formalism for requirements is highly abstract and the user is needed to have profound knowledge in using Techne, the logic-based requirements language which has been developed by the authors. The goal of this rather abstract approach is to provide a clearly structured, graph-based guidance for implementation decisions. The approach puts emphasis on dependent problem parts and their requirements.

A co-evolution scenario arises when changing requirements restate the requirements problem which the authors state is not solved yet. An interface to system design level is not discussed. The whole software project needs to be modeled using Techne.

Souza et al. [SLAM13] regard requirements that cause the evolution of other requirements. Their approach is based on goal-oriented modeling. The system the stakeholders expect is supposed to be modeled as a set of goals. During run time, events can be monitored. The concept of evolution requirements (EvoReqs) based on event-condition-action schema, is used to adapt the goal model. Security is only considered as a side note. The adaptation capabilities are fully based on the rules as provided upfront and as part of the closed system, there is no mechanism of accessing knowledge like general security knowledge.

Salehie et al. [SPO+12] present a requirements driven approach to adaptive security which aims at identifying and evaluating changing assets at run time to dynamically enable different countermeasures. A causal network is build upon a goal model and a threat model to analyze system security in different situations. The causal network needs to be maintained manually and dealing with unanticipated events is not covered.

Stefan Farfeleder proposes a framework for requirements engineering focusing on embedded systems in his PhD thesis [Far12]. The focus is especially shown by covering failure types and failure rates of components in embedded systems. The framework features a modularized knowledge base given by (domain) ontologies, thus also supporting reusability. Requirements are defined using the so-called Boilerplates notation. Knowledge of the ontology is used to support definition of requirements, analyses to check requirements regarding completeness, consistency, and ambiguity. The knowledge base is compiled by using the import mechanism and importing smaller ontologies covering components of a system. A core taxonomy is presented which can be compared to the concept of the security upper ontology used in S^2EC^2O. The framework supports context change for example by, in some cases, automatically updating the ontology structure when requirements change. Additionally, comparing two versions of a domain ontology for review is supported.

Chapter 6

Co-Evolve Design-Time Models by Assessing Context Evolution

The preceding chapter focused on detecting an evolution of the system's context, represented by the Security Context Knowledge. As soon as changes to the SCK have been detected, the impact on the system under consideration needs to be assessed. In case the system is affected by the given context evolutions, appropriate co-evolutions need to be determined and applied, so that the system is compliant with its security requirements again.

This chapter shows how a software system can be designed so that it is secure at its initial deployment. We will further introduce an extended notion of security requirements to better support, for example, technical progress. We will specify a catalog of rules to coordinate assessing context evolutions and co-evolution actions. We will also show an example of a catalog entry. Finally, we will present how to co-evolve a system at design time semi-automatically.

This chapter thus contributes to the following research questions:

RQ2: *How can rules be formalized that are able to preserve the system's security given knowledge evolution?*

RQ3: *How can these rules be used to carry out a semi-automatic co-evolution given a context evolution?*

Section 6.1 elaborates on the relation of context evolution and co-evolution of the system under consideration. It illustrates which part of the S^2EC^2O approach is covered in this chapter, how it relates to the overall approach, and how the respective components interact.

S^2EC^2O needs to be aware of the security requirements the system under consideration shall be compliant with. To solve this, Section 6.2 introduces the concept of *Essential Security Requirement* (ESR) to decouple basic security needs of a system and their concrete technical realization. In that section, we also elaborate on the question of how a system is made initially compliant with its security requirements using S^2EC^2O, prior to evolution. This is especially needed as a baseline to perform evolution-based security analyses.

Whenever the Security Context Knowledge evolves, this is considered as *Delta SCK* that needs to be analyzed. In S^2EC^2O, various sources of context evolution are supported. Besides evolution of the SCK as covered by Chapter 5, monitor findings from the run-time monitoring are also supported (as we will discuss in Chapter 7). To solve the challenge of variety and ease analysis, this delta information needs to be structured. Thus, Section 6.3 introduces a model to appropriately specify the delta information and supporting processing for co-evolution.

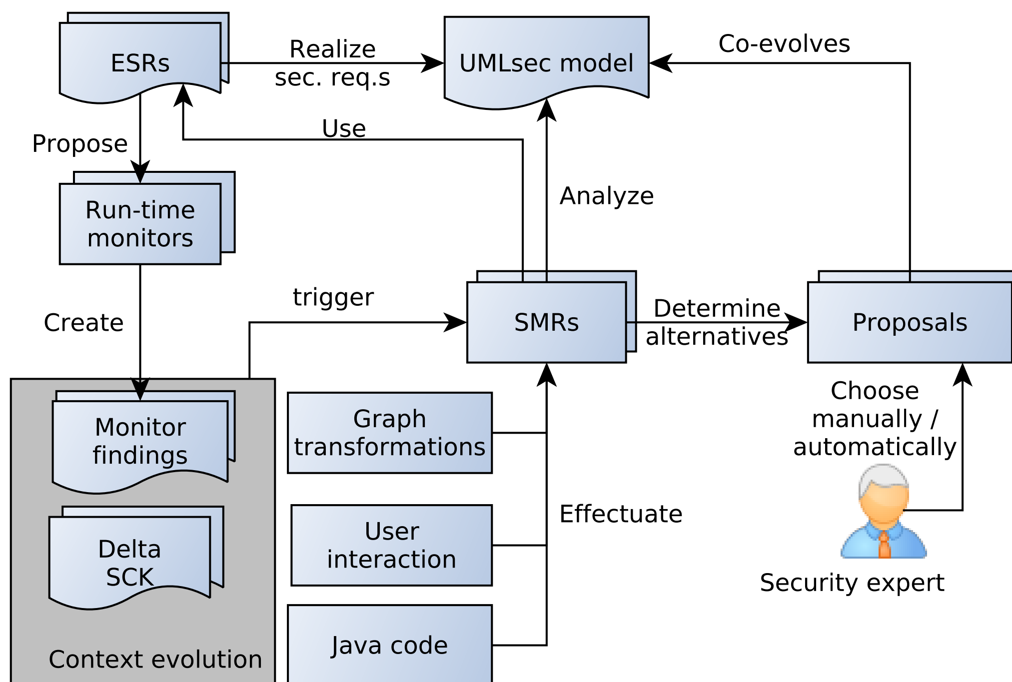


FIGURE 6.1: Concepts used in the co-evolution approach and their relation

As soon as the context evolution has been detected, appropriate co-evolution operations need to be determined. Section 6.4 introduces the concept of Security Maintenance Rules (SMRs). These rules are triggered by context evolution events and, based on these, infer several co-evolution alternatives.

Finally, Section 6.5 introduces an algorithm for design-time co-evolution on context evolution. The algorithm contributes to the S^{2}EC^{2}O process. Section 6.6 concludes the chapter with related research. We will discuss this chapter's contribution to the research questions in Section 11.2.

6.1 Leveraging Context Evolution for System Co-Evolution

In this section, we focus on the part of S^{2}EC^{2}O responsible for the design time co-evolution. Figure 6.1 shows an overview of S^{2}EC^{2}O's components that make use of context information and realize the system co-evolution.

In the upper center of the figure, the system under consideration is shown as *UMLsec model*. Using Essential Security Requirements (ESRs) and Security Context Knowledge (SCK), an ordinary system design can be extended with security annotations. Essential Security Requirements also contain information on how security properties can be monitored during run time (*Run-time monitors*). As soon as a system is built according to S^{2}EC^{2}O and is in production, context evolution information can be collected. This information can than be used to investigate it and, with regard to the system, react to it.

In the lower left of the figure, potential sources for context evolutions are depicted. Firstly, there is delta information regarding the SCK as discussed in Chapter 5 (*Delta SCK*). Second, there are *monitor findings* coming from the run-time phase.

S^2EC^2O supports monitoring security properties at run time. The run-time phase of S^2EC^2O is covered in the Chapters 7 and 8.

The context evolutions (*Monitor findings* and *Delta SCK*) then trigger Security Maintenance Rules (SMRs). These rules cause the co-evolution of a system. To accomplish this, SMRs can determine alternatives for co-evolutions, collected in *proposals*. To realize this, they can access all relevant data sources. First of all, queries of the system (model) can be conducted. The context evolution information can also be investigated, since it may feature references to the SCK or the system model. As the Essential Security Requirements also contain information on how the security properties are to be realized in general, these can also be used by SMRs to infer co-evolutions. Regarding methodologies to analyze the system and also to realize co-evolutions, S^2EC^2O currently supports but is not limited to the following: graph transformations, user interaction (i.e. let the user provide system-specific information), and also Java code, for example to manipulate models via a reflective API.

Finally, the proposals composed by SMRs containing co-evolution alternatives, need to be inspected. This can be done by the security expert manually and also in conjunction with S^2EC^2O semi-automatically.

Thus, context evolution information raised by external sources triggers Security Maintenance Rules that analyze the system, to verify, if it needs to be co-evolved. If a co-evolution is necessary, appropriate steps are selected semi-automatically.

6.2 Initial Compliance to Security Properties

To accomplish the overall goal of S^2EC^2O, preserving a system's security by leveraging context knowledge, we need to make sure that the system is compliant with its security requirements at the beginning as well as that it is enhanced by annotations to enable S^2EC^2O. Using a system that is known to be secure initially as a *baseline*, it is possible to assess incoming context changes. This is one of the tasks covered by the Essential Security Requirements (ESRs) which we introduce in the following.

The security expert has to express the basic security needs of the system, so that obligations for the system design can be inferred. We show how a given software system is fitted into S^2EC^2O initially. The process that needs to be run through is S^2EC^2O's initialization process shown in Section 2.2.3 (on page 15).

6.2.1 Security Context Catalog Meta Model

To support the management of common security knowledge, S^2EC^2O provides the SCK we introduced in Chapter 4.1. The counterpart which, when used in conjunction with the SCK results in an ordinary security requirement again, is introduced in this section and called *Essential Security Requirement* (ESR). S^2EC^2O decouples security requirements into two parts. Security requirements are split into the bare security requirement and all information relevant for a concrete implementation. This supports building a software system that fulfills its security requirements at initial design, i.e. before the context may evolve. The motivation for this split is the underlying perception that in many cases, the context, affecting the correct way of realizing a security requirement is more often and quickly subject to change than the fact that a security requirement change occurs [RGB+14a, RGB+14b].

ESRs are independent of concrete technologies and algorithms; their actual implementation depends on the given domain and environment. For example, when we take *secure encryption* as an *essential* requirement, it needs different algorithms

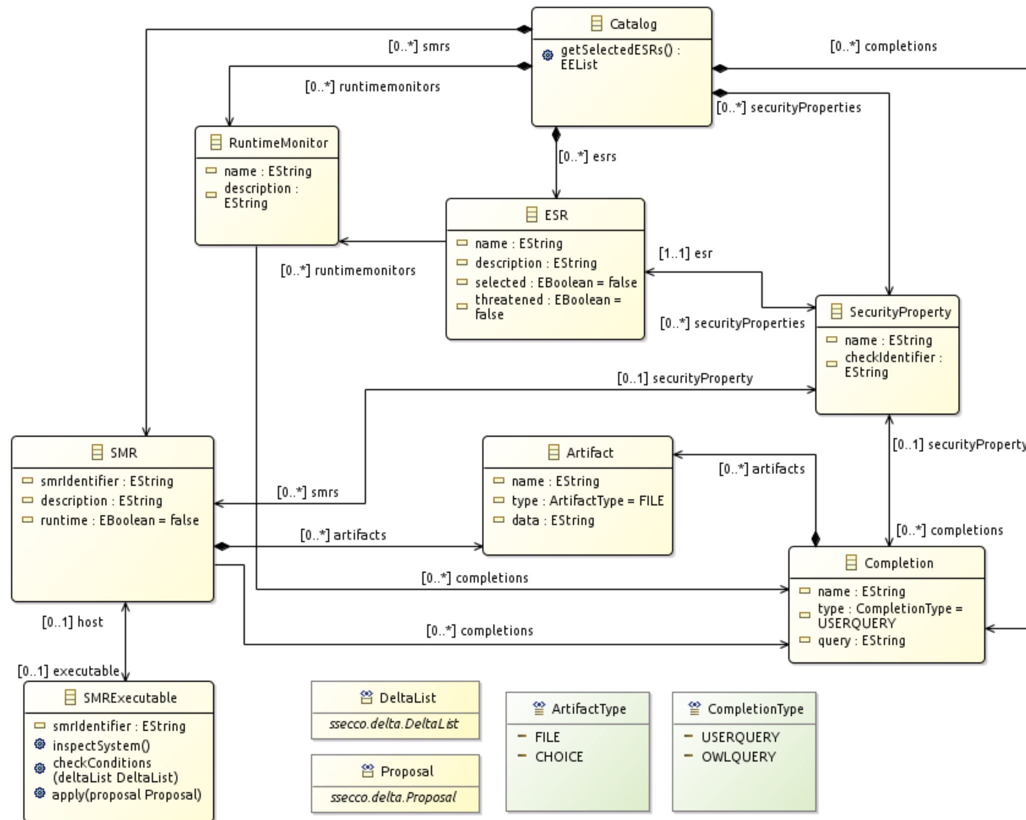


FIGURE 6.2: Meta model of the Security Context Catalog

and parameters when applied to an ordinary web application as when used in the banking sector. To implement a system that fulfills given ESRs, extensive knowledge about current technologies, security principles, laws, and many others may be involved.

According to S²EC²O's overall structure shown in Figure 2.3 on page 12, Essential Security Requirements have a central role in the approach. Beginning with an Essential Security Requirement, the Security Context Knowledge (SCK) is queried to reify security requirements. Furthermore, they propose monitoring to check if given security requirements hold during run time. The system can be co-evolved by triggering SMRs coming from ESRs.

We define Essential Security Requirements as part of a meta model. S²EC²O provides a number of Essential Security Requirements to choose from, called *Security Context Catalog*. The Security Context Catalog thus embeds ESRs. Figure 6.2 shows the catalog's meta model.

The meta model is specified as an Ecore model and used in the prototypical implementation of S²EC²O (see Chapter 9). The `Catalog` element has relations to all other element types which, for instance, makes navigation easier. Biermann et al. call meta models of this kind *rooted*:

> "An EMF instance model is called rooted if there is one container which contains all other elements transitively. Although EMF instance models do not need to be rooted in general, this property is important for storing them, or more general, to define the model's extent." [BET12]

In what follows, we walk through the meta model's classes one by one and outline them.

ESR An Essential Security Requirement has a name and a description. The name is a concise, short name to identify the Essential Security Requirement. The description is used to provide the human readable explanation of the Essential Security Requirement, as well as assumptions regarding the modeling as part of the SCK and system model for the underlying checks and Security Maintenance Rules to work properly. The `selected` attribute is used to mark the Essential Security Requirements which have been marked as relevant by the user. The `threatened` attribute is used to indicate that the respective ESR is currently threatened because a context evolution has taken place that invalidates the security properties as obliged by the ESR. An ESR element has associations to run-time monitor elements to specify relevant monitoring mechanisms to be used.

SecurityProperty An Essential Security Requirement proposes a number of security properties the system design has to be compliant with. Each of them is identified by a name. S^2EC^2O currently focuses on, but is not limited to security properties that can be checked using the UMLsec approach and accompanying tools. We discuss this in detail in Section 6.2.3. As in [Gär16], we use the term *security property* for one or more security requirements an asset needs to fulfill.

The `checkIdentifier` property is used to associate, for example, certain UMLsec security requirements. A security property can associate a number of SMRs which can be relevant to co-evolve a system, when the respective security property is not valid anymore. For security requirement checks regarding UMLsec, the prefix `UMLsec` is used, followed by `::` as delimiter. In case that checks regarding the SCK use OWL technologies, the prefix `OWL` is used. With statement logic, combination and/or exclusion of security property identifiers is possible. An example is as follows:

```
checkIdentifier=UMLsec::securelinks AND OWL::Reasoner
```

RuntimeMonitor To also cover the run time, an Essential Security Requirement additionally is linked to a number of run-time monitors. These provide the possibility to monitor an ESR during run time. Run-time monitors, like ESRs themselves, are identified by a short name and a human-readable description.

The elements discussed so far are used to let a user express the system's security needs and how they need to be integrated into the system design to provide the desired degree of security. Apart from that, run-time monitors are used to check compliance of security properties during run time. However, as soon as a violation is discovered by a monitor finding or an evolution of the SCK, the system needs to be co-evolved. This part is handled by Security Maintenance Rules (SMRs).

SMR A Security Maintenance Rule is specified according to the *Event-Condition-Action* schema [Day94] and consists of three parts: the *ON* part is the event triggered by a SCK evolution or monitor finding. The *IF* part is used to define how to evaluate if the current trigger is relevant for the given rule (i.e. certain *conditions* hold). In the *DO* part, actions needed to co-evolve the system are defined. A detailed discussion of SMRs follows in Section 6.4.

The `runtime` property is used to indicate if the given SMR carries out operations at run time. Using this, S^2EC^2O is able to distinguish which co-evolution operations are relevant for design- and run-time. The `SMRExecutable` class also belongs

to the SMR and is used to associate executable instances of SMRs. We introduce `SMRExecutable` for technical reasons, which we will discuss in Section 9.1.4 on 114.

Completion As we argued above, the Security Context Catalog is technology- and system-independent. However, to come into effect, system-specific data is inevitable. For example, when a system is required to be secured by encryption, a concrete encryption algorithm needs to be chosen from the context knowledge (SCK). To specify what is needed to *complete* elements of the Security Context Catalog to be usable for a concrete system, `Completion` elements can be used.

Not only security properties may need additional knowledge so that they can be implemented correctly. The same applies to run-time monitoring components as well. Not only knowledge from the SCK may be necessary (`OWLQUERY`) but also data provided by the user (`USERQUERY`). For example, a run-time monitor is able to monitor the configuration of an application server. The user then needs to provide the file path to the configuration files to be observed.

The Completion elements are used to realize this information demand.

Artifact A given completion may need more than one such completing elements (in the simplest manner: a number of files to be monitored rather than a single file). Thus, a completion can be connected to a number of artifacts.

Both `Artifact` and `Completion` are used by the major components to reference details and system specific artifacts, as shown by the associations in Figure 6.2. Besides `FILE`, the artifact type `CHOICE` is also supported to define completion elements the user has to choose from.

To conclude, the Security Context Catalog features and links the necessary elements to accomplish the following goals:

- Define basic security needs,
- select the security needs for the specific system,
- query the context knowledge to gather current technology information and further relevant knowledge, and
- query the user to gather system specific knowledge or to make choices if the information gathered from the SCK is ambiguous.

In the succeeding section, we demonstrate the workings of ESRs using the example of a concrete ESR for secure communication.

6.2.2 Example of the S^2EC^2O Initialization Process

We demonstrate how a system can be designed as *S^2EC^2O aware* by using one ESR from the catalog as a starting point. We come back to the example of ensuring encrypted communication we already used in the previous chapters.

The ESR *Secure Communication* has the goal to make the communication between nodes in distributed systems secure. To accomplish this, at design time, the UMLsec stereotype «secure links» is recommended.

Figure 6.3 shows an example of this Essential Security Requirement in the notation of a UML object diagram. We omit object names and values of description attributes for better readability.

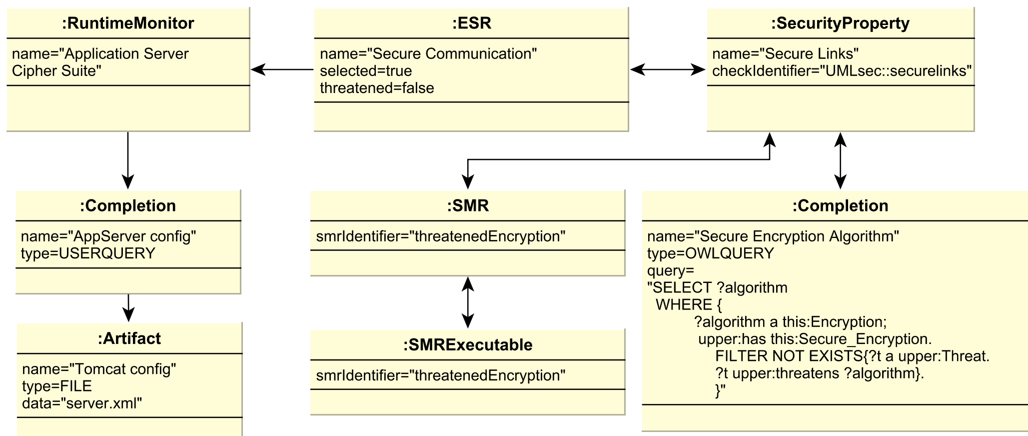


FIGURE 6.3: Example of an Essential Security Requirement

The system under consideration shall be compliant with this ESR, so the attribute `selected` is set to true. No context evolution is known at this point of time that threatens this ESR, so `threatened` is set to false.

The ESR is connected to a security property enforcing the UMLsec check for «secure links» using `checkIdentifier`. In detail, a tool for static checks independent of evolution can be provisioned with checks to run. We give details on how to check the system's security prior to evolution in Section 6.2.3.

The completion object is used to query the SCK for an encryption algorithm for which currently no threat is known. This is done using a SPARQL query. We introduced a similar query in Section 4.4 on page 40.

To support run-time monitoring, the ESR is connected to a `RuntimeMonitor` object to monitor application server configuration.

For example, distributed systems can be realized using an application server. In this case, a run-time monitoring can consider the configuration of the server, thus the offered cypher suites. In the example, this is the case. A Tomcat application server is queried and the run-time monitoring of S^{2}EC^{2}O can be provisioned to monitor `server.xml`.

The security property is connected to one SMR. The Security Maintenance Rule `threatenedEncryption` is associated to the security property as shown above and connected to a `SMRExecutable` with the same identifier. This doubling is due to technical reasons and discussed in detail in Section 9.1.4 (on page 114).

This Essential Security Requirement is used in the S^{2}EC^{2}O initialization process defined in Section 2.2.3 (on page 15) as follows: Once the user chooses it from the Security Context Catalog, it is used by S^{2}EC^{2}O to guide the user in how the system design needs to be altered to meet the desired security requirement. In this case, the SCK can be queried without further interaction with the user. Monitoring a configuration file does not make additional coding necessary but the file's location needs to be provided by the user so the monitoring can be instantiated.

6.2.3 Check System's Security prior to Context Evolution

After the given system design has been adapted according to the obligations given by S^{2}EC^{2}O, it needs to be ensured that the system meets its security requirements prior to evolution. S^{2}EC^{2}O is focused on security vulnerabilities introduced by

evolutions, so we assume that the system is compliant with the security requirements initially. This can be achieved using existing tooling for security checks at design time. Currently, S^2EC^2O refers to the UMLsec approach for secure systems design as introduced in Section 3.2. UMLsec is supported by the CARiSMA tool platform [APRJ17].

6.3 Coordinate Context Evolutions

In the preceding sections, we discussed how to build a system that is designed in a way that it fulfills all security requirements at its initial design on the one hand and is build in a way to enable the S^2EC^2O approach on the other hand.

We introduced various sources and types of context evolution, which can be summed up as a *Delta to the Security Context Knowledge*, in Chapter 5. As the S^2EC^2O delta process shows in Section 2.2.3 (on page 17), an additional source of context evolutions are findings of the run-time monitoring, leading to a change of the security knowledge. Monitoring a system's security compliance at run-time is discussed in Chapter 7.

In Section 5.1 we argued that an arbitrary number of differences in the SCK can occur at the same time. Obviously, there may be cases in which only a combination of certain deltas is relevant to a given Security Maintenance Rule.

Context evolutions need to be processed to determine possible co-evolutions. Thus, it is necessary to have the deltas in a formalized manner, and in a structure that supports coordination of possible co-evolution operations.

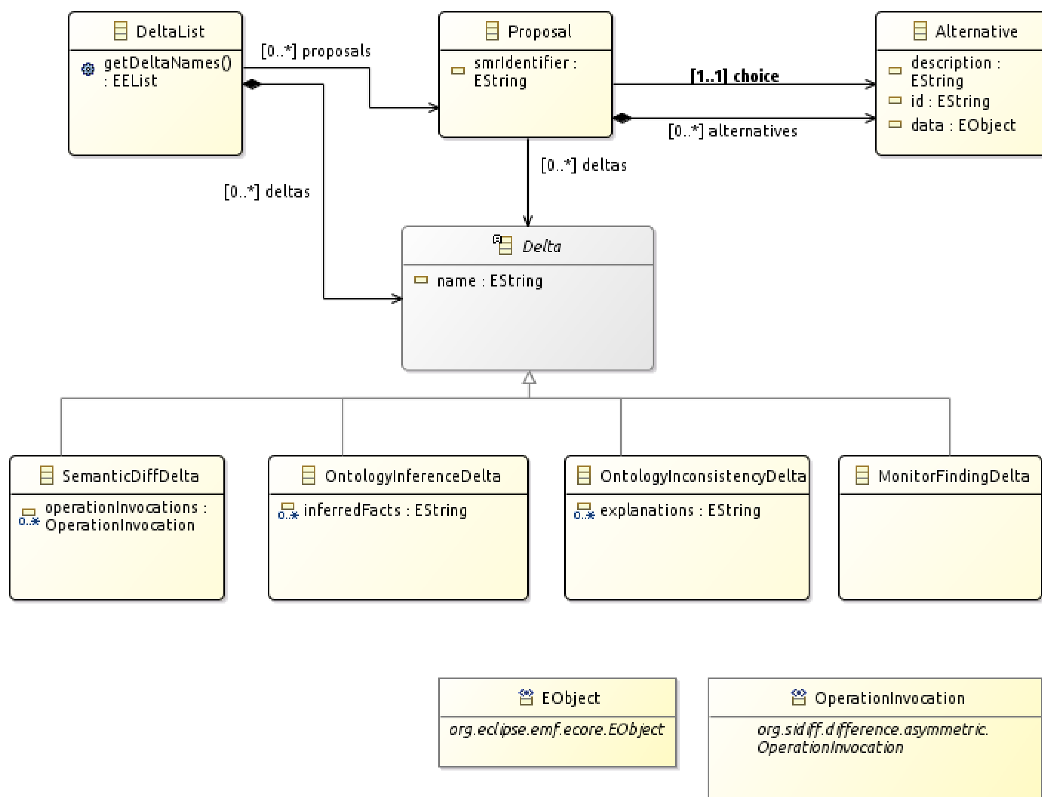


FIGURE 6.4: Meta model of delta information in S^2EC^2O

To solve this issue, we introduce a delta information model and specify it by its meta model, as Figure 6.4 shows. We introduce the meta model by discussing all of its major components.

Delta This abstract class is used to differentiate between the various possible delta types. The meta model currently supports but is not limited to:

1. Semantic diffs as discovered by SiLift regarding two versions of the SCK,
2. facts inferred from a reasoner, and
3. inconsistency and explanations for the current SCK version that come from reasoning contradictory facts.

Additionally, there is also a class prepared to model a finding as discovered by a run-time monitor.

DeltaList This class is used to collect a number of deltas that have been discovered since the last run of S^{2}EC^{2}O's delta process, shown in Section 2.2.3 (on page 17). As stated above, it is conceivable that for certain co-evolution operations, multiple specific deltas need to be considered jointly.

Proposal Security Maintenance Rules are the components used to provide and apply co-evolution operations (for a detailed discussion see Section 6.4). Naturally, a given SMR can provide co-evolutions for a limited number of deltas. Hence, with `Proposal`, a SMR can submit proposals to be considered in S^{2}EC^{2}O automatically or by the user manually. For subsequent steps, a proposal needs to contain the SMR's name it is related to and feature associations to the set of deltas it covers. A SMR may offer different co-evolutions to react to the same evolution, i.e. alternatives. For example, a given insecure encryption algorithm can be exchanged by a number of alternative algorithms. Hence, the class `Alternative` is used to model a number of alternatives. An alternative consists of a human-readable description and an ID for internal reference. With `data`, a property is provided that can be used to store data necessary to carry out the co-evolution later on. With regard to the `choice` association, the user needs to select exactly one alternative for each proposal to guarantee that for every delta a reaction will be carried out.

6.4 Co-Evolution at Design Time

In the preceding sections, we put emphasis on how evolution of a system's context can be structured and formalized into a list of deltas and how this can be used to support structuring the process of selecting co-evolution alternatives. In this section, we introduce the concept used in S^{2}EC^{2}O to co-evolve the system under consideration, namely *Security Maintenance Rules (SMRs)* [BJR+14].

The goal of a SMR is to recover the security of a system when for a certain security property, in the end an ESR, a set of threats is discovered. To accomplish this, a SMR may analyze the system context, details of the context evolution, and the SCK to check if the SMR is applicable. Subsequently, it may propose possible co-evolutions and finally apply them, i.e. adapt artifacts of the system.

Ultimately, by bringing all SMRs as provided in the Security Context Catalog into action, they can react to endangered ESRs by determining a series of co-evolution

operations semi-automatically. The series of operations is considered to be necessary to re-establish the system's compliance with all ESRs again.

To support this combination of external events to be assessed and composing actions to be performed, Security Maintenance Rules are following the *Event-Condition-Action* principle [Day94]. This means that a rule follows this schema:

ON Event **IF** Condition **DO** Action

Event indicates the deltas of the SCK and monitor findings as discussed above which can be used to get relevant security properties and ESRs. Using the *Event*s, possible relevant Security Maintenance Rules can be selected, but deeper investigation is needed (for example run-time vs. design-time SMRs) which is defined by the second part:
The *Condition* examines if the system is in a non-compliant state. This is realized using model queries, compliance checks, etc. with respect to the (unchanged) ESRs. A query can utilize the deltas and inspect the system as well as the SCK in detail to ensure that preconditions for the SMR to be applied are met. For example, SMRs addressing information stored in state chart diagrams should only be applied to a model containing respective information.

With regard to SMRs as introduced by Figure 6.2 (on page 66), the class `SMRExecutable` provides three operations which are used to realize the Event-Condition-Action principle as follows:

- `checkConditions(deltaList DeltaList)` is used to realize the *Condition* part. By using the parameter `deltaList`, the SMR is able to access the current list of deltas and, for example, add proposals.
- `inspectSystem()` can be used to examine the given system, thus realizing the *Event* part.
- As soon as the user or an algorithm has selected a proposal, the method `apply(proposal Proposal)` is called, realizing the *Action* part. The proposal which is to be used by the SMR to co-evolve the system is submitted as a parameter.

Figure 6.2 (on page 66) shows a combination of two classes `SMR` and `SMRExecutable`. For technical reasons, we decided to split SMRs in these two classes. This solution allows to combine an Ecore-based model on the one side and provide separate implementations for every SMR as Java classes on the other side. We provide further details in Section 9.1.4.

Regarding the *Action* part of the SMRs (i.e. the `apply()` method), various types of reaction are supported by S^2EC^2O. Java code is used as the common basis for the `apply()` method. Supported approaches that can be used are depicted in Figure 6.1. The most sophisticated possibility is to directly manipulate the system model by using graph transformation rules specified using Henshin (see Section 3.6). To pre-allocate nodes of the transformation rules with concrete model elements, we make use of the *partial match* mechanism of Henshin. Thus, we only need a reduced set of simple rules which we can concrete with information gathered in preceding steps. As there are certain limitations regarding Henshin (for example regarding paths through models), direct manipulation of the system model using EMF and reflection code is also possible. With regard to the Security Context Catalog meta model we introduced in Section 6.2.1 (on page 65), SMRs can also be associated with `Completion` instances which means that the user can be requested to provide additional input to

further parameterize reactions (like defining new class names, choose desired position of new elements in model hierarchy, etc.).

6.5 Semi-automatic Co-Evolution of Models

After a context evolution has emerged, appropriate co-evolutions need to be determined, supplied with additional knowledge, and finally applied. Throughout this chapter, we introduced concepts to couple S^2EC^2O with the system development, how to check the system's security upfront, and how formalize and coordinate delta information. In this section, we present an algorithm to realize the *delta handling* that is used to put all the parts together and realize the design-time co-evolution.

```
1  loadCatalog();
2  loadModel();
3  semanticDiff();
4  executeReasoning();
5  for (Extension e : pluginRegistry.getAll("ssecco.esr.model.smr")) {
6   SMRExecutable smrExecutable pluginRegistry.createExecutable(e);
7   bindSMRExecutable(smrExecutable);
8   smrExecutable.checkConditions(deltaList);
9  }
10 requestUserChoices();
11 for (Proposal p : deltaList.getProposals()){
12  smrExecutables.get(p.getSmrIdentifier()).apply(p);
13 }
14 saveModel();
```

LISTING 6.1: Pseudo code for delta handling

Listing 6.1 presents the algorithm for delta handling in Java-like pseudo code. First of all, the Security Context Catalog and the system model are loaded (lines 1-2). After that, the context evolution information needs to be gathered, by calculating the semantic difference between the given two versions of the SCK (line 3) and by reasoning about the current SCK version (line 4). Both methods trigger the ontology differencing mechanisms we discussed in Chapter 5. Execution of these methods populates a `deltaList` as introduced in Section 6.3. Thus, all delta information can be accessed in a sophisticated, structured way. After that, a loop is initiated that instantiates and triggers all available SMRs (lines 5-9). Every SMR is first instantiated (line 6) and then bound to its respective `SMR` object of the loaded Security Context Catalog (line 7). Finally, the `checkConditions()` method is called so that every SMR can analyze the model, the SCK, determine if there are relevant deltas, and submit proposals that are subsumed by the `deltaList` (line 8).

After collecting all proposals, the user is requested to choose alternatives wherever there is more than one alternative for one proposal (line 10). To actually realize co-evolutions, a reaction of a SMR has to be *applied*. This means that the reactions are carried out at the system model. As we discussed in preceding sections, there are different kinds of reaction that can be used as a co-evolution, triggered within the `apply()` method. For all proposals that have been submitted by the SMRs (and a choice has been made), they get applied (lines 11-13). Finally, the co-evolved system model is saved (line 14).

Afterwards, it has to be checked if no ESR is endangered. If an ESR is now violated that was respected before execution of the algorithm, or if there is an ESR that still is violated, the assistance of the security expert becomes necessary. The security expert needs to investigate the ESRs, eventually manually adapt the model to prevent side effects, and consider a full security check using methods as introduced in Section 6.2.3. In particular, this becomes necessary if two security requirements

are to be applied that (logically) contradict each other (such as non-repudiation and anonymity).

6.6 Related Work

In this section, we discuss related work of this chapter. As the publications that we took into account only cover a some aspects of this chapter, we categorized them.

6.6.1 Analyze the Impact of Changes with respect to Co-Evolution

When an evolution takes place, the software artifacts under consideration can become inconsistent due to unintentional side effects.

The *Water wave phenomenon* inspired Li et al. [LZSL13] to develop an impact assessment approach based on call graphs. First they analyze the core which consists of the direct affected software artifacts. After that, the call graph is analyzed, taking the interference of different changes into account. They claim that their nature inspired approach has fewer false positives compared to common call-graph approaches. Their approach is focused on predicting how big (i.e. number of methods to change) the impact of changing a number of methods in a given source code project will be. Opposed to this, our approach aims at analyzing impact regarding security properties.

Bouneffa et al. [BA14] propose a process to support a change impact analysis for applications modeled with *Business Process Models*. Semantic knowledge about artifacts and change operations that is represented in an ontology is used to realize a change management, while the system itself is represented as a graph. The authors present graph transformations for numerous atomic as well as composite changes to support system evolution. A part of the respective RHS contains impact elements as part of the graph. The authors do not focus on how the knowledge represented by the ontology is managed and what the reaction to determined change impacts should be.

The approach does not support processing of requirements or other overall properties. The change management presented instead serves for keeping a big model consistent. Apart from that, the authors do not argue how transformation of elements already annotated with impact elements should look like.

Okubo et al. [OKY11] regard the impact of software enhancements on security by involving patterns of enhancements. The overall goal is to enable the developer to estimate and compare the amount of modifications needed by different countermeasures. The proposed security analysis process uses security requirements patterns to identify threats and security design patterns to find countermeasures (see also [OKY12]). The approach assumes evolution of ordinary requirements and then estimates possible impact on the security requirements.

In contrast to S^2EC^2O, their approach does not cover evolution. Besides, application of the approach leads to having a guidance of which patterns for security precautions or vulnerability mitigations should be applied. There is no integration with an actual system design so far. Patterns are specified by natural language. The effect of knowledge evolution to natural language or semi-formal sources is not covered.

Lehnert et al. [LFR13] investigated on the challenges that, after an evolution has taken place, development artifacts may be inconsistent to each other. To address this issue, they propose a rule-based *impact analysis* (IA) for a number of typical project artifact types (models, source code, tests). Consistency is managed by an overall

meta model. The goal of this multi-perspective analysis is to detect which artifacts are affected and how. *Impact Propagation Rules* distribute changes between the artifacts. The approach focuses on preserving consistency of different development artifacts. It does not cover security aspects, and an assumption is that the evolution is planned.

In S^2EC^2O, unplanned evolutions triggered by external changes are considered with the goal of preserving the functionality of the considered system.

Sun et al. [SLWZ13] provide an approach to support *change impact analysis* of artifacts of object-oriented programs. Programs are formalized as graphs. This especially holds for classes, their relationships, and their properties. Change sets are used to determine impact sets to provide ongoing consistency between artifacts.

The approach does not consider requirements or the requirements level and mainly focuses on structural changes between classes like changes of visibility or call dependencies.

6.6.2 Vulnerability and Attack Management

A main type of security threat arises from the constant emergence of new vulnerabilities and corresponding attacks.

An investigation of Kühn et al. [KM14] focuses on zero-day exploits, in which previously unknown vulnerabilities are exploited by attackers. Big players such as Microsoft or Facebook face a desperate situation where conventional security precautions seem to be overwhelmed by a rapidly increasing number of these exploits. However, rather than addressing the question of how vulnerabilities can be avoided upfront, their current reaction is to take part in the race by conducting bug bounty programs.

In contrast, S^2EC^2O builds upon community knowledge. Using layered ontologies, knowledge can be hierarchically structured. As soon as an attack is discovered (i.e. knowledge about it is made explicit), it can be shared publicly, which speeds up the vulnerability fixing process.

Alhazmi et al. [AMR07] define a metric called *vulnerability density*, which puts the number of vulnerabilities of a product in relation to its overall bug count. They define a logistic and a linear model and measure its fitness regarding the publicly available bug data of a number of Microsoft Windows releases and two Red Hat Linux releases. The authors state that their metric can be used to predict the number of vulnerabilities to be expected. This method could be useful for S^2EC^2O to gain additional knowledge and pointing out potential system parts which may be vulnerable.

Since S^2EC^2O focuses on the system level, there is a trade off regarding the granularity of vulnerabilities and attack types it can address. We focus on design-level vulnerabilities. Implementation-level vulnerabilities such as buffer overflows must be detected and fixed on the source-code level, for which a plethora of tools exist.

Chapter 7

Assess Security Compliance During Run Time

The approach shown in the preceding chapters provides steps which lead to a system model annotated with security requirements. ESRs offer a way of beginning secure systems modeling by expressing basic security needs independent of concrete technologies. ESRs also provide knowledge about how implemented security requirements can be monitored at run time (i.e. run-time monitors).

Thus, S^2EC^2O as presented so far, aids the user in managing a security knowledge base, calculating differences in it and reacting to evolutions of it with appropriate co-evolutions. However, in many situations, the security of a system cannot be fully checked at design time. For example, if a system's behavior highly depends on the user's input or code is loaded dynamically during run time, static checks are not sufficient. The methods presented in the preceding chapter, especially the concepts *Essential Security Requirement* (ESR) and *Security Maintenance Rule* (SMR), can be used to extend the security requirements management into the run time.

In this chapter, we close the gap by introducing components of S^2EC^2O that realize run-time monitoring. Moreover, information gathered during run time enhances design-time artifacts to support the security expert in understanding attacks and fixing possible design flaws. Thus, this chapter contributes to the following research question:

RQ4: *How can information coming from the system execution be used, to assess the quality of the security requirements compliance during run time?*

Section 7.1 introduces the run-time monitoring part of S^2EC^2O and relates it to the rest of S^2EC^2O. It presents an approach of coupling the system's model and source code together, and how security annotations of the Security Context Catalog used during design time can be monitored at run time.

Section 7.2 introduces a method of bridging the gap in secure systems engineering between UMLsec models and Java source code. We will introduce a concept of source code annotations to support mapping UMLsec annotations with their source code counterpart. Furthermore, we will propose a synchronization of security annotations at source code as well as model level. We will also show a mapping between source code and model annotations. This is exemplified using the specific UMLsec security property «secure dependency».

In Section 7.3, we will discuss the S^2EC^2O run-time component in detail and provide an example of how to use it. The run-time component realizes the run-time monitoring of S^2EC^2O by detecting security violations, and providing run-time call traces. It is also able to carry out *countermeasures* when a security violation is detected at run time. The developer is able to specify these countermeasures, for example as part of the source code.

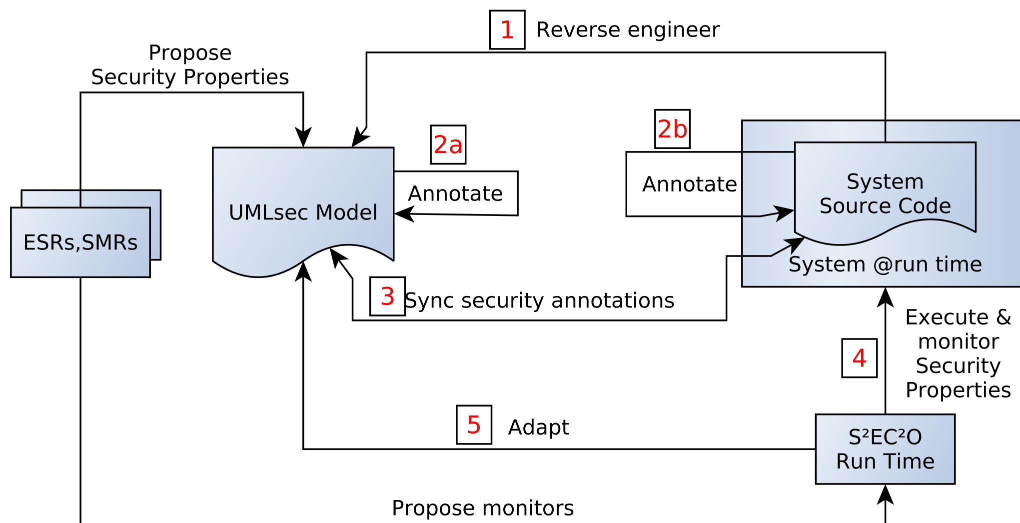


FIGURE 7.1: Approach to realize run-time monitoring of security properties specified at the model level

Section 7.4 shows how run-time logging information can be used to support round-trip engineering. For example, a call sequence leading to a security violation can be logged by the S^{2}EC^{2}O run-time component and fed back into the system design to support the security expert in understanding a malicious sequence of calls, thus providing support in fixing vulnerabilities of a system permanently. Logged data can be used to detect previously unknown classes, the data can also be processed to show a behavioral sequence diagram related to the system model to trace back how an attack may be realized.

We conclude this chapter with Section 7.5 by presenting relevant work of the research area.

We will review this chapter's contribution to the research question in Section 11.3.

7.1 Run-Time Monitoring with Run-Time Insights

This section gives an overview of the run-time approach and shows how it is integrated in the other components of S^{2}EC^{2}O as they have been introduced so far. A detailed, more technical discussion follows in Section 7.3. When security properties of a system need to be monitored at run time, several challenges occur. In most cases the system to be executed is provided as source code. Thus, there is a need of coupling code and model. The source code may differ from the model because often a system model is on a more coarse grained abstraction level than the source code. Also, even when model-based engineering is employed, the source code may evolve independently from the model, for example by connecting or exchanging external libraries. Thus, security requirement annotations made in the code may also be synchronized to the model and vice versa.

There is a need for a run-time component that realizes the monitoring. Regarding security-relevant applications, it is desirable that the monitoring mechanism is in best case not a part of the code to be monitored. This fosters that monitoring can be manipulated or switched off from the monitored code, by bypassing access

restrictions. For example, in Java, reflection and aspect-oriented programming are two methods of manipulating behavior of running code.

To tackle these issues, we introduce a run-time monitoring approach. Figure 7.1 gives an overview of it. The monitoring activities are realized by the *S^2EC^2O run time*-component. We detail its usage step by step according to the annotated numbers:

1. As we already discussed, model-based software engineering may lead to the fact that source code and model are not congruent. Using existing tooling, it is possible to reverse engineer additional model elements by analyzing the source code.

2. The security expert needs to annotate the model with security requirements. This can be achieved by annotating the model directly as defined by the S^2EC^2O initialization process as introduced in Section 2.2.3 (on page 15) and detailed in Chapter 6). This is depicted as step 2a. Additionally, code my be annotated directly (step 2b).

3. The annotations of the UML model can be synchronized with the source code and vice versa. This can be realized using graph transformation techniques. This step especially is necessary in case a mixture of source code and model annotations is used.

4. Applying the preceding step results in source code annotated with security annotations as proposed by the ESRs.

   The annotated Java source code is compiled and equipped with monitoring extensions. The resulting program is monitored during execution by the *S^2EC^2O run time* for security violations.

5. The monitoring can lead to discovery of executed code that is, so far, not part of the model. For example, a call dependency between two classes that is not part of the design and may also not be determined by static checking. In this case, the model can be adapted to document this behavior. The security expert can then decide if this behavior is either malicious or benign but raises the need for adapting the system model.

   More generally, in the case of a monitor finding, the run-time information can be generated into the model and related to existing elements like classes, to help the security expert in gaining insights regarding how an attack may have been performed.

In the subsequent sections, we show how the run-time monitoring shown as step 4 in Figure 7.1 is realized in detail. We also introduce possible reactions the security expert can define that are to be carried out during run time in case a security violation is discovered.

To limit the scope of this thesis, the approach presented here focuses on *secure dependency* as UMLsec property to be monitored at run time, but the approach can be extended to support additional annotations and monitors.

7.2 Specifying Security Properties

First, it is desirable to find and fix security vulnerabilities as early as possible in the development process. Alternatively, it is desirable to provide support to automatize detection of and reaction to breaches [BOCB+17]. But, unfortunately, many security violations (for instance previously unknown ones or vulnerabilities based on Java reflection) are hard to detect statically in the system design or source code [MNGL98, EL02, CM04].

Second, the ongoing modularization of software makes a project depend on external libraries. This can introduce vulnerabilities into a project and put the overall system's security at stake.

For example, OWL API [MMG], the Java library to work with OWL 2 ontologies as introduced in Section 4.2.1, depends on 39 libraries (version 4.5.1). Thus, system administrators may not know whether the software product they are running depends on libraries which may contain vulnerabilities. This is security critical because these external libraries get access to the developer's and/or administrator's system.

More concretely, security properties defined in a system which relies on trusted libraries might not be satisfied any longer if, for example, a new library version is introduced having vulnerabilities that are not known by now. Keeping track of all potential vulnerabilities, particularly those discovered only recently but not yet fixed, can be a challenging task.

We conclude that, even if a system features a design that is proven to be compliant with its security requirements, and even if the source code has been implemented in all conscience, dependency to external libraries can put security at risk. On this account, we want to tackle this challenge by providing run-time monitoring based on security requirements to be used at both the source code level and model level. Security requirements at the source code level may better reflect implementation details, while model annotations can be used to provide a better overview.

7.2.1 Specification of Security Requirements at Model level

To support modeling of security requirements at the model and also the source code level, a concept is needed that works for both levels and also allows coupling between model and source code level. A tight coupling may be unwanted because in model-based engineering, the source code, in comparison to the system model, may be subject to manual adaptations. Regarding specification of security requirements in system models, we make use of UMLsec as used in the design-time co-evolution part of S^2EC^2O in Chapter 6. The approach we present in this chapter is focused on but not limited to ⟪secure dependency⟫. We chose it, because it is a security property to handle compliance with security requirements between two partners, being appropriate for considering working with external libraries.

We shortly summarize the security property *Secure Dependency* of UMLsec and then discuss, how we extend its definition into run time for this thesis. Secure dependency originally is concerning the static structure of the system. It ensures that call and send dependencies between objects respect the security requirements on the data that may be communicated along them. Detecting all dependencies which can occur at run time statically, on the models or a concrete implementation, has been shown to be statically undecidable. The use of Java reflection depending on user input is one of the challenges to this matter [MNGL98, LWL05].

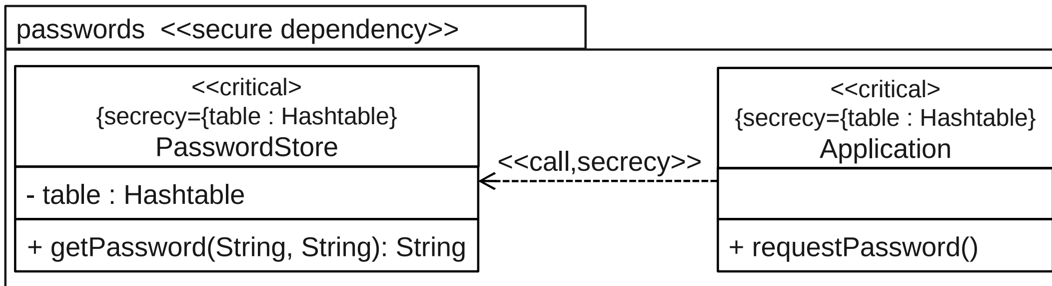


FIGURE 7.2: Example of UMLsec secure dependency application

The following definition, adapted from [JJ05] addresses *secrecy*. The *integrity* case is defined analogously.

A subsystem fulfills secure dependency *iff for all* `call` *or* `send` *dependencies d from an object C to an object D the following conditions hold:*

1. for all $s \in D.members$: $s \in C.secrecy \Leftrightarrow s \in D.secrecy$,
2. for all $s \in D.members$: $s \in C.secrecy \Rightarrow d$ is stereotyped *secrecy*, where s refers to the signature of a member.

We show how «secure dependency» is used by an example in Figure 7.2. It shows a UML model of a password manager, inspired by the secure storage concept of the Eclipse IDE [Ecl13], providing a public Application Programming Interface (API) to different applications and requiring secure dependencies. The class `PasswordStore` provides a method to retrieve passwords from the store via a public API. Also, a table with `private` visibility in which all passwords are stored is part of the class. Stored passwords can be requested using the method `getPassword(String, String):String` which checks a master password before returning a stored password. Accordingly, this operation accepts the ID of the application for which a password has been stored and the master password as parameters. As the class `PasswordStore` is annotated «critical» and the tagged value `secrecy` holds the signature `table:Hashtable`, all classes with a dependency stereotyped with «call» have to respect this secrecy security level. This in turn holds for `Application`, also equipped with the respective `secrecy` tagged value. The security-relevant connection between both classes is represented by «critical» containing this signature and a «secrecy» stereotype on the dependency.

In the implementation of a system specified in such a UML model, the dependencies stereotyped with «call» and «send» are usually implemented as method calls and field accesses. Even if a static model does not suffer violations, at run time it has to be guaranteed that the security properties specified at design time are not violated. A violation can occur due to an exchanged library or due to malicious code.

```
1 class MaliciousApplication {
2  public Hashtable readPasswords(PasswordStore s) {
3   Field f = s.getClass().getDeclaredField("table");
4   f.setAccessible(true);
5   return (Hashtable) f.get(s);
6  }
7 }
```

LISTING 7.1: Source code of a malicious application

Listing 7.1 shows an excerpt of a modified application using the API of the password manager in a malicious way. It tries to read all passwords stored in the password manager by using the Java reflection API for accessing the `private` password

table of the class `PasswordManager`. To achieve this, (see lines 3 to 5) at first a `Field` object representing the table is requested, set to accessible and finally the value of this field is requested and returned for the `PasswordManager` object given to the method as parameter.

7.2.2 Security Requirements at Source-Code Level

UMLsec originally does not support annotation of source code, but it provides developers with possibilities to annotate (UML-)models with security annotations.

A similar mechanism also exists for Java, namely Java annotations. With Java annotations it is possible to define extensions to the syntax to annotate source code. Hence, we introduce a set of Java annotations that realizes source code annotation to specify annotations as defined in UMLsec. We focus on annotations for methods and fields as these are the most relevant constructs if the call and data flow of a system is to be traced.

Especially, we support monitoring of the secure dependency property by adding a `@Critical` Java annotation that is semantically identical to «critical», as well as Java annotations for directly annotating class members: `@Secrecy` and `@Integrity`. As «critical» has the tagged values `secrecy` and `integrity`, the `@Critical` annotation has parameters `secrecy` and `integrity` which provide, as well as «critical», arrays of member signatures provided as strings.

```
1  @Critical(integrity={"checkMasterPassword"})
2  class PasswordStore {
3   @Secrecy
4   private Hashtable table = new Hashtable();
5
6   public String getPassword(String id, String pwd){
7    if(checkMasterPassword(pwd)){
8     return table.get(id);
9    }
10   throw new SecurityException();
11  }
12 }
```

LISTING 7.2: Source code of the password store with security annotations

Listing 7.2 shows an adapted version of the *PasswordStore* introduced in Figure 7.2. The tagged value `secrecy={table:Hashtable}` of «critical» is represented by a `@Secrecy` annotation on the `table` field in line 3 of the example. Additionally, the security requirement `integrity` is specified for a member with the signature `checkMasterPassword` in the `@Critical` annotation (see line 1). The method is called in line 7.

The next section elaborates on how the annotations of these two artifact kinds are mapped.

7.2.3 Mapping of Model Level and Source Level Annotations

To synchronize UMLsec annotations of a UML model with annotations in source code, mapping between this different kinds of annotations is needed.

Table 7.1 shows a mapping between UMLsec stereotypes as introduced by the UMLsec profile and respective source code annotations.

UMLsec stereotypes provide abilities to model all information regarding security levels within the tagged values `secrecy` and `integrity` of «critical». Similar to this we defined a `@Critical` Java annotation with the parameters `secrecy` and `integrity`, as can be seen in the first three rows of Table 7.1.

TABLE 7.1: Mapping between UMLsec model and source code annotations

UMLsec stereotypes			Java annotations		
stereotyped	stereotype	tagged value	annotated	annotation	parameter
Classifier	«critical»		Class	*@Critical*	
Classifier	«critical»	*secrecy*	Class	*@Critical*	*secrecy*
Classifier	«critical»	*integrity*	Class	*@Critical*	*integrity*
Classifier	«critical»	*secrecy*	Method/Field	*@Secrecy*	
Classifier	«critical»	*integrity*	Method/Field	*@Integrity*	

«critical» on classifiers is equivalent to the `@Critical` annotation on classes as well as their corresponding tagged values and parameters. In addition to the `@Critical` annotation on classes, we can directly annotate methods and fields using the mapping with `@Secrecy` and `@Integrity`. Usually, methods and fields are annotated by explicitly stating them as part of the respective secrecy and integrity tagged value of «critical». To avoid errors by mistyping and raise clarity and readability in larger classes, we also support that methods and fields can directly be annotated with `@Secrecy` and `@Integrity` respectively.

7.2.4 Synchronizing Model and Code

Considering the problem of mapping UML elements to Java code, mappings already have been defined in various reverse engineering approaches [Ton05, LAS17]. Unfortunately, existing mappings only consider a one-shot mapping. Thus, the challenging part is to keeping up with continuous evolution of both UML model and Java source code. Both can evolve independently. Existing approaches are using graph transformation techniques providing model synchronization to deal with the issues arising from this evolution [PKLS15, LAS17] and can also be applied to our annotations.

Regarding steps 1 and 3 of the run-time approach as defined in Figure 7.1 (on page 78), we may have to reverse engineer UML diagrams from source code and have to keep both in sync afterwards. This does also apply to annotations as introduced before.

Plain UML class diagrams can be reverse engineered from Java source code using existing tools like MoDisco, Visual Paradigm or Architexta [Theb, Vis, i3l]. Unfortunately, existing tools usually do not support synchronization of the reverse engineered UML models with the source code.

Leblebici et al. already specified rules for transforming Java programs represented as instances of the MoDisco [Theb] meta model into UML class diagrams using their graph transformation tool eMoflon [eDT, LAS14]. To utilize this, a source and a target model need to be provided (i.e. a UML model and a MoDisco model representing the source code) as well as the changes on one of these two.

Regarding changes to the UML model, Papyrus [Thec] is able of directly providing fine grained edit operations. Papyrus is a very powerful UML editor which also directly supports the UMLsec profile as part of CARiSMA, the current tool support for UMLsec [APRJ17].

The eMoflon tool can use the MoDisco representation of source code as well as a UML model as input and take model change events of Papyrus as delta information into account. Leblebici et al. [LAS17] propose a set of TGG rules able to synchronize

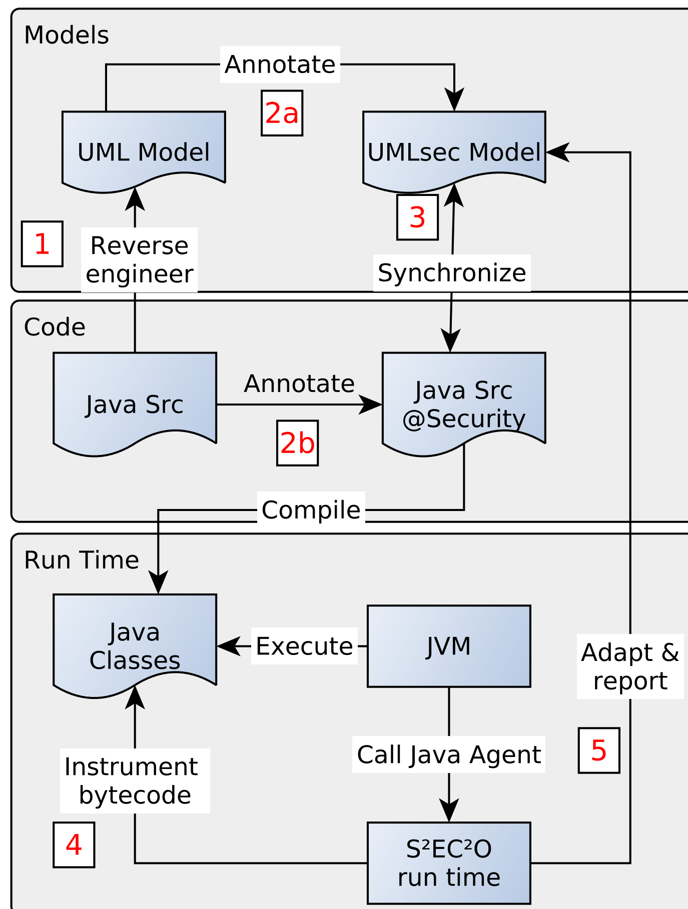


FIGURE 7.3: Structure of the run-time monitoring approach regarding software development abstraction layers

UML models and code. Rules to support the UMLsec-related annotations need to be added analogously.

7.3 Round-Trip Engineering Approach for Security Monitoring

Figure 7.3 shows the approach in detail that we sketched in Figure 7.1 (on page 78). It especially shows the different abstraction levels in model-based engineering and which steps of the approach couple the respective levels. In the following, we discuss the steps presented in Figure 7.3, emphasizing how the different layers are coupled. Afterwards, we present the realization of the steps 3, 4, and 5 by examples.

1. As we discussed in Section 7.1, it also applies to model-based engineering that source code is developed independent of the model. Step 1 can be realized using existing reverse engineering tools like MoDisco [Theb].

2. After the system model is at hand, it needs to be annotated according to the security annotation obligations given by S^2EC^2O. This can either be done by annotating the system UML model directly with UMLsec annotations using

Papyrus [Thec] (step 2a), or by annotating the source code directly using the Java annotations and mapping to UMLsec we introduced in Section 7.2.4.

3. Security annotations in model and code need to be kept in sync. This can be realized using graph transformation techniques such as Triple Graph Grammars (TGGs) and respective tooling like eMoflon [LAS14]. To let a graph grammar work with source code, a model representation of the source code is needed. This (step 3) can also be achieved using existing tooling like MoDisco.

4. Using steps 1 to 3, source code as well as a UML model can be annotated by the security expert according to the security annotation obligations. Thus, a UMLsec model annotated with security requirements can be coupled with Java source code annotated with respective security annotations. The annotated source code is compiled into Java classes. The compiled source code is executed by an unaltered JVM, supported by a Java agent [Ora]. A Java agent basically is a Java program equipped with a special `premain` method that is called by the JVM prior to the `main` method of the actual program. The JVM can be instructed to execute a Java agent along with an arbitrary Java program by providing the Java agent's main class as a command line argument. S^{2}EC^{2}O run time is realized as a Java agent.

   The monitoring approach is realized in a way that the *monitoring* code is separated from the *monitored* code. This is achieved using bytecode instrumentation. Section 7.3.1 gives details on which code snippets are used to instrument the application's code. Throughout this chapter, we already mentioned several types of reactions that are conceivable, as soon as a security violation has been discovered by the monitoring. One obvious type of reaction is to equip the code with actual countermeasures to mitigate or even defend an exploit.

   In Section 7.3.2 we discuss in detail, how countermeasures are actually implemented and executed. Thus, the S^{2}EC^{2}O run-time component is started alongside the system to be monitored and instruments it. Security annotations at the model level lead to executable source code where security properties are monitored.

5. While monitoring the system, monitor findings can occur. First of all, dependencies discovered during run time which are not part of the system design can be discovered. This especially applies to external libraries and call dependencies which cannot be checked statically. These insights can be used to adapt the UMLsec model automatically, contributing to round-trip engineering. For example, reports of incidents and automatically induced countermeasures can be shown as part of the model, by relating to existing classes. Thus, the security expert is able to design a system on the model level and getting feedback at the same level. Section 7.4 shows the round-trip mechanisms supported by S^{2}EC^{2}O with their realization.

7.3.1 Verification at Run Time

This section shows, using secure dependency as an example, how run-time monitoring using via instrumentation is realized.

Step 4 in Figure 7.1 (on page 78) is to execute the annotated source code and to monitor the execution for security violations. To ensure that we detect every security violation with respect to secure dependencies, we have to check all related method

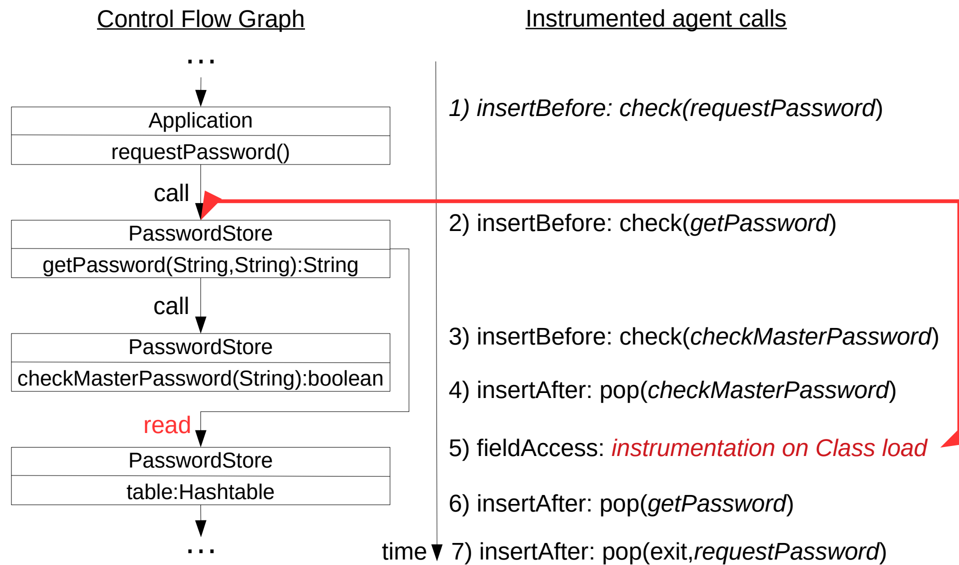


FIGURE 7.4: Events monitored at run time

calls and field accesses for their compliance with the specified security properties. We thus need to trace the call flow and take action as soon as a method is entered or exited. We reify monitoring by instrumenting the compiled code using Javassist, a mature framework for bytecode manipulation of Java programs [Chi, Chi00].

The instrumentation needs to take place at run time because it is not foreseeable which classes will be loaded at run time. Moreover, additional classes can also be dynamically loaded at run time via the Internet. To instrument the program to be monitored, we use a Java Agent which is called before the main method of a Java program is called. The source code transformation is triggered every time a class is loaded. The agent instruments appropriate code to conduct the secure dependency check at run time. Classes of the JRE core are not instrumented because they cannot be altered. S²EC²O's classes are not instrumented to avoid an infinite loop.

First of all, all annotations relevant for ⟪secure dependency⟫ are gathered. This covers the `@Critical` annotation as well as individual `@Secrecy` and `@Integrity` annotated methods, and fields. After that, every method that is annotated with at least one of the annotations or referenced by the `@Critical` annotations, the monitoring code is instrumented in a way that it is inserted before and after the actual method body. Additionally, all fields are instrumented with secure dependency checking code.

The agent provides a global set of stacks for call traces, one stack per thread. Whenever a method is entered, the conditions of secure dependency need to be checked: It is checked if the method to be executed is trustworthy. Additionally, the method is pushed to the stack for tracing purposes. After the code of a method is finished and before the `return` finally is initiated, the method is popped from the stack.

Figure 7.4 exemplifies the operating principle of the S²EC²O run time component with an example. We show a sample execution of an application accessing the *PasswordStore*. On the figure's left hand side an excerpt of the control flow graph is shown. First, the method `requestPassword` in the class `Application` is called. This method in turn calls the `getPassword` method of the class `PasswordStore` (its implementation is shown in Listing 7.2 on page 82). According to its implementation, at

first the method `checkMasterPassword` is called and eventually the requested password is returned from the field `table`.

The right hand side of Figure 7.4 shows the instrumentations that are issued alongside the method calls and the field access. The run-time monitor we propose is realized distributed over instrumented blocks:

- *before a class is loaded* to check for potential forbidden field accesses in,
- *before a method body begins* to check the secure dependency conditions (check if the called method is trustworthy), and
- *after a method body* has been executed to clean up the stack of called methods. All field accesses are monitored regarding the compliance with 《secure dependency》 as well.

Regarding the *secure dependency* property, whenever a method is entered or exited or a field access may happen, this is relevant as we have to check if the call or access of the respective members is allowed. To accomplish this, whenever such a relevant event occurs, we need to trace the control flow graph backwards and check both if the originating method is annotated as expected and if the accessed member is annotated as requested by the originating method.

To realize the instrumentation as above, we make use of the three Javassist methods `insertBefore`, `insertAfter` and `fieldAccess`. The (byte-)code of the first two is inserted before and after the body of a respective method. Before a method is executed, the secure dependency check is run and the method is put on the stack. After all outgoing edges of a method node (in the control flow graph) have been processed, the method is popped from the stack.

Regarding fields, every field access is investigated regarding 《secure dependency》 and eventually adapted.

In what follows, we discuss the code snippets that are instrumented into the application. Listing 7.3 shows the check for field accesses, while Listing 7.4 shows the code that gets executed before an actual application method is run. Listing 7.5 shows the code snippet that is run after an actual application method has been executed.

```
1  input var fieldAccess
2
3  // Checking secrecy only for read accesses
4  if (fieldAccess.isRead() & fieldHasSecrecy) {
5   if (!secrecy.contains(fieldSignature)) {
6    fieldAccess.replace(counterMeasureFieldRead(fieldAccess, field));
7   }
8   if (secrecy.contains(fieldSignature) & !fieldHasSecrecy) {
9    fieldAccess.replace(RESULT=fieldValue);
10  }
11
12 // Checking integrity only for write accesses
13 if (fieldAccess.isWrite() & fieldHasIntegrity) {
14  if (!integrity.contains(fieldSignature)) {
15   fieldAccess.replace(counterMeasureFieldWrite(fieldAccess, field));
16  }
17  if (integrity.contains(fieldSignature) & !fieldHasIntegrity) {
18   fieldAccess.replace(NEW_VALUE=fieldValue);
19  }
20 }
```

LISTING 7.3: FieldAccess code to perform the field access check during run time

```
1 global var stackSet // Global set of stacks of called methods for threads
2
3 checkSecrecy(stack.peek(), method(this))
4 checkIntegrity(stack.peek(), method(this))
5 stack.put(method(this))
```

LISTING 7.4: BeforeMethod check part to perform secure dependency check at run time

```
1 global var stackSet // Global set of stacks of called methods for threads
2
3 stack.pop(method.(this))
```

LISTING 7.5: afterMethod to pop the method from the stack

The approach presented here also makes use of logging and debug output necessary for realizing the run-time monitoring's subsequent functions, but in the listings we show in this section, we omit method calls for logging, printing debug messages, etc. for clarity reasons.

To verify the secure dependency property, we need to trace which method is the originating one of the current access to a method or field. To realize this, we store a method call stack (line 1 in Listings 7.4 and 7.5) whose top is the last method with outgoing accesses. A global map of stacks is provided by the S^2EC^2O run-time component. Each thread is assigned a stack using the thread itself as key.

In case a new method is entered, we have to check if the security requirements of this method are fulfilled by the caller and vice versa. The method `checkSecrecy` validates, by using the source and sink of the given access, if the secure dependency property holds for the `secrecy` security requirement. The method `checkIntegrity` works analogously for `integrity`. In case one of the validations fails, we provide various reactions which we will introduce in Section 7.3.2.

In addition, to verify if the caller (always top of the `stack`) and the entered method provide all annotations required by *secrecy*, we use `checkSecrecy` in line 3 in Listing 7.4. In line 4, the same for integrity with the method `checkIntegrity` is done. Afterwards, we put the entered method on top of the stack as it will be the source of the next calls. When all statements of an method have been executed, the method is popped from the stack (see line 3 in Listing 7.5).

```
1 checkSecrecy(caller, thisMethod){
2  if (caller!=thisMethod && caller!=null){
3    boolean callerHasSecrecy = caller.getSecrecyAnnotations.contains(
        thisMethod);
4    if (!callerHasSecrecy){
5    violations.add("secrecy");
6      return earlyReturn();
7    }
8  }
```

LISTING 7.6: CheckSecrecy method to check the compliance of the secrecy requirement

Listing 7.6 shows the pseudo code for the `checkSecrecy()` method. In case there is a valid caller at the top of the stack and the calling class is not the class itself, the check is carried out (line 2). It is then checked if the caller has the signature of the instrumented method in its set of `secrecy` annotations (line 3). If this is not the case (line 4), a violation has been detected. One type of countermeasure we will introduce in the succeeding section is called *early return* (line 6). In case a violation has been detected and such a method is defined, it is called.

Regarding field accesses, these are instrumented using the `fieldAccess` method of Javassist. This code snippet is also executed at class loading time. In contrast

to the other snippets, it does not precede or succeed the regular method execution, but it eventually manipulates the code for field accesses in all methods. The agent we provide applies the instrumentation only if the respective field is not declared in the loaded class. A variable containing reference to the field that is going to be accessed is available via the instrumentation API in the `fieldAccess` object. Also at class loading time, the sets `secrecy` and `integrity` are populated by collecting all respective field/method and `@Critical` annotations of the class.

The secrecy check is done for reading accesses (see line 4 of Listing 7.3). If the field to be accessed is not contained in the annotations of the calling class, the secrecy requirement is violated. In this case, the behavior of the field access is replaced by retrieving and returning the *early return* value (see line 6). We will give details on early return in the succeeding section. In case the signature of the field is an element of the `secrecy` set but the field itself does not have the secrecy annotation, the field to be accessed violates the secure dependency requirement. In this case, the access is still granted but the violation can be logged.

For the case of integrity check (see line 12 and following), the check is implemented analogously. In case of violation (line 15), the value written into the field is the early return value. It thus can be used to have a clearly defined value in the field in case of violation. If no early return value is defined, the write access is omitted by writing the same value into the field that it had before the write access came up. In case of reflective field accesses, method calls declared in `java.lang.reflect.Field` are instrumented, the check itself is realized analogously to Listing 7.3.

7.3.2 Countermeasures

Plain run-time monitoring is a passive method so that security violations are just recorded. This does only provide limited benefit for production environments. In this case, security violations are not prevented. A security run-time component has a substantial benefit if it is able to react to violations detected at run time (*Countermeasures*).

S^2EC^2O supports four kinds of countermeasures that go beyond simple shutdown and facilitate reactions to both keep the system running and prevent violations and thus prevent data from harm at run time:

1. Log actions of potential attacks for future evaluation.
2. End the attack by shutting down the application.
3. Return a statically defined value instead of the real value.
4. Call operations implementing sophisticated countermeasures.

In the simplest case, the reaction is to just log the call or access which leads to a violation and all calls and accesses which take place after the violation. These can be used as a plain textual log file or leveraged sequence diagrams as we describe in subsequent sections. This different kinds of logging will not prevent damage caused by the occurred violation but they enable system developers to study the violation and adapt the system to prevent future damage.

To actively encounter a violation, we provide a number of reactions to stop exploiting a system and thus combine logging with additional countermeasures we discuss in the remainder of this section.

The simplest reaction is to terminate the system and notify the system operator. Especially when given a complex system where only a small part would be affected

by a vulnerability, this may be unwanted. Otherwise, provoking a system shut down can also be some kind of denial of service attack.

As an alternative to keep the system running and to actively prevent it and its data from harm, we support to change return values and values stored in fields in case a violation occurs. For instance, returning `null` is a valid alternative as it is a well-known reaction in case of unforeseen or unusual situations. This prevents the system from disclosing real data to an attacker and the attacker may not notice that the security violation has been noticed. For this reason the security annotations to be applied on methods support to have primitive, statically defined *early return* values. The values of accessed fields can be altered also.

Depending on the data returned and how often the same data is returned, the probability of the attacker becoming suspicious might increase. Returning the same bogus data repeatedly might allow the attacker to study how the system behaves in case of an attack and thus rise additional breaches. To cope with this, we also support to specify methods to be called to provide an early return value in a more sophisticated and dynamic manner. Additional countermeasures can be triggered, too.

In both cases, early return values are defined by a parameter `earlyReturn` of `@Secrecy` and `@Integrity`. This parameter can be any primitive type, String, `null` or the name of a method, within the class and without parameters, which should be called. The called method can perform any operation accessible from object's scope whose member has been accessed. To avoid accidental use of methods providing countermeasures at the regular program execution, we additionally provide `@CounterMeasure`: Whenever a method annotated in such a way is entered at run time, S^2EC^2O run time can prohibit this call by issuing returning `null`.

```
1  public class PasswordStore {
2   @Secrecy(earlyReturn="secure")
3   private Hashtable table = new Hashtable();
4
5   @CounterMeasure
6   public Hashtable secure(){
7    for(String key : table.keySet()){
8     table.put(key, randomString());
9    }
10   return table;
11  }
12 }
```

LISTING 7.7: Specification of a countermeasure for the field `table`

Listing 7.7 is a code snippet regarding the password store example. It demonstrates the usage of calling an additional method to determine an early return value: `secure():Hashtable` will be called if a security violation of the secrecy property of the field `table:Hashtable` occurs at run time. This method replaces all passwords stored in the table with randomly generated strings to look like real passwords.

7.4 Support Security Fixing with Run-Time Insights

After the detection of security violations, even if they have been prevented by the S^2EC^2O run time, the system still needs to be adapted to reduce the attack surface permanently.

For this purpose, the data logged by the S^2EC^2O run time can be used. To support a design-cycle on the model level, data collected during run-time can be

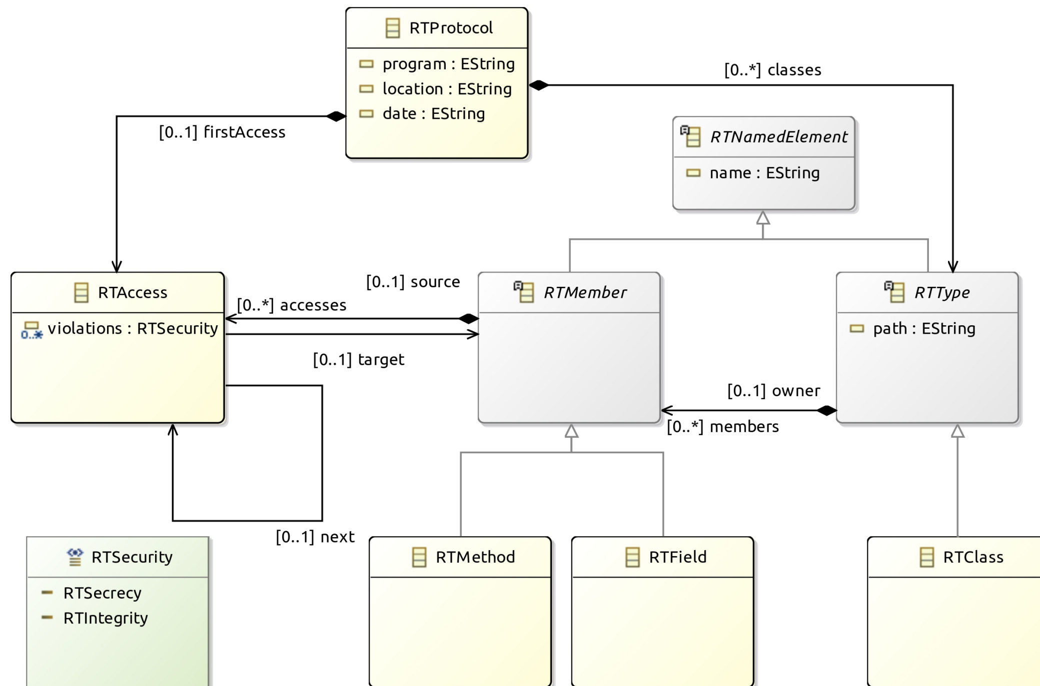


FIGURE 7.5: Ecore meta model for run-time protocols in S²EC²O

leveraged for the model level.

This automated evolution (see step 5 in Figure 7.1, page 78) covers:

1. addition of missing UML elements to the system model, and
2. documentation of security violations as sequence diagrams with direct references to involved UML elements.

In the following we show how this two evolution steps can be realized based on the gathered data.

7.4.1 Run-time Protocol for Subsequent Analysis

To realize the features as introduced above, run-time data needs to be gathered. The S²EC²O run time realizes this by instrumenting the application not only with the monitoring code representing the security check and countermeasures, but also by logging and documenting relevant calls including the involved classes.

This is currently realized by writing a protocol using the JavaScript Object Notation (JSON). JSON has the advantage of being platform independent and supported widely in diverse products. Writing the protocol can happen either off-line or online. In the latter case, JSON objects can be transferred via a WebSocket instantaneously, while the off-line variant has less latency.

Figure 7.5 shows our meta model that offers supplementary services for the analyses we introduce in the subsequent sections. For analyzing and utilizing the JSON protocol, the JSON protocol is parsed and an instance of the Ecore meta-model is built. This provides easy to access types and other convenience methods provided by the EMF. The meta model basically represents the content of the protocol. Besides some basic meta data like name of the executed program and location of the

program, methods, fields and classes can be distinguished. For easy access, class paths are also stored. The `RTAccess` class with its associations helps for tracing call and access sequences. The enumeration `RTSecurity` is used to specify the kind of security property violation that has been discovered.

```
{ "date": "2019-06-06",
  "location": "/home/is/where/wifi/connects/automatically/bin/.",
  "calls": [
    { "id": 0, "prev": -1,
      "clazz": "malware.MaliciousApplication",
      "member": "malware.MaliciousApplication.main(java.lang.String[])"},
    { "id": 10, "prev": 0,
      "clazz": "malware.MaliciousApplication",
      "member": "malware.MaliciousApplication.readPasswords(passwords.
          PasswordStore)"},
    { "id": 11, "prev": 10,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.getTable()",
      "violations": ["secrecy"]},
    { "id": 12, "prev": 11,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.secure()"},
    { "id": 13,"prev": 12,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.getTable()"},
    { "id": 14, "prev": 12,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.getTable()"},
    { "id": 15, "prev": 12,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.randomString()"},
    { "id": 16, "prev": 12,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.getTable()"},
    { "id": 17, "prev": 12,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.randomString()"},
    { "id": 18, "prev": 12,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.getTable()"},
    { "id": 19, "prev": 12,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.randomString()"},
    { "id": 20, "prev": 12,
      "clazz": "passwords.PasswordStore",
      "member": "passwords.PasswordStore.getTable()"},
    { "id": 21, "prev": 0,
      "clazz": "malware.MaliciousApplication",
      "member": "malware.MaliciousApplication.sendPasswords( java.util.
          Hashtable)"}
  ]}
```

LISTING 7.8: S^{2}EC^{2}O trace of run-time monitoring

Listing 7.8 shows the trace written during execution of the malicious application from Listing 7.1 (on page 81). We omit the attributes `bin` that reference the classes' base directory, since it is the same for every class. Every call object is identified by a numeric `id`, while `prev` eventually refers to the id of the caller. `-1` as `prev` means that it is the first method call. ID 11 shows the call of `getTable()` and the detected secrecy violation. What follows is a call of the early return method `secure()` with ID 12.

7.4.2 Addition of Missing Elements

One possibility to leverage the data gathered in the protocol is to add elements to the system model which are newly discovered during run time. This is especially

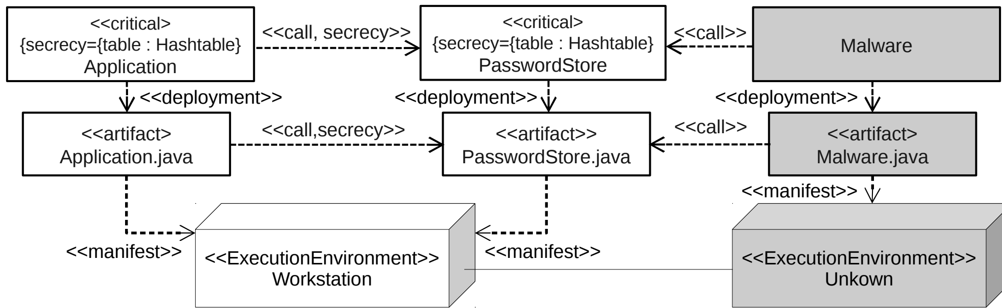


FIGURE 7.6: Deployment and manifestation of classes with evolution

important in case a security violation takes place using classes or methods which have not been anticipated. Another benefit, regarding usage of external libraries, is to reveal call dependencies between the application under consideration and the library code that have not been anticipated during development either.

The graphical depiction of a call trace as part of a UML model is both clearer than inspecting a call trace file and eases adapting the system model to the actual usage, because all relevant elements could be added automatically.

For example, in the top left hand side of Figure 7.6, the call between the two classes from the password store example we introduced in Figure 7.2 (on page 81) is shown.

Below these two classes we see on which artifacts they are deployed and on which execution environment they are manifested. The gray elements on the right hand side have been automatically added and are showing the malware we introduced in Listing 7.1 (on page 81) which has not been considered by the system developers.

Detection of missing elements is realized as follows. At first, the current UMLsec model is queried for all classifiers. After that, the protocol written during program execution as introduced in Section 7.4.1 is parsed into its EMF representation and then traversed in the same order as it has been written (i.e. chronologically). Every classifier being referenced in the protocol is tried to be resolved using the cached model classifiers. If this fails, the respective classifier is added to the model.

7.4.3 Documentation of Security Violations

To understand an attack, it is not only necessary to show a developer which method call or field access lead to an security violation or which structural differences exist, but it is of special interest which sequence of actions the attacker tried to perform. For the specification of call sequences, UML provides sequence diagrams [Obj17]. Sequence diagrams allow developers to understand which parts of the system are involved in a specific call sequence as the according model elements are directly used in the diagram.

As we argued above, there may be associations that are not covered by the model and cannot be detected statically. This especially applies to dynamic behavior introduced by libraries and reflective calls. While a JVM is monitored during execution of the target program (see step 4 in Figure 7.3 on page 84), S^2EC^2O run time keeps track of every method which has been entered and not exited yet.

Thus, we are able to check continuously if a call edge detected in the monitoring has respective elements in the model. If not, the tool can feed this information into the model by adding respective elements.

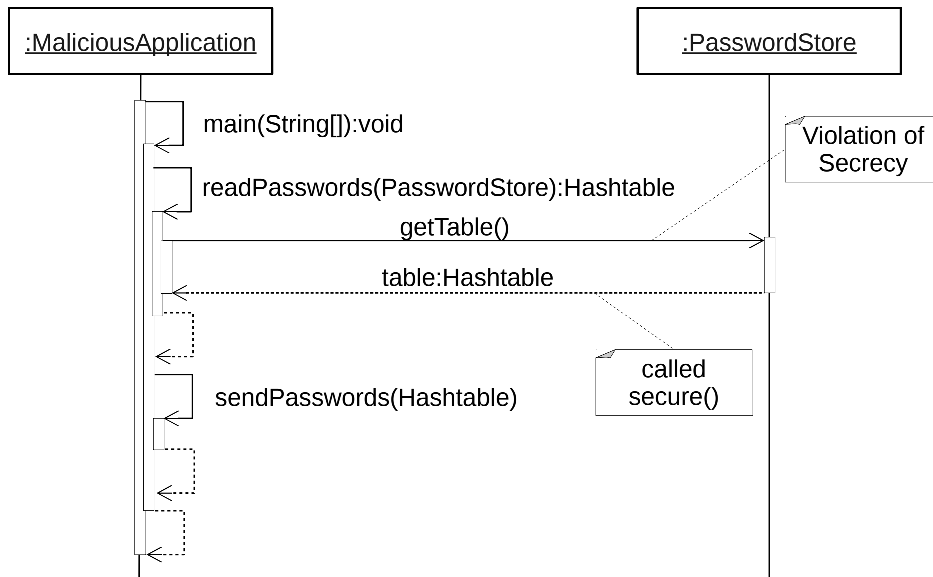


FIGURE 7.7: sequence diagram generated by S^2EC^2O run time

Figure 7.7 shows the sequence diagram based on the logged call sequence (see Listing 7.8 on page 92)caused by the malicious application we introduced in Listing 7.1 (on page 81).

The root of the security violation is the access to the field `table` visualized by a call of a `get`-operation which is annotated with *Violation of Secrecy*. Which countermeasures have been executed is also shown as annotation on the return value. In this case the operation `secure()` has been called as specified in Listing 7.7 (on page 90).

Furthermore, the sequence diagram contains information on which operation caused the violation (`readPasswords`) and by which operation this operation has been called (`main`). Also, beginning with the violation all following accesses have been recorded, fostering analysis of the malicious application's behavior. In this case this is just one additional method call from the method `main` to the method `sendPasswords`.

7.5 Related Work

In this section, we discuss related work of this chapter. As the publications that we took into account only cover a some aspects of this chapter, we categorized them. A number of approaches exist to support security at design and also at run time, but few approaches cover coupling the phases so far. For example, TamiFlex uses information gathered at run time about reflective calls for static analyses [BSS+11], while WebSSARI inserts run-time guards into web applications where a static analysis is not possible [HYH+04].

7.5.1 Taking System Context into Account at Run Time

Brézillon et al. [BM04] propose a model for building context-sensitive security policies. It is based on contextual graphs as introduced in their previous work [BPP02]. The authors define a *security context* as security relevant processes and mechanisms which form the environment of a user or an application. For example, access control mechanisms and cryptographic protocols are considered for the security context.

Contextual graphs are an extension to decision trees and describe possible action sequences when considering a (system) context which can have influence on possible course of actions. Determining the context and execution of actions do not take place independently but in an interleaved way. As soon as *contextual nodes* have gathered a sufficient amount of context information, fixed actions are automatically triggered. The authors claim that contextual graphs support incremental acquisition of knowledge.

This approach conducts knowledge management and solely uses a graph to model context knowledge and actions to be executed. Reactions to unforeseen changes, especially evolutions, are not considered.

Bodden et al. present an approach to lower the time to invest for run-time verification of big programs [BHL+07]. Given a sufficient number of end users, parts of the run-time verification is distributed among end users. Instead of instrumenting the whole product, only a partition of the program is instrumented at a time. Run-time verification is based on execution traces. Using regular expressions, traces for unwanted behavior are specified. The authors implemented two variants, noticing generally a high instrumentation overhead.

The approach has the goal of performing a security analysis of one given system, while S^2EC^2O's goal is to *protect* deployed system instances against security violations.

7.5.2 Undiscovered Program Activities

Lee et al. focused on Android and its powerful possibilities of inter-app communication, especially *activities* [LHR17]. These can result in a malicious flow of data or actions performed which cannot be anticipated upfront. Depending on how a developer uses and/or allows activities, the ability of starting another app may enable an attacker's app to inject arbitrary activities into the victim's app. Ultimately, a user interaction can be hijacked and the sandbox mechanism can be break through as the injected malicious activities can run in the context of a vulnerable victim app. To demonstrate the threat potential, the authors built malware targeting the Facebook app which realized that the malware was called when the user clicked a legitimate Facebook icon in another, benign app. The authors claim there are hundreds of ways to launch activities which calls for supporting app developers. Thus, they propose a static analysis tool making use of an operational semantics of the activity life-cycle, unveiling potential vulnerabilities.

In contrast to that, S^2EC^2O aims at providing the developer a lightweight model extension to cover up security risks in early design phases, coupled with the source code and run time.

Siveroni et al. [SZS10] conducted research on supporting design and verification of secure software systems, putting emphasis on the early stages of development like requirements elicitation. The proposed approach, a UML-based *Static Verification Framework*, realizes static verification of properties given in a specification language also provided by the authors. It enables to reason about temporal and general properties of a UML subset, for example UML state machines. Formal verification is carried out using the SPIN model checker.

The approach focuses solely on the early stages of software design and thus only properties that can be checked statically.

7.5.3 Security Monitoring

Ion et al. [IDC07] investigated the security policy architecture of J2ME (Java for mobile devices). While the Java Standard Edition provides an extensible security architecture, this is not the case for J2ME. J2ME's security manager is static, cannot make decisions depending on run-time properties, and has no ability to specify a per-application policy. The authors constitute that only a coarse-grained security policy approach exists. The authors present a modified version of the J2ME VM which is able to deal with security policies defined in the Security Policy Language (SPL) at run time. Flexibility of the security models for mobile computing platforms and the granularity at which the policies come into effect can be specified and enforced. These are the key contributions, with no considerable overhead.

In contrast to that, S^2EC^2O uses a native interface of the JVM API and thus does not require changes to the VM. By incorporating model-based design, we support developers in gaining additional knowledge about the how the code behaves during run time.

Costa et al. also present an approach that features a more fine-grained and flexible security mechanism for J2ME [CMM+10]. It mainly features a Policy Enforcement Point (PEP), intervening security-relevant calls and responsible for calling policy checking methods, as well as the Policy Decision Point (PDP), which interacts with the rest of the system to decide if the requested access is granted. Policies are defined based on the ConSpec language that has been defined by the Security of Software and Services for Mobile Systems (S3MS) project. Based on the 4 scopes object, session, multi session, and global, API-calls, for example such that open HTTP connections, can be bound to conditions. The authors implemented their approach in two different ways and compared both variants. First, similar to [IDC07], by adapting the Java VM. An issue encountered is that key words (for example method names) referred to by policies are restricted to the methods that the current PEP implementation can intercept. The second implementation, called in-lining, is based on byte-code manipulation. This requires the author's implementation to be installed on the mobile device. The user application (MIDlet) is decompressed, every API call in its byte code is preceded and succeeded by calls to the PDP. The authors evaluated their realizations in terms of performance overhead and in both cases noticed an overhead below 5%.

Compared to S^2EC^2O, the first variant requires a modified Java VM, while the second one manipulates the application code. However, the approach requires a high amount of formalization.

Hiet et al. propose to secure Java Web applications by monitoring information flows [HTMM08]. Their work extends Blare, an already existing intrusion detection tool on operating system level which considers processes es black boxes. The proposed tool JBlare makes internals of Java applications available to Blare. A policy based intrusion detection is then realized by tracing inter-method flows in Java applications, supported by the JRE calling an internal security manager before every I/O access. The authors encountered a substantial slow down of factor 12 for loading and factor 4 for the execution time. Blare requires a modified Linux kernel to run on, while JBlare requires a modified JRE, which are heavy-weight assumptions regarding the target environment.

However, detection of intrusion is covered in the publication, but, in contrast to S^2EC^2O, reacting to breaches or preventing them is not tackled.

Staicu et al. investigated on security vulnerabilities in Node.js-based applications [SPL18]. Node.js applications can also run outside browsers as desktop applications. The authors identify two APIs they call injection APIs, giving direct system access without benefits of a sandbox. Calls to these APIs can be used maliciously if not secured properly. Even more pervasive than Java is today, Node.js code is not only executed by browsers (desktop as well as mobile), Node.js applications are also able to be run as desktop applications and then interact with the system and without benefits of a sandbox. The authors show in a large-scale study of 235,850 Node.js applications vulnerabilities against injection attacks. For instance, a routine doing a backup can be misused to delete all existing backups, just by properly injecting a parameter for the command line code that is executed on the host system. The authors present an approach targeting this issue by computing a template of values passed to APIs that are prone to injections. In a second step they synthesize run-time policy from that to monitor fulfillment during run time. Checking is supported by their approach during design time (where possible using static checks) and during run time using code rewriting. The code-rewriting part shows to have a sub-millisecond overhead.

In contrast to S^2EC^2O, the approach of Staicu et al. is tailored to a specific programming framework and a specific kind of security vulnerability. The run-time enforcement relies on rewriting the code, which raises the question of how the enforcement parts of the rewritten code are endangered. S^2EC^2O's code manipulation is restricted to prior and after method execution, for example. However, they show the significance of early security analyses.

Chapter 8

Co-Evolve Run-Time Components of Systems

In the preceding chapter we showed how the security of a system can be monitored during run time. This is accomplished by providing a set of Java annotations which can be used to bridge the gap between a system model annotated with security requirements and code which can be executed. The model and code of the system can be coupled using TGGs. Countermeasures against detected violations at run time can be defined statically in the code.

The S^2EC^2O run time is realized by a Java agent accompanying the running code and providing central data structures like a call stack to realize the security checks at run time. Using S^2EC^2O's run-time monitoring approach, it is possible to design a secure software system, bring the security annotations to the code level and provide monitoring of security requirements at run time.

However, when the environment evolves, situations can arise where the system implemented to be compliant with the security requirements becomes insecure. Depending on the complexity of the surrounding infrastructure and the complexity of the security mechanisms to be adapted, it may take some time until an adapted system version is deployed.

Regarding systems where it is necessary to provide a continuous service, for example information systems for hospitals or infrastructure, it may be acceptable to take a calculated risk for the system whilst avoiding downtime. Thus, a system needs to be adaptable during run time so that it can react instantly to a security threat to lower the security risk while the permanent solution is being designed, tested and deployed.

We will reuse a part of the S^2EC^2O approach introduced in the preceding chapter and extend it to contribute to the following research question:

RQ5: *How can information gathered at run time be used to adapt the system when the context evolves, avoiding shutdown or additional design cycle?*

In this chapter, we tackle this research question by providing an adaptation approach that builds upon the run-time monitoring approach we introduced in the preceding chapter. The approach is able to adapt a system's behavior at run time as instant reaction, in case security threats are discovered. Section 8.1 first gives a motivating example from the traveling domain, exemplifying the goal of the adaptation approach. After that, we give an overview of the approach by defining relevant components and how they interact. Subsequently, we show where and how S^2EC^2O's monitoring approach can be extended to support adaptability. After introducing extensions to the modeling abilities and code annotation support, interaction (i.e. communication) between the run-time agent and the S^2EC^2O *service* (as we introduced in Section 2.2.2 on page 11) is defined. The interaction is further defined

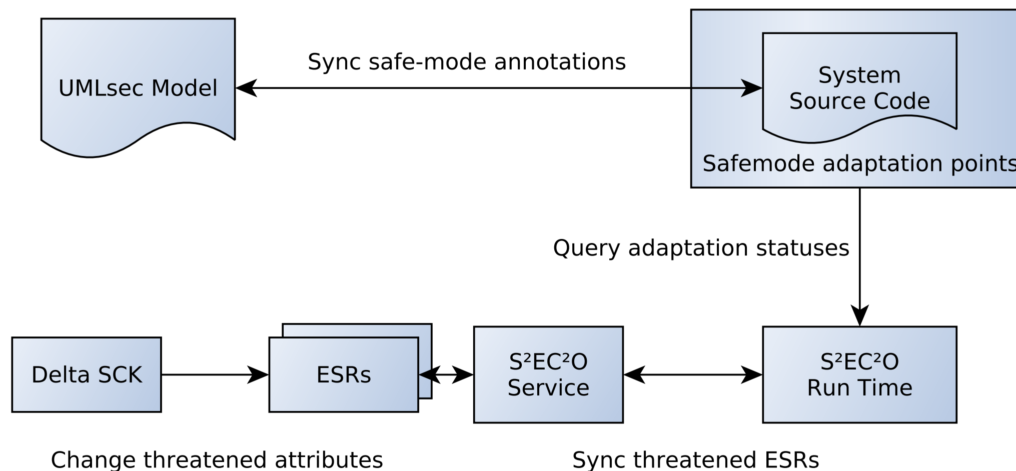


FIGURE 8.1: Overview of the S^2EC^2O run-time adaptation approach

based on JSON objects which can be transmitted. Section 8.2 closes the chapter by presenting relevant work of the research area. We will review the contribution of this chapter to the research question in Section 11.4.

8.1 Run-Time Adaptation Approach

As we described in the preceding chapter, as soon as a system designed to be secure is in production, security monitoring should be also put in place. In the S^2EC^2O approach as presented in Chapter 7, security as well as countermeasures are defined statically. This is insufficient in case of newly discovered security threats, when a system shutdown is not sustainable. This especially applies to systems providing services in critical domains, like medical information systems, pumping stations, etc. Apart from that, there are also systems which, when completely shut down due to security violations, would cause substantial losses in sales. In these cases, it may be desired to adapt the system's behavior to limit the attack surface and taking a calculated risk in favor of continuation of the service.

8.1.1 Motivating Examples

We give an example that also gained popularity in the media. On Dec 20th, 2018, the biggest German railway company, Deutsche Bahn, noticed fraud that has been committed using the online booking services and reimbursement coupons [Deu18]. Apparently, shutting down the whole booking system was not a viable option. So, as a reaction to the fraud, Deutsche Bahn restricted payment possibilities for tickets for certain payment methods until the fraud could be permanently prevented. For example, direct debit (*Lastschriftverfahren*) was forbidden for certain kinds of tickets.

Another example for this kind of need for adaption over shutdown affected a wide range of distributed systems. For a long time, the cryptographic hash algorithm SHA-1 was used, for example, to store passwords in databases. This algorithm was considered secure until the year 2005 when a method was published to break the security mechanism [WYY05]. Since the security of the authentication may depend directly on the security of the hash algorithm, the developers of an affected system can react to this change in the security context by replacing the algorithm

with another from the SHA-2 family. Replacing the hash function may take some time. Apart from the mere development effort, pending processes like testing, quality assurance and even additional business processes hindering deployment need time.

The kind of adaption to limit the risk of fraud or restrict the attack surface may be application specific. Consider an online shopping service. In an affected system, one can take a calculated risk as follows: For example, when regular user monitoring shows that most customers buy for less than 100 EUR per month, this limit can be used as a general *safe mode* for the system: limiting the maximal monthly turnover to 100 EUR per customer.

As a result, a few customers might be prevented from spending more money. Most customers, however, would not notice the limit since they spend less money anyway. The company's business is not obstructed and the turn-over is only endangered to a small extent during the time of patching the algorithm. When the new algorithm is in place, the limit can be removed. This strategy ensures that a sufficient degree of security is preserved at all times and with an optimal trade-off to limit negative impact on business.

Adaptation Approach Overview

The run-time approach we introduced in the preceding chapter realizes run-time monitoring and also carries out countermeasures at run time. The drawback is that the security checking code as well as the countermeasures are deployed statically with the code. In other words: once a system is running, there is no possibility of adapting its behavior additionally to reflect newly discovered threats. The run-time adaption approach we introduce in this section tackles this challenge.

Figure 8.1 depicts an overview of the run-time adaptation part of S^{2}EC^{2}O. It realizes adaptivity using annotations in the source code as well as the model level. It works by enabling the security expert to specify alternative methods that are called in case a given ESR is threatened. The Security Context Catalog already supports providing the information which ESRs are currently threatened by the `threatened` flag for each ESR (see Section 6.2.1). An ESR is considered *threatened*, when, according to the current SCK, a *threat* against a given ESR is modeled, the system under consideration is affected by this threat, and, up to now, no co-evolution has taken place to mitigate that.

The run-time *adaptation* approach is realized by building upon the S^{2}EC^{2}O run time, the run-time *monitoring* approach we presented in the preceding chapter. Precisely, the S^{2}EC^{2}O run time is reused and extended. We extend the run-time agent, so that state information about threatened ESRs are synchronized with the Security Context Catalog.

In the motivating examples, we determined the need to alter the system's behavior to, for instance, restrict the service or deactivate certain parts of the system. We call this *safe mode*.

We support this by introducing a `@SafeMode` Java annotation and a respective UML stereotype. The security expert is enabled to annotate the model or source code, respectively, with annotations that mark certain methods of the system relevant for a *safe mode*. With every annotation, a redirection method as well as a set of redirection conditions can be defined. Not only the threatened ESRs, for which the redirection shall be used, can be defined, but also the redirection method itself, which is to be called in as a *safe mode* method instead of the annotated one.

The run-time agent can query the Security Context Catalog, to check which ESRs are threatened currently. It needs a counterpart station during run time that provides access to the Security Context Catalog. This is realized by *S^2EC^2O service*.

The approach presented in this chapter contributes to the delta-handling process of S^2EC^2O as presented in Section 2.2.3 (on page 17). It realizes the activity *Adapt system@run-time*.

8.1.2 Adaptation Support at Design Time

The security expert needs to annotate the system at design time, to make it adaptable during run time. He needs to define which methods are relevant for the safe mode and for which threatened ESRs a reaction is requested.

```
1 @Target({ElementType.METHOD}) @Retention(RUNTIME)
2 public @interface SafeMode {
3
4   String[] conditions()  default {""};
5   String   redirection() default  "";
6
7 }
```

LISTING 8.1: Declaration of safe mode Java annotation

Listing 8.1 shows the declaration of a Java annotation `SafeMode`, which can be used to annotate methods. The string array defined in `conditions` is used to define the cases of threatened ESRs for which an alternative method shall be called. The string given in `redirection` defines which method shall be called instead when at least one of the ESRs defined in `conditions` is threatened.

Using fully qualified class names, it is also possible to load the method dynamically at run time from an arbitrary class. This can be realized using the technique of *early return* values, which we introduced in Section 7.3.1 (on page 85).

```
1  @SafeMode(
2   conditions = { "Data Integrity" },
3   redirection = "processOrderRestricted"
4  )
5  public void processOrder(){/* ... */ }
6
7  private void processOrderRestricted(){ /* ... */ }
8  }
```

LISTING 8.2: Example of safe-mode annotated application method

Listing 8.2 gives an example of using the safe-mode annotation. We consider the train ticket booking system as described in Section 8.1. We assume two methods for processing a customer's order. First, the method `processOrder` to be used for regular use. Second, a safe-mode variant that processes the order more restrictively, eventually processing additional checks, examining the customer's order history, etc. We assume an ESR *Data Integrity* as part of the Security Context Catalog which is used to express data processed by the system shall retain its integrity.

The adaptation during run time is then realized as follows: The S^2EC^2O run time instruments the method, so that prior to executing its body, it checks if one of the condition holds, i.e. one of the ESRs is currently flagged as threatened. If this is the case, the method specified with the `redirection` annotation is executed instead.

Thus, the S^2EC^2O run time needs to be informed about the current Security Context Catalog state. In the succeeding section we will describe how it communicates to query the set of threatened ESRs.

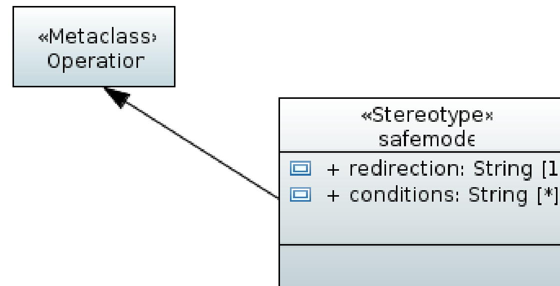


FIGURE 8.2: UML profile to support *safe mode* adaptations

Figure 8.2 shows the UML profile we define to be used to specify adaptation annotations at the model level. Its semantics is defined analogously to the Java annotations in Listing 8.2.

8.1.3 Adaptation at Run Time

At run time, there is a need for communication between the running system (i.e. the S^{2}EC^{2}O run-time agent) and the components of S^{2}EC^{2}O maintaining the Security Context Catalog (S^{2}EC^{2}O service). Moreover, S^{2}EC^{2}O and the system under consideration may not be executed on the same machine.

First, when the security expert triggers a run-time adaptation, there is a need of controlling the running system so that it gets adapted. This can happen when S^{2}EC^{2}O processes a SCK delta which leads to the insight that an ESR is threatened.

Second, when the system under consideration is started, the S^{2}EC^{2}O run time needs to query the Security Context Catalog for all ESRs which are currently flagged as threatened. Thus, the agent synchronizes the adaptation state of the monitored application to the state currently stored in the Security Context Catalog. To realize this, the S^{2}EC^{2}O run-time agent as introduced in the preceding chapter can be extended with a static class providing an interface for data exchange, for example a TCP socket. A widely used, light-weight technique for data exchange between distributed systems is to use JSON objects. In the following, we exemplify the operating principle of the adaptation at run time based on example JSON objects transmitted between the run-time agent and S^{2}EC^{2}O service.

Startup: Pull all ESR States

```
1 {
2   "DocumentType": "Pull ESRs",
3   "Timestamp": "2018-11-27 23:42:20.206",
4   "UID": "..."
5 }
```

LISTING 8.3: Example JSON object to query threatened ESRs

When the S^{2}EC^{2}O run time, accompanying the monitored application, is started, it connects to the S^{2}EC^{2}O service and queries the Security Context Catalog for all currently threatened ESRs. Listing 8.3 shows an example request sent from the run-time agent. A unique ID (UID) and a time stamp are part of the message

so that S^{2}EC^{2}O can judge regarding the age of the message and its uniqueness.

```
{
  "DocumentType": "All Threatened ESRs",
  "Timestamp": "2018-11-27 23:55:20.206",
  "UID": "...",
  "ThreatenedESRs": ["Secure Communication","Secure Login"]
}
```

LISTING 8.4: Example JSON object response to query threatened ESRs

A typical response is proposed in Listing 8.4. By referencing the same UID, the agent can match the response to the original request.

After receiving the threatened ESRs, the S^{2}EC^{2}O run time has this information available for all methods that are `@SafeMode`-annotated. Whenever such a method is to be called, the agent is queried if at least one of the ESRs defined in the `@Conditions` annotation is currently threatened. If this is the case, the agent initiates execution of the redirection method instead. If not, the ordinary method execution is resumed.

Push Newly Threatened ESRs

When the delta process of S^{2}EC^{2}O, as defined in Section 2.2.3 (on page 17), is executed and ESRs turn out to be threatened, (newly) threatened ESRs can be pushed to the S^{2}EC^{2}O run time.

```
{
  "DocumentType": "Push Threatened ESRs",
  "Timestamp": "2019-01-01 00:01:00.333",
  "UID": "...",
  "ThreatenedESRs": ["Secure Communication","Secure Login"]
}
```

LISTING 8.5: Example of S^{2}EC^{2}O service pushing threatened ESRs to the run-time agent

Listing 8.5 shows an example JSON object for that case. The set of threatened ESRs is submitted as an array containing the respective ESR identifiers.

```
{
  "DocumentType": "Push Threatened ESRs ACK",
  "Timestamp": "2019-01-01 00:01:00.333",
  "UID": "...",
  "ThreatenedESRs": ["Secure Communication","Secure Login"]
}
```

LISTING 8.6: Example response of the run-time agent to a received set of threatened ESRs

The S^{2}EC^{2}O service can be certain that the run-time agent received the message if it receives an acknowledgement JSON object as exemplified in Listing 8.6.

Reset ESR State after Threat is repealed

As we discussed in the introduction of this chapter, the purpose of adapting the running system is to do this only temporarily. The goal is to prevent a shutdown and lower the risks for exploits, while a permanent solution is being designed, implemented, tested, and brought into production.

After the permanent solution is in production, it is safe to remove the `threatened` flag of the ESRs in question.

This can be accomplished by issuing new *Push Threatened ESRs* messages containing the smaller amount of ESRs.

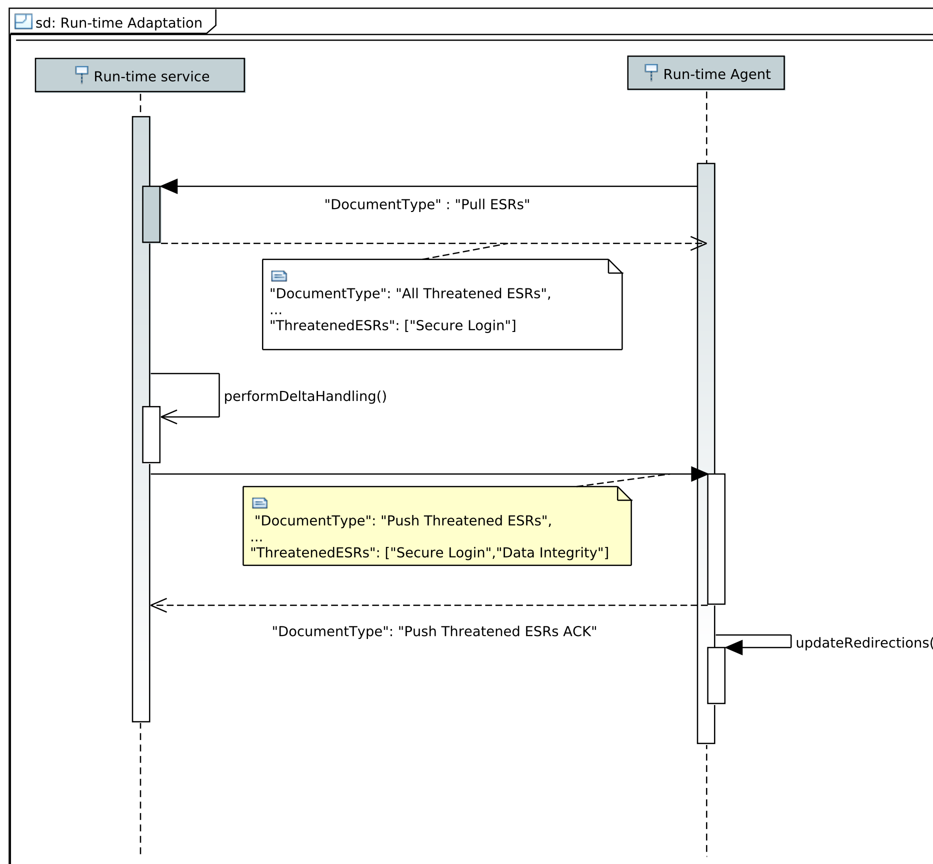


FIGURE 8.3: Sequence diagram showing adaptation interaction of the example

8.1.4 Example of Adaptation Interaction

Figure 8.3 shows the interaction between S^{2}EC^{2}O service and an application which is executed (instrumented) along with the S^{2}EC^{2}O run time. We consider the example as introduced in the beginning of this chapter. The sequence starts with the S^{2}EC^{2}O service already running. The run-time agent is started alongside the monitored application. It instruments all annotated methods of the application to query its internal state regarding redirection methods. The agent pulls the current threatened ESRs from the Security Context Catalog by issuing the respective query. The *threatened* states of both systems are now in sync.

At a later point in time, we assume the SCK changes and executing S^{2}EC^{2}O's delta process leads to new insights. In this example, the ESR *Data Integrity* is now flagged es threatened. This leads to a push message that informs the run-time agent by submitting an updated list of threatened ESRs. The run-time agent updates its internal state and redirects method calls accordingly then.

8.1.5 Traceability of Adaptations

```
1 {
2   "DocumentType": "SSECCO Report",
3   "SessionStart": "2018-11-27 12:52:20.206",
4   "Type": "    ",
5   ...
```

LISTING 8.7: S²EC²O report header for logging interaction with the run-time agent

To keep track of all incidents where adaptations have been affected, all of the messages introduced above are to be logged in S²EC²O reports. To differentiate between these objects and reports of other activities within the S²EC²O approach, a JSON object with a header shown in Listing 8.7 can be used, while `Type` can be one of the following:

- `ESR states pulled`
- `ESR status change`
- `ESR status change ACKed`

The object is then completed by adding the original message as introduced above.

8.2 Related Work

In this section, we discuss related work of this chapter. As the publications that we took into account only cover a some aspects of this chapter, we categorized them.

8.2.1 Application Behavior Adaptation

Tun et al. [TYB+18] propose an approach for behavior adaptation to retain compliance to security requirements. They focus on web-based distributed systems, precisely a document sharing server like ownCloud. As attack type, they investigated man-in-the-middle attacks, for example, parameter tampering attacks and cookie poisoning. The effect of such an attack is that, for example, a document sharing invitation of an inviter can be tampered, so that the request the server receives contains additional users. The authors assume that the user notices this behavior and actively revokes the sharing grant for the respective malicious users. Adaptation in the presented approach is realized using statistical machine learning techniques. In the example, a component containing the classifier analyzes every incoming sharing request and compares it to share revoke requests that have happened in the past. If a user in the sharing request is considered suspicious, the user is removed from the request, thus avoiding granting access to the malicious user in the first place.

The approach also focuses on adapting behavior to keep a system secure. However, adaptation cannot be controlled from an external system, contrasting S²EC²O. In their example, the authors make the assumption that a successful maliciously altered sharing request will always be discovered and acted upon. In contrast to S²EC²O, the approach focuses on adaptive behavior for a specific attack type and does not take the overall system development and maintenance into account.

Morin et al. present an approach to build a system which is able to adapt its security behavior at run time [MMF+10]. The approach is focusing access control.

It is considered a problem of inflexibility that in most systems, access control rules and also mechanisms are hard coded and weaved into the business logic. Thus, changes that need to be made to the access control policies take time as they require a design cycle (called *request for change*-process in the publication). By leveraging *models@run-time* and software components that can be composed into a product, the authors propose an access control component that is able to be updated at run time and the business logic that is built according to an architecture model that can be bound to an also provided access control model.

Compared to S^2EC^2O, the authors also gain adaptability of a system by providing possibilities to react to evolved security circumstances at run time. However, the approach currently is restricted to access control. It is built on *models@run-time* and also requires the running system to be built according to a concrete meta model.

8.2.2 Development of Adaptive Systems

Franch et al. [FGO+11] propose a model-based approach to develop adaptive, service-based systems. The requirements for system development are elicited using the goal-oriented notation *i** and classified into service categories. Every category is assigned a concrete metric which can be monitored during run time. For each category, different services are defined and the according adaptation rules based on this metrics are configured. To adapt the system based on the monitored run-time data, violations of service rules and error states are determined.

Security is not considered in this publication, but it gives insight on how to refine adaptive systems systematically from requirements engineering to deployment.

Evesti et al. [EPS10] present an architecture for security adaptation at run time for mobile and embedded systems. The authors propose to evaluate changes in the surrounding environment that may cause threats and to select appropriate security mechanisms automatically. On this account, they extend the ontology presented by Herzog et al. [HSD07] to model various security knowledge. This includes assets, security goals, threats, vulnerabilities, and countermeasures. The ontology includes a taxonomy for different context levels: (1) Situation context, (2) Digital context, and (3) Physical context. Each level contains concepts affecting the security of the system.

The authors note that automatic context monitoring is incapable of collecting all the required data, but no further suggestions for social-technical monitoring are made. S^2EC^2O features a semi-automated process of monitoring a system's context.

Salehie et al. present an approach to protect valuable assets at run time [SPO+12]. It aims to enable different countermeasures in face of various kinds of changes that may occur at run time. The authors' work is based on an asset model, a corresponding goal model, representing the requirements of the system, and a threat model. These models are used to generate a causal network required to analyze security in different situations at run-time and to enable effective countermeasures if required. The evaluation of the approach shows to potential of adaptive security for software systems. However, the presented approach is not capable of dealing with new threats that have not been considered while developing the threat model. To implement the presented approach, they developed the tool SecuritTAS presented in [PMS+12]. It supports software engineers in modeling assets and corresponding goals based on the requirements of the system as well as threats.

The tool appears rather sophisticated; unfortunately, software engineers need a lot of expertise in developing the required models as no security knowledge support is provided. S^2EC^2O aims at providing a light weight approach and fosters the re(-use) of as much knowledge that already exists publicly as possible.

Omoronyia et al. [OCS+13] propose an approach for developing adaptive systems which consider privacy. The systematic approach supports the user in compiling privacy-relevant requirements into software, based on a MAPE cycle. Privacy needs of the users are at first investigated to enable controlled disclosure of personal data. Attributes are identified which should be monitored to detect privacy threats. Threats are to be discovered before the user passed his data. Finally, a tool is used to assess the degree of threats coming from usage as well as passing personal data. Using a behavior model, a context model, and the privacy requirements of the user, the requirements to be monitored are chosen. The selected attributes are monitored to ponder benefit of using data against disclosing it. The conduction of disclosure is supported by a feedback loop which takes the system specification into account, because a system adaptation may be necessary as countermeasure. The proposed framework has been implemented as a tool [OPS+12]. The authors show that software not respecting privacy requirements fails at regulating the information flow, i.e. passing-on or disclosing personal data.

In contrast to that, S^2EC^2O is a light-weight extension to an existing software development process. It accompanies conventional software engineering instead of being a whole process on its own. Adaptation is realized by adapting system components or aspects during run time. This is also and especially possible for legacy systems, regardless of the engineering approach they have been built with.

8.2.3 Security-Aware Systems at Run Time

Xiao [Xia09] proposes an approach to realize security, access control to be precise, using models at run time. In detail, the approach is built on agent-driven software architecture. The approach is built using a system of models and rules. Assumptions on the software architecture of the software system to become security aware are also made as the business logic of the system has to be defined using so-called reaction rules. The access control is then defined by a set of policy rules that are shared among the agents. The agents in turn execute rules and have the connection to ordinary software components. The agents work goal-oriented, while the goal is that defined by the business logic. The approach realizes an extended version of Role-based access control (RBAC).

While gaining a flexible system out of the approach, it assumes the system to be built around the so-called agent-oriented model-driven architecture. The approach is restricted to access control, as well as evolution support only is provided for access control policies. Thus, the behavior of the system itself cannot be altered at run time.

Nhlabatsi et al. present an approach where assumptions about security requirements are monitored at run time [NYZ+15]. Design-time assumptions (as location of employees and (mobile) devices) are elicited. Security requirements are iteratively progressed and refined into system specification information. During run time, logs are written which continuously document a confidence level based on sensor data. The confidence level is used to monitor the validity of assumptions at run time. Causality is determined using temporal logic. If an incident occurs, logs are investigated automatically in order to identify the security control that is insufficient. While not addressing software engineering, the proposed approach still shows how design time requirements (assumptions) can be formalized and monitored during run-time.

The approach is not targeted directly against software development, but it shows a way of how design time requirements, i.e. assumptions, can be formalized and monitored during run time.

Chapter 9

Prototypical Implementation

In the main chapters of this thesis, we elaborated on the central aspects of the S^2EC^2O approach and the respective research questions. The covered aspects of S^2EC^2O include modeling of context knowledge using ontologies (see Chapter 4), detecting and assessing differences in ontologies (see Chapter 5), co-evolution of system models (see Chapter 6), monitoring security requirements at run time (see Chapter 7), and adaptation of the system behavior at run time (see Chapter 8).

This chapter shows the prototypical support that has been developed for S^2EC^2O. We point to the fact that this chapter directly builds upon the S^2EC^2O approach as introduced in the above mentioned chapters. Thus, we recommend the reader to read the preceding chapters beforehand.

During design and implementation, the focus was to provide artifacts that base on common and/or widespread technologies, so that the implementation can be extended easily. Thus, we also put emphasis on the expandability and flexibility of S^2EC^2O's prototypical implementation.

Section 9.1 gives an overview of S^2EC^2O tool's architecture. It explains from which components it is composed and how they depend on each other. Section 9.2 relates the S^2EC^2O tool to the S^2EC^2O process. It elaborates on which process tasks are accompanied by the S^2EC^2O tool and highlights technical subtleties.

The S^2EC^2O tool is designed as a *critiquing system*. As defined in [Gär16], the goal of such a critiquing system is to take a problem description and a proposed solution as input, check the solution, reveal mistakes and propose improvements [FNO+93]. In the case of S^2EC^2O, the system model and security measures are taken as input. These parts are checked against the Security Context Knowledge, eventually updated by knowledge deltas. Improvements are proposed for example as co-evolutions.

9.1 Architecture Overview

The S^2EC^2O tool has been developed as a set of plug-ins for the Eclipse platform. Eclipse was chosen for a number of reasons.

First, the Eclipse Plug-in Development Environment (PDE) provides a versatile way to develop functional units (i.e. components) independently. The Eclipse plug-in mechanism makes use of the dynamic module system OSGi (Open Services Gateway initiative). Components can be stopped, started and exchanged during run time. Apart from that, the PDE provides for instance a run time plug-in registry which can be used by plug-ins to find other plug-ins that provide certain services and to request specific ones. Loading/starting the needed plug-ins and executing

all necessary code is done automatically and realized by using standardized interfaces and plug-in meta data. One mechanism to be used for that is called *extension point*.

Second, the Eclipse Modeling Framework (EMF) provides a widespread base for model-based development and tools [SBMP08]. Several tools used for this thesis also are built upon EMF. For instance the graph transformation tool Henshin [ABJ+10, SBG+17], the tool for detecting semantic deltas, SiLift [KKOS12], and from preliminary work of the author's working group, CARiSMA [APRJ17], the platform for compliance and risk analyses, use the EMF as common base. For manipulating OWL ontologies, an OWL EMF meta-model is provided [W3Ca]. Using EMF as common base eases interoperability, as tools like Henshin only have to be built to be compatible with Ecore meta-models and then support all models for which an Ecore meta-model exists. Moreover, EMF also provides persisting (EMF-)models to XML Metadata Interchange (XMI) files ready to use.

Third, Eclipse EMF supports model-based software development. Meta-models for ESR and deltas are modeled as Ecore models and Java code to control these models. This has been used for a number of meta-models being part of S^{2}EC^{2}O.

As a prototypical implementation, we designed the S^{2}EC^{2}O tool to be modular and extensible, so that components can be exchanged or extended easily or components of the system can be used in other projects. The S^{2}EC^{2}O tool is controlled using an Eclipse IDE *View*. From the View, wizards for the S^{2}EC^{2}O processes for initialization and delta handling can be started. We give details on the GUI elements and how to use them in Section 9.2.

Figure 9.1 gives an overview of the architecture of the S^{2}EC^{2}O tool by a UML component diagram. The S^{2}EC^{2}O tool consists of the components highlighted by gray rectangles, while the other components resemble external tools and frameworks that have been used. The components, the S^{2}EC^{2}O tool consists of, are structured into four parts. These parts are shown by the rectangles numbered from 1 to 4. The dashed arrows between components resemble dependencies. All components stereotyped «Plug-in» resemble Eclipse plug-ins. In case of EMF meta-models, we omit the typical set of three plug-ins (`.model`, `.edit`, and `.editor`) and just show one generic meta-model component.

(1) The core components realize the S^{2}EC^{2}O process. `Process` is the main plug-in, containing the GUI code, integration into the Eclipse UI, and actually executing the S^{2}EC^{2}O process.

    The plug-in `Core` provides interfaces that need to be used by all other components (`ModelProvider`, for example). The `Core` plug-in does not depend on any other plug-in.

    The meta models for *delta* handling and the *Security Context Catalog* as introduced in Chapter 6 are realized as Ecore models. They are used by the core plug-ins. These plug-ins are used to extend the core to process the Security Context Catalog and delta information in a model-based manner using EMF. Thus, these models are crucial components of the core.

    `Process` also comprises two components providing essential capabilities. `Report` is used to create and manage JSON reports, both for documenting S^{2}EC^{2}O's activities and also to realize system adaptation.

    `DataModel` is a central model. It is used by all components being part of S^{2}EC^{2}O process to retrieve and provide their data.

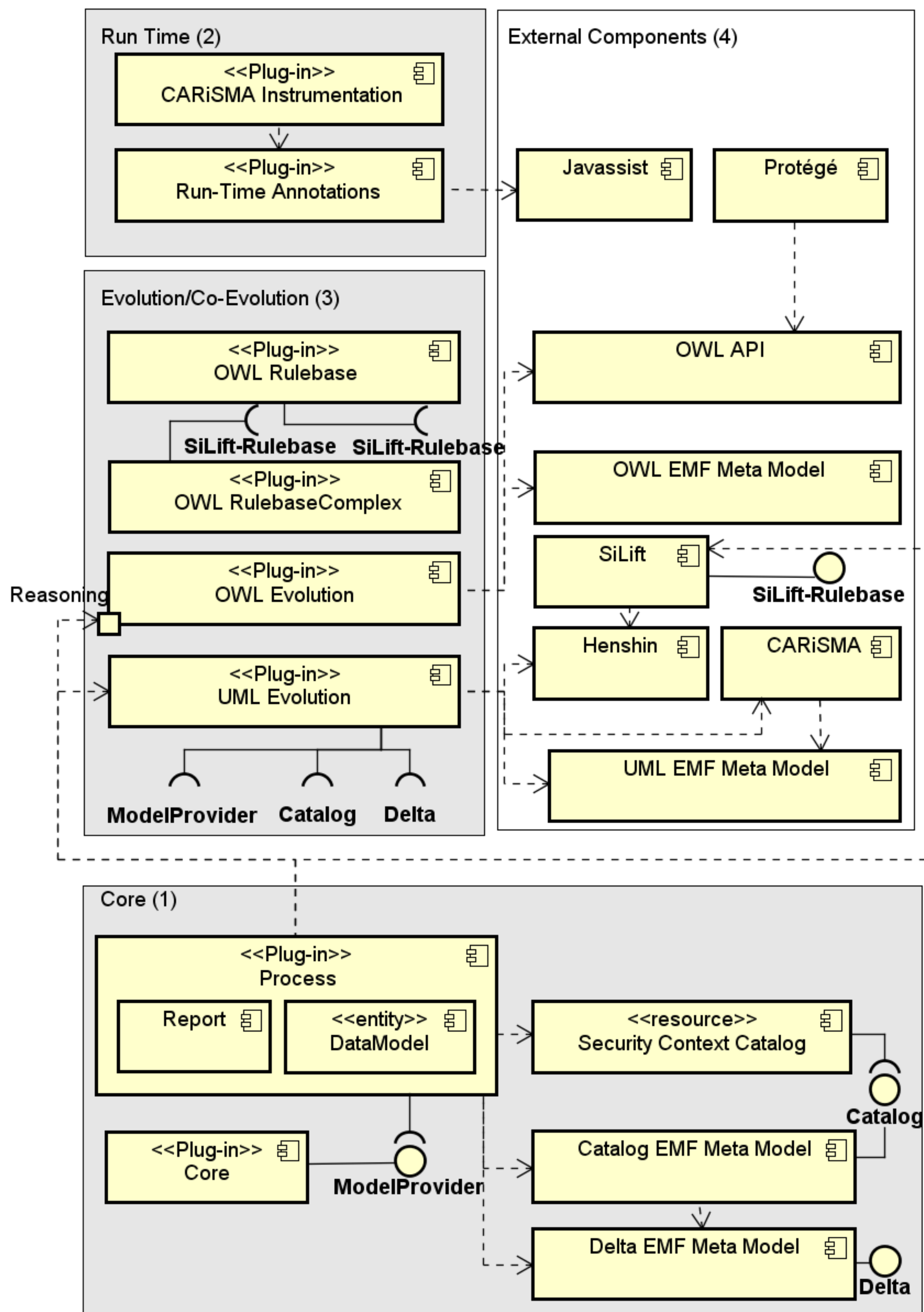


FIGURE 9.1: Overview of the S^2EC^2O tool's architecture

(2) The support for the S^{2}EC^{2}O tool's run-time phase as introduced in Chapter 7 is provided by two plug-ins. The plug-in `Run-time annotations` provides the annotations to enhance the capabilities for Java source code and thus provides support for security annotations to be read at run time. `CARiSMA Instrumentation` realizes instrumentation of source code and building a Java agent that realizes the run-time monitoring. This is done using the bytecode instrumentation framework *Javassist*.

(3) Four plug-ins have been implemented to realize analysis and application of evolution and co-evolution of OWL ontologies and UML(sec) models.

All respective dependencies and functionalities are encapsulated in subject-specific plug-ins. This lowers complexity of plug-in dependencies and eases extension of the S^{2}EC^{2}O tool and/or further development of certain components.

`OWL Evolution` contains code to analyze the Security Context Knowledge that is managed in OWL ontologies. The plug-in realizes a transformation from OWL ontologies in formats supported by the OWL API to an instance of the EMF meta model. Furthermore, reasoners as provided by the OWL API can be used. Thus, the plug-in uses the OWL API as well as the OWL EMF meta model. It provides a number of reasoning operations (summed up in the figure as a `Reasoning` port).

The plug-ins `OWL Rulebase` and `OWL RulebaseComplex` provide differencing rules to realize semantic differencing as described in Chapter 5. This is done by using SiLift and leveraging the SCK ontologies transformed into the EMF format. The names `Rulebase` and `RulebaseComplex` are used to name the rule base for syntactic, low-level differences and complex, semantic, edit operations. As the components need to be used by SiLift, they implement the respective interface, provided by SiLift.

`UML Evolution` realizes model co-evolution for UMLsec models as shown in Chapter 6. To realize analyses of UMLsec models, it makes use of the UML EMF meta model as well as supplementary services provided by the CARiSMA platform. This includes utility code and UMLsec UML profiles. By using the Security Context Catalog meta model of S^{2}EC^{2}O, the SMRs can be accessed to determine appropriate co-evolutions. Co-Evolutions are then realized and applied by leveraging the Henshin graph transformation framework. Additionally, supplementary Java code can be used to support queries as well as co-evolutions. The component `UML Evolution` provides its capabilities of accessing UML models by implementing the `ModelProvider` interface. Interaction with the Security Context Catalog (`Catalog`) and `Delta` elements is also realized by implementing the respective interfaces.

(4) The remaining elements are external tools, components, and frameworks that have been used to implement the S^{2}EC^{2}O tool. Protégé has been chosen as the tool to manage the knowledge of the SCK because it is a widespread and mature tool to design and manage ontologies. To the best of the author's knowledge, no tool with a similar reputation and active community existed for the Eclipse platform throughout development. Nevertheless, the OWL API is the technical foundation for Protégé and can also be used from plain Java programs.

The EMF meta model for OWL provides a standards-driven way to manage ontologies within the EMF.

The EMF also provides a UML meta model to process UML(sec) models easily.

The CARiSMA platform mainly is used because it provides support for the UMLsec extension, especially the UMLsec UML profiles.

Henshin, the graph transformation framework and tool, is used by SiLift. This creates synergies, because using Henshin, the user can model semantic differencing rules for ontologies as well as co-evolution actions for UMLsec models. This is realized by modeling rules that alter UML models which can be further customized. This leads to a high flexibility while the number of rules can be kept low. In addition, Henshin is used for specifying model queries.

9.1.1 System Model Design

Regarding the design of the software system that should be used with S^2EC^2O, the user needs to annotate the system model according to the obligations as shown by S^2EC^2O. For this step, we refer to the CARiSMA platform [APRJ17]. The annotation of UML models with security annotations realized with CARiSMA is also built as a set of Eclipse PDE plug-ins. It enhances the EMF- and Eclipse-based UML editor *Papyrus* to work with UML profiles realizing UMLsec extensions [Thec]. Furthermore, security checks for various security annotations are provided. To restrict effort for this thesis, we assume the class names of the system to be unambiguous, so that a distinction based on package names is not necessary.

9.1.2 Management of Security Context Knowledge

The required ontologies are modeled by the user using Protégé [Sta]. Protégé makes use of OWL API to persist ontologies, for example in OWL/XML format. During the implementation, a dependency problem arose. OWL API is built using the Maven build system. Thus, dependencies are also managed by this build system. However, Eclipse, while also supporting Maven projects, has its own dependency management system for PDE plug-ins that does not integrate with Maven projects. For the S^2EC^2O prototype, this was solved by building OWL API externally and linking it into the S^2EC^2O tool.

By using Protégé as a ready-to-use workbench for modeling and managing OWL ontologies, implementing a respective tool could be avoided. As Protégé is built also upon the widespread and mature OWL API, the ontologies created with it can import data from and export to various formats and sources.

OWL API has its own meta model of ontologies. In preliminary work of the author's working group, a transformation from ontologies corresponding the OWL API meta model to ontologies corresponding the Ecore OWL meta model provided by W3C [W3Ca] has been implemented. This way, both standard tools and frameworks can be used for managing and processing ontologies inside and outside of the Eclipse platform.

9.1.3 Employing Reasoners via API

At the time of writing, we did not succeed in using reasoners via API, to process their result in S^2EC^2O. Thus, the step of triggering a reasoner from the S^2EC^2O tool could not be realized due to technical difficulties. For example, an ontology being

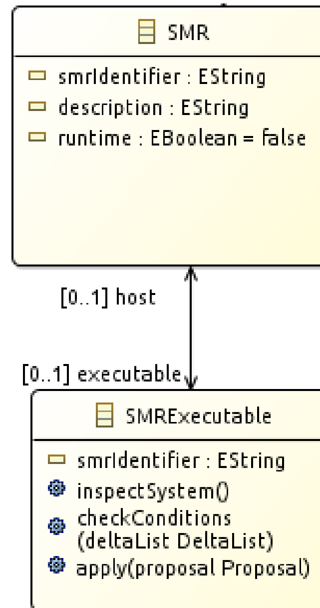


FIGURE 9.2: Excerpt from Figure 6.2 (on page 66) to illustrate relation between `SMR` and `SMRExecutable` classes

inconsistent was reasoned to be consistent by the tested reasoners. Another reasoner aborted when the inconsistency was detected and let no possibility of getting explanations. Apart from that, the same reasoners, when used as they are shipped with Protégé, produced the results as expected.

Thus, the reasoning results currently need to be generated manually. In Chapter 10, we will show reasoning results as screenshots of Protégé.

The S²EC²O tool and the Security Context Catalog currently already uses the elements for ontology inference and inconsistency explanations as provided by the delta information meta model (see Figure 6.4 on page 70). Furthermore, we provide two text input dialogs as part of the S²EC²O tool's delta wizard. The user can use these to provide expressions in Manchester Syntax, representing inconsistency explanations or inferred facts, respectively.

9.1.4 Realizing specific Implementations for Ecore Objects

While designing the S²EC²O tool, a challenge concerning a gap between Ecore and Java code arose. The Security Context Catalog, as we introduced it in Section 6.2.1 (on page 65), is designed as an Ecore meta-model to take advantage of the Eclipse EMF, for example in terms of persisting models. Figure 9.2 shows an excerpt of the Security Context Catalog's meta model. Consequently, the meta model features a class for as Security Maintenance Rule. When executing the S²EC²O process, SMRs are important for the operative parts. For instance, model queries need to be executed and co-evolutions need to be applied.

To provide high flexibility and limit the implementation effort by not introducing additional mechanisms, the operative parts of SMRs are implemented as Java code. This is especially necessary because model queries and co-evolutions carried out by Henshin need to access Henshin via its Java API.

This gets difficult because, by design, an *operation* specified in an Ecore class can only have one implementation. As the class `SMR` is used as a template for SMRs, every SMR instance needs its specific implementation of the methods

`inspectSystem()`, `checkConditions()` and `apply()`. We solved this issue as follows: We split `SMR` into the model-related part and the implementation-specific part, called `SMRExecutable`. The necessary implementations are done using ordinary Java classes that extend the *`Impl`* class as generated by EMF, in this case `SMRExecutableImpl`. The additional class `SMRExecutable` is introduced to not disrupt object relations to other EMF objects and persisting the Security Context Catalog. Persisting ordinary Java objects would be conceivable but only with loss of Ecore features. Trying to persist an Ecore model without these objects would also cause problems because the associations would be tied to the rest of the Ecore model loaded at the time of persisting the model.

This challenge was solved by adding the `smrIdentifier` as additional property to `SMRExecutable`. In the S^{2}EC^{2}O tool, the whole Security Context Catalog, excluding SMR executables, can be loaded, managed and saved easily as designed in EMF. A SMR can be associated with one SMR executable by using the same `smrIdentifier` string. `SMRExecutable` classes need to be provided and somehow *bound* to the SMR elements in of the loaded Security Context Catalog. To provide flexibility here, we defined an Eclipse extension point `ssecco.esr.model.smr`. For all SMR executables, we specify such extensions, so that the S^{2}EC^{2}O tool can query all available implementations easily from the Eclipse registry at run time.

When the Security Context Catalog is loaded, the S^{2}EC^{2}O tool's core gathers and instantiates all available SMR executables. Using the `smrIdentifier` string, all SMR executables are bound to their respective SMRs EMF instance by setting the associations correctly. Therefore, all Ecore features remain functional for the Security Context Catalog, serialization of Java objects can be avoided, and individual implementations for the SMR operations are realized while providing a flexible class allocation handled by the Eclipse extension point registry.

9.2 Relation of S^{2}EC^{2}O's Initialization Process and Realized Prototype Artifacts

In the preceding section, we gave an overview of the S^{2}EC^{2}O tool's architecture. It shows which components have been implemented and how they depend on each other. It also summarizes which external tools and frameworks have been chosen to realize the prototype.

This section puts emphasis on the S^{2}EC^{2}O process shown in Section 2.2.3 (on page 15) and Section 2.2.3 (on page 17) and how it is to be executed with the aid of the S^{2}EC^{2}O tool. We support the discussion by figures showing the involved parts of the processes and also the architecture as introduced in Figure 9.1.

9.2.1 Initialization Process: Make System S^{2}EC^{2}O aware

The initialization process is started by the user. The process is accompanied by the S^{2}EC^{2}O tool using a wizard. It covers the first five tasks of the initialization process as shown in Figure 9.3: *Choose desired ESRs from catalog*, *Propose design time checks*, *Query SCK for details*, *Complete sec. checks by choices*, and *Provide additional (system) knowledge*. Figure 9.4 shows an overview of the involved architecture components respectively.

The Security Context Catalog is loaded and used to display available ESRs to the user. As a first step, the user has to decide, which ESRs are to be followed. Figure 9.5 shows a screenshot of the respective wizard page. Available ESRs of the catalog

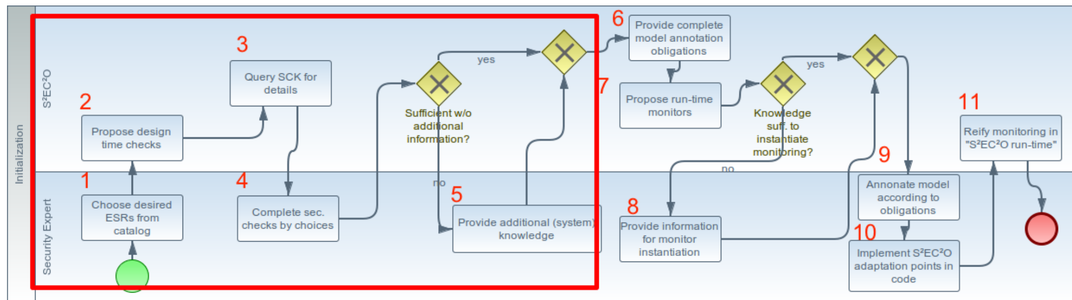


FIGURE 9.3: S^2EC^2O initialization process: involved tasks 1-5

are shown. By traversing the associations of the selected ESRs, security properties requested by given ESRs are determined. The security properties that are required by the respective ESR are shown, as well as the description of the selected ESR.

The S^2EC^2O tool informs the user about necessary security properties by displaying them in a list. The steps described up to this point correspond to the tasks 1 to 2 of the initialization process (*Choose desired ESRs from catalog* and *Propose design time checks*).

The user is then obliged to complete the security checks by choosing details. For example, a specific encryption algorithm needs to be selected from a set of appropriate ones.

Figure 9.6, for instance, shows a screenshot of the initialization wizard, where the user needs to select an encryption algorithm, because more than one algorithm modeled in the SCK fulfills the requirements as specified by the Security Context Catalog. This relates to tasks 3 and 4 of the initialization process (*Query SCK for details*, and *Complete security checks by choices*). After the user has completed all inputs, selecting the button *Check data to proceed* issues a consistency check of the selected or given input. When the check does not detect any problems, the user can proceed to the next wizard page.

The SCK is accessed using the OWL component that provides a port with reasoning functionalities. In this case, SPARQL is used as query language.

According to task 5 of the initialization process (*Provide additional (system) knowledge*), the user may be required to provide additional system knowledge. For example, a mapping of system classes to concepts in the SCK may be required to be provided or completed. This needs to be done by the user, but is guided by the wizard using instructions.

Currently, the `checkIdentifier` property of security properties as defined in the Security Context Catalog meta model (see Figure 6.2 on page 66) is not evaluated by the S^2EC^2O tool. But, this property fosters future tool development by easing a technical integration of the S^2EC^2O tool with OWL reasoning, CARiSMA and other tools for design time security checks.

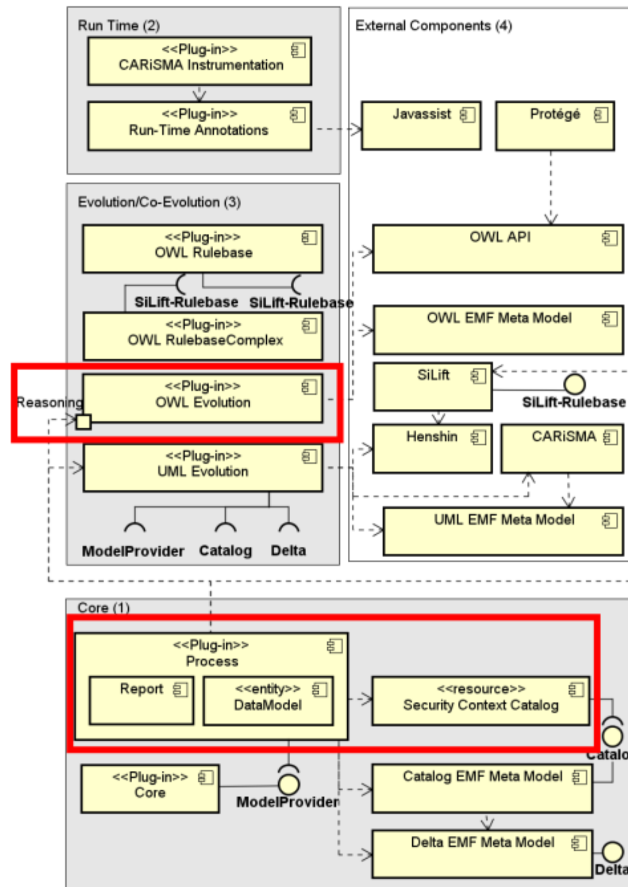


FIGURE 9.4: Components of the S²EC²O tool relevant for initialization of a S²EC²O aware system

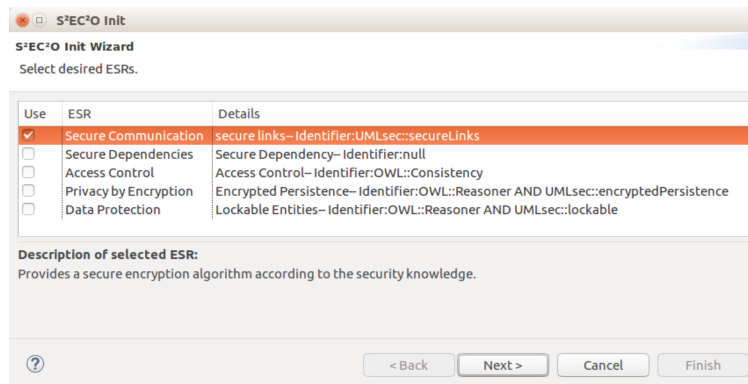


FIGURE 9.5: Begin of initialization wizard: Choice of ESRs

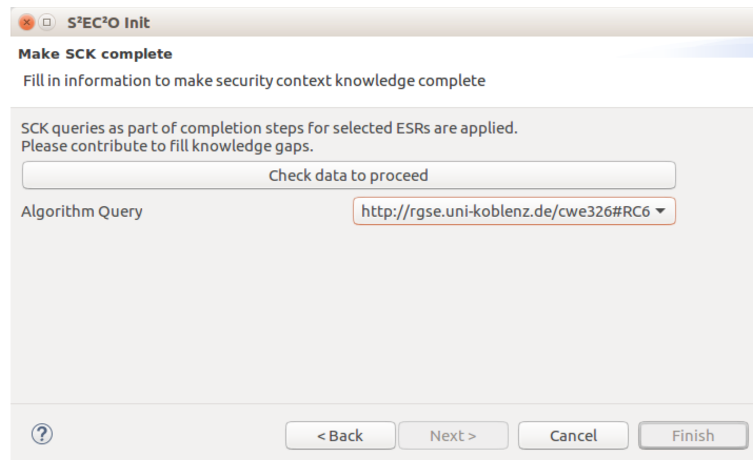


FIGURE 9.6: S^2EC^2O initialization wizard: choose details from SCK

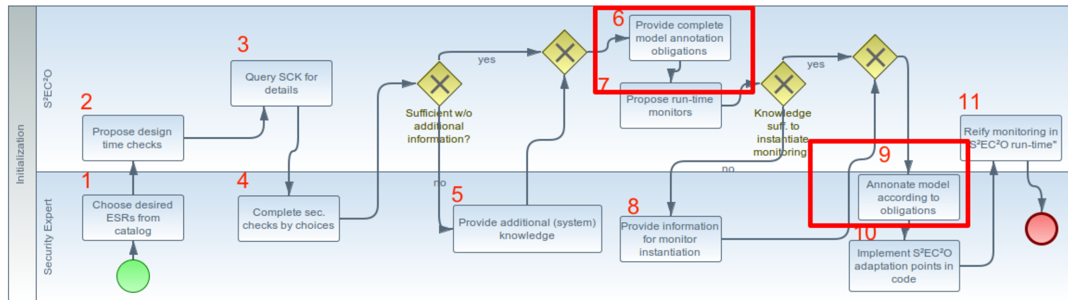


FIGURE 9.7: S^2EC^2O initialization process: involved tasks 6 and 9

9.2.2 Initialization Process: Initial Compliance

After the Essential Security Requirements have been selected and completed by the user, he needs to implement them in the system model design. The wizard supports this step by listing all security properties. This corresponds to the tasks 6 and 9 of the process (*Provide complete model annotation obligations* and *Annotate model according to obligations*). The respective tasks are highlighted in Figure 9.7. Task 6 is also realized by traversing the Security Context Catalog.

The S^2EC^2O tool then determines concrete obligations for model annotations that need to be applied to the system model. The annotation of UML models with security annotations is supported by CARiSMA [APRJ17], as far as checks are available.

In parallel, the user's selections are logged using the report component and persisted as JSON object later-on.

9.2.3 Initialization Process: Run-Time Monitoring

Up to now, S^2EC^2O tool's initialization wizard has covered the tasks of the initialization process relevant for the design time. In the succeeding steps, the wizard treats what is relevant for the run time.

The S^2EC^2O tool determines matching run-time monitors for the selected ESRs. This again is realized by traversing the associations of the Security Context Catalog. Each ESR may be associated to a set of run-time monitors (task 7, *Propose run-time monitors*). Figure 9.8 shows the initialization process of S^2EC^2O with relevant tasks highlighted. With regard to task 8 (*Provide information for monitor instantiation*), it may be necessary that the user needs to provide additional details of the system so that security properties can be instantiated and/or implemented correctly.

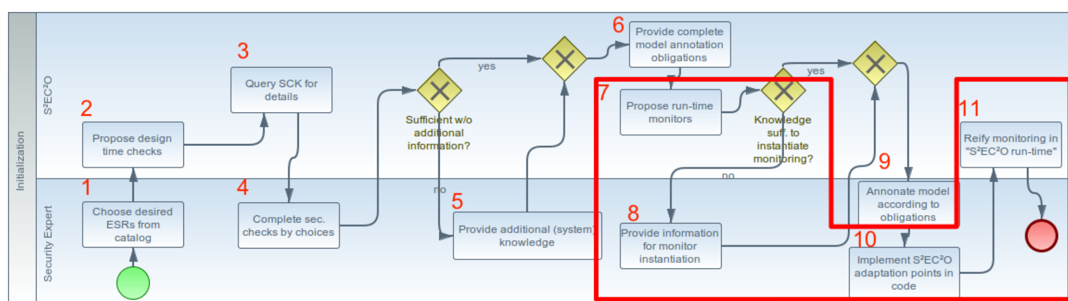


FIGURE 9.8: S^2EC^2O initialization process: involved tasks 7-11 excluding 9

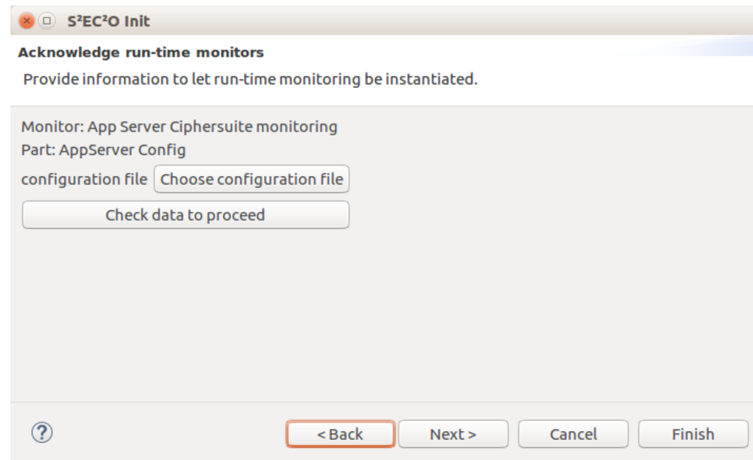


FIGURE 9.9: S²EC²O initialization wizard: provide information of the system to instantiate run-time monitor

This is realized by the wizard asking the user to provide that information. Figure 9.9, for example, shows an example of the wizard where the user needs to provide link to the configuration file of an application server, so that the run-time monitor *App Server Ciphersuite monitoring* can work correctly.

Task 7 (*Propose run-time monitors*) is also realized using the S²EC²O tool's OWL component and its reasoning operations. Information that needs to be provided additionally is modeled by using the `Completion` objects of the Security Context Catalog.

After this step, the S²EC²O tool's initialization wizard is finished and a report is written. It is persisted as a JSON object, which can be read by humans and also eases sophisticated processing by subsequent tool support.

```
1  {
2    "DocumentType": "SSECCO Report",
3    "Timestamp": "2019-01-11 19:16:23.281",
4    "Type": "Init Wizard run",
5    "SelectedESRs": [
6      {
7        "SelectedESR": "Secure Encryption",
8        "SecurityProperties": [
9          {
10           "PropertyName": "secure links",
11           "Completions": [
12             {
13               "Completion": "Algorithm Query",
14               "Artifacts": []
15             }
16           ]
17         }
18       ]
19     }
20   ]
21 }
```

LISTING 9.1: S²EC²O report logging execution of the initialization process wizard

Listing 9.1 shows an example of a report after execution of S²EC²O tool's initialization wizard. It follows the same base structure as the JSON files we introduced for the run-time adaptation approach in Section 8.1. In this case, the user selected the ESR `Secure Encryption`. For history and archiving reasons, the Security Context Catalog entries are dumped completely into the report. This fosters replicability,

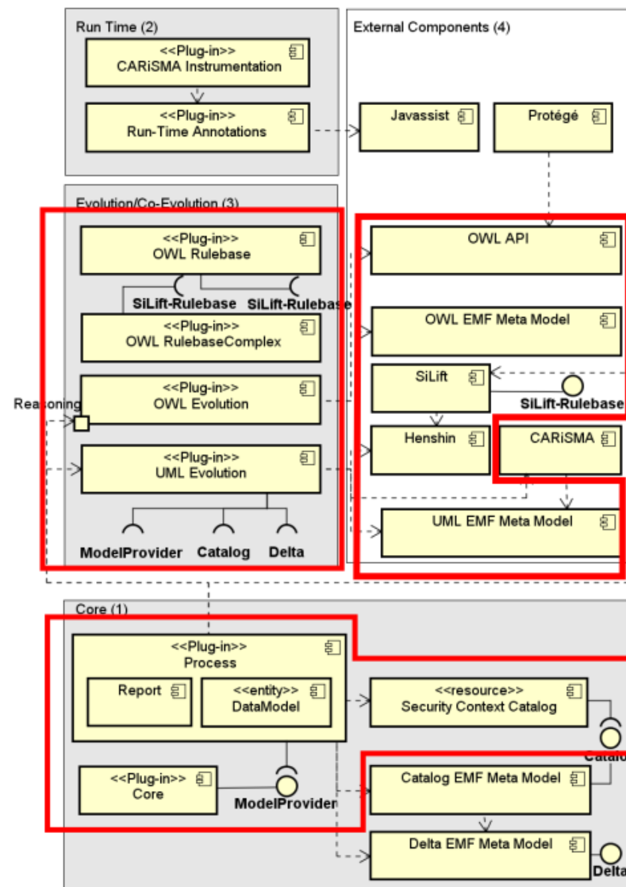


FIGURE 9.10: Components of the S²EC²O tool relevant for delta handling process

as the Security Context Catalog may also be subject to change (as we will discuss in Section 12.3).

For task 10 shown in Figure 9.8 (*Implement S²EC²O adaptation points in code*), no sophisticated tool support has been implemented. We provide Java annotations (introduced in Section 7.1 and Chapter 8) so that the user is able to add run-time annotations and adaptation points to the source code.

For the final task of the initialization process (task 11, *Reify monitoring in S²EC²O run-time*), regarding reification of the run-time monitoring, the user needs to run the system accompanied by the S²EC²O run-time. This is accomplished by starting the JVM with providing the Java agent (S²EC²O run time) as a parameter.

9.3 Relation of S²EC²O's Delta Process and Realized Prototype Artifacts

This section relates to S²EC²O's delta handling process we introduced Section 2.2.3 (on page 17) and how it is to be executed with the aid of the S²EC²O tool.

9.3.1 Delta Process: Determine Context Evolution from SCK

Figure 9.10 shows the components of S²EC²O tool's architecture which are used to analyze the SCK. In what follows, we will refer to the components of the architecture

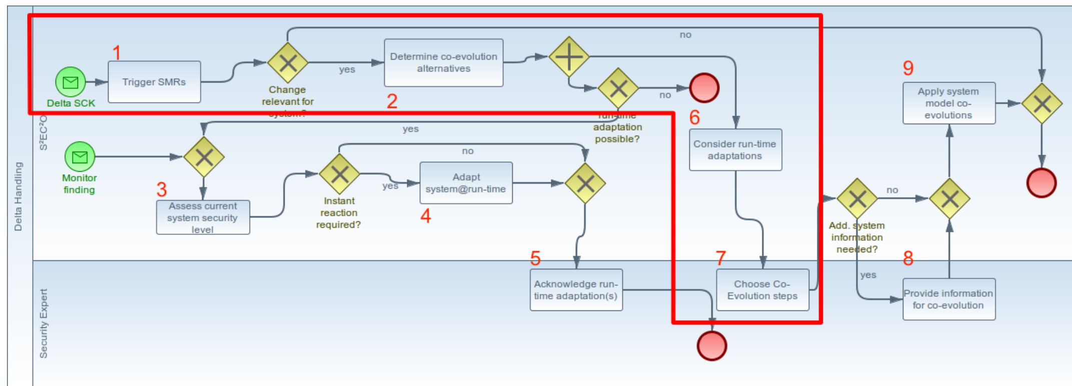


FIGURE 9.11: S^{2}EC^{2}O delta handling: involved tasks 1, 2, 6, and 7

when we elaborate on the according process tasks.

The delta handling process of S^{2}EC^{2}O has less user interaction than the initialization process. At the time of writing, the prototype supports the steps relevant for the design-time co-evolution.

The pseudo code for *delta handling* we introduced in Section 6.5 (on page 73) describes the behavior that is currently implemented. Figure 9.11 highlights the process tasks relevant for these steps (*Trigger SMRs*, *Determine co-evolution alternatives*, *Consider run-time adaptations*, and *Choose co-evolution steps*).

The delta handling process of S^{2}EC^{2}O is also implemented as a wizard. This wizard can be initiated by the user. Firstly, the delta information is collected by executing ontology reasoning and determining semantic differences between different versions of the SCK using *SiLift* and *Henshin*. This results in a list of deltas according to the *delta meta model* we introduced in Section 6.3. This behavior adheres to the process start event `Delta SCK`.

The available SMRs are gathered using traversing the *Security Context Catalog* (process task 1, *Trigger SMRs*). The run-time variants are instantiated as described in Section 9.1.4. The SMRs then can access the Security Context Catalog and query the system model to determine possible co-evolutions and prepare a set of proposals for the user (process task 2, *Determine co-evolution alternatives*). When a SMR considers a run-time adaptation as necessary, it can carry it out immediately (process step 6, *Consider run-time adaptations*).

After that, the user needs to make choices about the co-evolution alternatives (proposals) as determined by the SMRs (step 7 of Figure 9.11, *Choose co-evolution steps*).

Figure 9.12 shows the step of selecting co-evolution proposals in the S^{2}EC^{2}O tool delta wizard. The wizard page informs about all detected deltas. In this case, the delta *addEncryptionThreat* was detected two times. The user then needs to choose one co-evolution alternative for each SMR. Every co-evolution that is to be applied has to be selected using the check box. Whenever an alternative from the drop-down list is selected, a description which problem is going to be co-evolved how is displayed. In this example, an encryption algorithm `AES-256` is to be replaced by `RC6`.

9.3.2 Delta Process: Apply Co-Evolution Steps

After the user has finished making his choice, the wizard can proceed to the next step. Figure 9.13 emphasizes on the tasks 8 and 9 of the delta process, relevant for

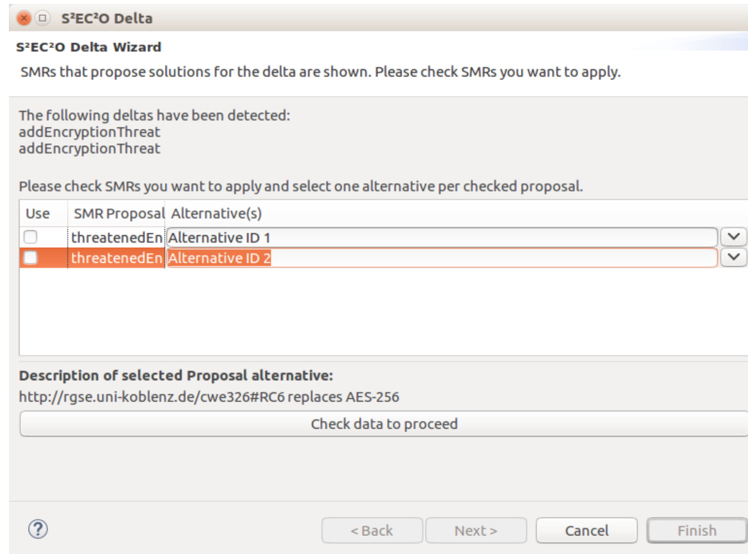


FIGURE 9.12: S^{2}EC^{2}O Delta Wizard: choice of alternatives

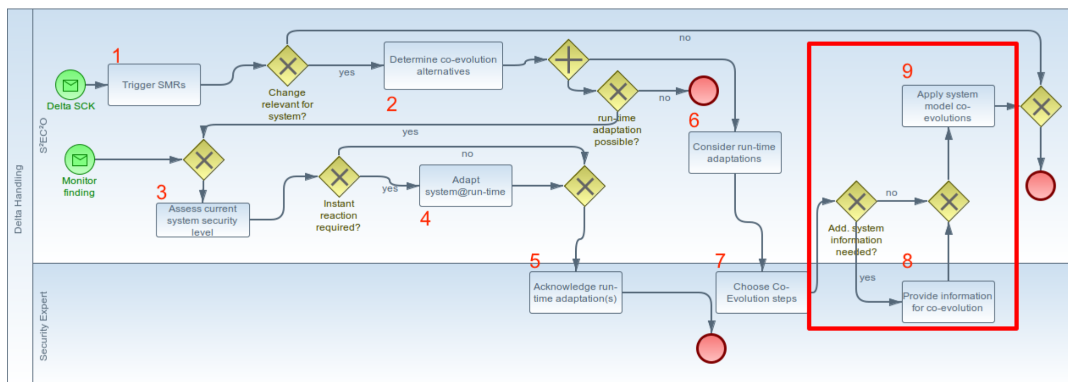


FIGURE 9.13: S^{2}EC^{2}O delta handling process: involved tasks 8 and 9

choosing and applying co-evolutions to the system design (*Provide information for co-evolution*, *Apply system model co-evolutions*).

The S²EC²O tool checks if the co-evolutions selected by the user require additional information. This is realized by analyzing the respective SMR's `Completion` elements. The S²EC²O tool eventually asks the user to provide information as required by the SMR completion elements. The dialog for completion of SMR information works analogous to the respective dialog of the initialization process shown in Figure 9.9 (on page 120). Access of the Security Context Catalog is completely handled by the components building the core of the S²EC²O tool (see Figure 9.10 on page 121).

After that step, the wizard's interaction with the user is finished. The SMRs are applied by calling all `apply()` methods and thus triggering *Henshin* model co-evolving transformation rules, reflective model code, and further methods as introduced in Chapter 6. The actions to be taken are defined in Security Context Catalog and, more precisely, in the implementations of the `SMRExecutable` classes. A report of the activities is written analogous to the report for the initialization wizard shown in Listing 9.1 (on page 120). For querying and altering the system model, mainly the component `UML Evolution` is involved. It is responsible for carrying out the code and graph transformations that effectuate the co-evolution.

As we already discussed, the S²EC²O tool currently offers no support for the run-time adaptation. We will discuss this in detail in Section 9.3.4.

9.3.3 Delta Process: Generating Run-time Findings

In the preceding sections, we discussed the case of *Delta SCK* as triggering event for the delta process. A second source for delta information is run-time monitoring, as highlighted in Figure 9.14. In this section, we put emphasis on how generation of run-time findings is implemented using run-time monitoring by the S²EC²O run time. We introduced the S²EC²O run time in Chapter 7.

Figure 9.15 shows the components of the S²EC²O tool relevant for run-time monitoring. To instantiate the run-time monitoring with design-time insights, the source code of the application needs to be annotated with security properties and appropriate countermeasures. Generation of monitored call sequences as sequence diagrams and the generation of missing model elements which appear at run time is supported, too.

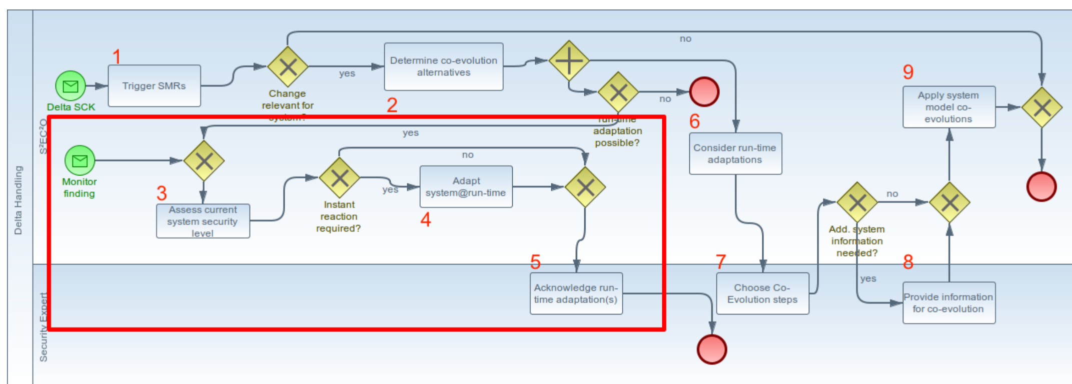


FIGURE 9.14: S²EC²O delta handling process: involved tasks 3, 4, and 5

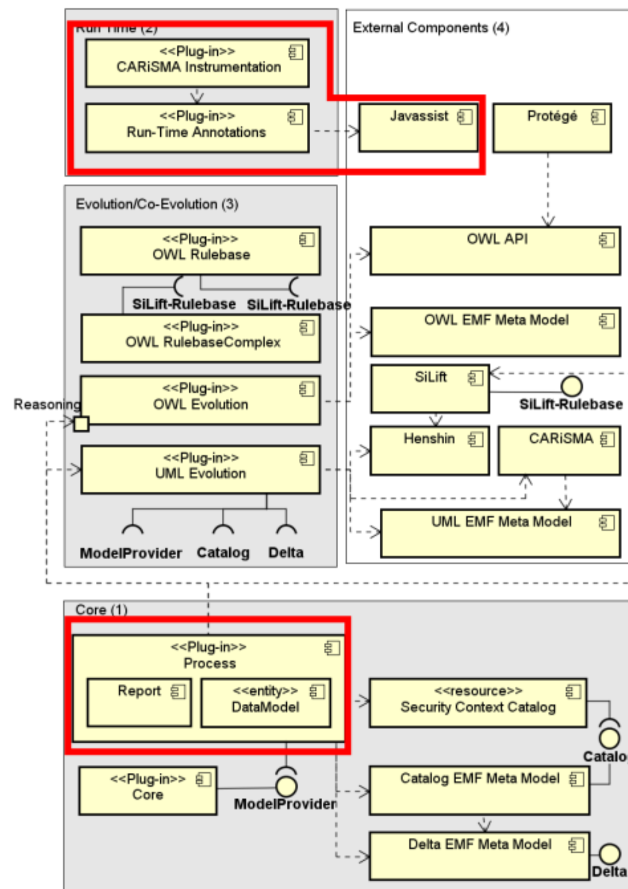


FIGURE 9.15: Components of the S²EC²O tool relevant for run-time monitoring

The source code annotations as specified in Section 7.2.2 are implemented as Java annotations and encapsulated in a respective component (`Run-time annotations`). Currently, annotations for «secure dependency» are supported.

At the time of writing, synchronizing model and code including their annotations is not supported by the S^{2}EC^{2}O tool. Nevertheless, by applying TGGs in either direction (from model to code or vice versa), it is possible to have this conversion in a one-shot manner.

To realize monitoring, we make use of bytecode instrumentation. To get access to the running system, we make use of a Java agent. *Java agent* is a concept provided by the official Java API and documented in the package `java.lang.instrument` [Ora]. To let the S^{2}EC^{2}O monitoring code be injected into existing classes of the application, we use the bytecode manipulation framework Javassist [Chi]. We implemented a Java Agent that can be called via the `javaagent` command line option of the Java Virtual Machine (JVM).

The JVM calls our agent whenever a class is to be loaded. The agent then transforms the bytecode of the class by injecting the code as shown as pseudo code in the Listings 7.3, 7.4, and 7.5 (see page 85) necessary to keep track of the call stack, issuing checks of the secure dependency conditions at appropriate times, and also produce additional report data to realize a model adaption. The run-time protocol is realized as introduced in Section 7.4.1 (on page 91). A JSON object containing the call trace is serialized. Detected security violations are documented in the file, too. This is also realized by the *Instrumentation* component in S^{2}EC^{2}O's architecture.

Thus, the run-time agent is able to inject the necessary S^{2}EC^{2}O monitoring code into all relevant methods at load time of the respective class. Static checking of potential malicious field accesses is also done when the class is loaded. The realized analysis of «secure dependency» is a hybrid analysis not depending on local availability of all classes. Since the agent is also called on classes loaded dynamically, from the Internet, for example.

Detecting System Evolution Automatically

There may be associations in the code that are not covered and cannot be detected statically. This especially applies to dynamic behavior introduced by libraries and reflective calls. While a JVM is monitored during execution of the target program (see step 4 in Figure 7.3 on page 84), the agent implementation keeps track of every method which has been entered and not exited yet using a stack for each thread.

To present the call stack in a graphical manner, the trace written by the agent can be used to create a sequence diagram and thus provide run-time feedback into the UML model (see step 5 of Figure 7.3). As the Java agent is able of keeping track of every method and field that is accessed, we are able to check continuously if a call edge detected in the run-time monitor that has corresponding elements in the model. If this is the case, then the elements can directly be related to the corresponding model elements. If not, the tool can feed this information into the model by adding respective elements using JSON objects.

9.3.4 Support for Run-Time Adaptation

At the time of writing, the S^{2}EC^{2}O tool does not cover run-time adaption in a way that S^{2}EC^{2}O is able to actively control the system's behavior. Thus, supporting tasks 3 and 4, as highlighted in Figure 9.14 (on page 124), namely *Assess current system security level* and *Adapt system@run-time* is not yet implemented. However, support for

task 5 (*Acknowledge run-time adaptation(s)*) is currently implemented to that extent that reports, for example coming from the S^2EC^2O run time, have to be acknowledged by the user. This is realized as an additional page of the delta wizard that displays the content of all new (i.e. non-*ACKed*) reports to the user.

The remaining work to realize the run-time adaption would require to implement the S^2EC^2O service component and the extensions proposed for S^2EC^2O run time as described conceptually in Section 8.1. However, the monitoring part as described in Section 9.3.3 is able to instrument code of the system. Hence, it is able to prevent violations of security properties (in this case, «secure dependency»). Alternative behavior can be also defined. But this is only possible as part of the source code and in a static manner, not updateable during run time.

This is realized as follows: The developer annotates the system model according to the obligations given by S^2EC^2O. We acknowledge the fact that often a system implementation is not on the same level of abstraction as the model. Thus, as connecting link we use class names.

The current implementation, supporting «secure dependency», also supports definition of countermeasures using the `@Countermeasure` source-code annotation. The `CARiSMA instrumentation` plug-in then instruments source code of the system under consideration.

If a violation of the security property is detected by the instrumented code during run time, the respective countermeasure actions are executed automatically. The Java Agent writes reports about detected security violations into JSON reports as introduced in Section 7.4.1.

9.3.5 Review of the Prototype's Unique Features

In this section, we review the features the S^2EC^2O tool provides and thus realizes S^2EC^2O. The initialization- and delta-handling process of S^2EC^2O both are supported by a wizard that guides the user through the process steps.

The S^2EC^2O tool supports management of the SCK by using the OWL API to access ontologies created with Protégé. Thus, a widespread workbench can be reused, avoiding additional implementation overhead. The S^2EC^2O tool is able to analyze the knowledge base stored as part of the layered ontologies in terms of leveraging reasoning and also by making use of semantic diffs as calculated using the SiLift approach.

The S^2EC^2O tool is able to query models using, for example, graph transformation techniques as provided by Henshin. Moreover, SMRs are built on native Java code and thus can be implemented in a highly flexible manner. The wizards the S^2EC^2O tool offers make use of the Security Context Catalog, it is thus available to the user in a convenient way. The SCK is also used in the processes by issuing SPARQL queries. Design-time model co-evolution is supported by issuing graph transformations. Thus, the S^2EC^2O tool realizes S^2EC^2O, so that the user is able to co-evolve a system at design time.

Moreover, the prototype also supports run-time monitoring. We provide Java annotations which can be used to accompany a system's source code at the one side, and matching with model elements and the UMLsec annotations at the other side. We are able to instrument existing application code with our monitoring code. The user is able to specify countermeasures to be carried out immediately in case the monitoring detects a security violation. Moreover, round-trip engineering is supported by the S^2EC^2O tool by writing trace information that can be used to adapt and extend the system model.

In the subsequent chapter, we will exemplify the S^2EC^2O tool's features by carrying out a case study.

Chapter 10

Case Study: Applying S^2EC^2O to iTrust

In the previous chapters, we introduced and discussed S^2EC^2O theoretically as well as its foundations. In Chapter 9, we introduced the S^2EC^2O tool, a prototypical implementation of S^2EC^2O.

In this chapter, we especially show the applicability of S^2EC^2O and the S^2EC^2O tool by presenting a case study. In the study, we apply S^2EC^2O to a medical information system called *iTrust*. The system, its relevance for this thesis and its basic architecture are introduced in Section 10.1.

We point out the fact that this chapter directly relates to the preceding chapter. Thus, the reader is recommended to read it upfront.

S^2EC^2O focuses on co-evolution of software where a system's security is put at stake by evolution of its context. Section 10.2 elaborates on possible context evolution sources that have been taken into account for this chapter.

The context evolutions may lead to security vulnerabilities. Section 10.3 gives a classification of them regarding their relevance.

Section 10.4 introduces the structure of the following sections. We will cover five examples, each of them in a separate section.

The goal of this chapter is to study the applicability and the effectiveness of S^2EC^2O with regard to the iTrust medical information system.

10.1 Introduction to iTrust

The target system for this case study targets is the open source healthcare system *iTrust*. It has already been used in preliminary work [BSG+18, Gär16, BGR+15, GRB+14]. iTrust is a role-based medical information system implemented in Java for the Java Enterprise Edition (JavaEE) platform. Its purpose is to provide patients with medical information, let medical staff organize their daily work, and provide a messaging system so that all users can communicate with each other.

The iTrust project was initiated at North Carolina State University and is currently maintained by the Realsearch Research Group [MSW12]. It is a fully operational system. Its development artifacts, in particular, requirements and code, are publicly available. Its use cases are documented in natural language (we use version 27 of the requirements and the corresponding code version 21 for our study), some of them addressing privacy and security issues regarding personal and medical data. At the time of writing, iTrust was moved from a completely open structure to a Gitlab system available to the students of the involved university. However, getting more recent artifacts for this thesis on request was possible. At the time of writing, the artifacts before the move to Gitlab were still available [MSW].

As a large system maintained over a long period of time, iTrust is a well-suited example of a long-living system with substantial practical relevance: First, it underwent organizational changes. Being in use since 2004, responsible architects and developers changed several times, and functional requirements have also changed.

Second, it was subject to substantial functionality changes. Over the years, new use cases have been implemented and others have been removed, according to continuously changing requirements. For example, in requirements version 27, the system is implemented based on 39 use cases, while the overall sum of documented use cases is 79 (see [MSW]).

As a healthcare system, iTrust is particularly prone to privacy and security issues [Bow13]. All of these properties are also typical to a long-living system used in industry.

10.1.1 Architecture of iTrust

iTrust is a role-based system, having a hierarchical role model. In total 11 roles are defined, while the case study will focus on the following six roles (role descriptions adapted from [MSW]):

Health Care Personnel (HCP) All of designated licensed health care professionals, licensed health care professionals, and unlicensed authorized personnel, as defined below.

Patient When an American infant is born or a foreigner requests medical care, each is assigned a medical identification number and password. Then, this person's electronic records are accessible via the iTrust Medical Records system. Every kind of operation, office visit, treatment, diagnostic tests, etc., is targeted against a patient. A patient can log into the system, have access to all data that is relevant for treatment.

Licensed Health Care Professional (LHCP) A licensed health care professional that is allowed by a particular patient to view all approved medical records. In general, a patient does not know this non-designated health care professional, such as an emergency room doctor, and the set of approved records may be smaller than that granted to a designated licensed health care professional.

Designated Licensed Health Care Professional (DLHCP) A licensed health care professional that is allowed by a particular patient to view all approved medical records. Any LHCP can be a DLHCP to some patients (with whom he/she has an established relationship) and an LHCP to others (whom he/she has never/rarely seen before).

Unlicensed Authorized Personnel (UAP) A health care worker such as a medical secretary, case manager, care coordinator, or other authorized clerical-type personnel. An unlicensed personnel can enter and edit demographic information, diagnosis, office visit notes and other medical information, and can view records.

Lab Technician (LT) A clinical worker that runs diagnostic tests on samples gathered from patients during office visits. The lab technician can specialize in blood work, tissue work, or general.

iTrust is built as a JavaEE application and based on JavaServer Pages (JSP) to provide a web interface. Regarding the WebRoot, in the root directory, a login page

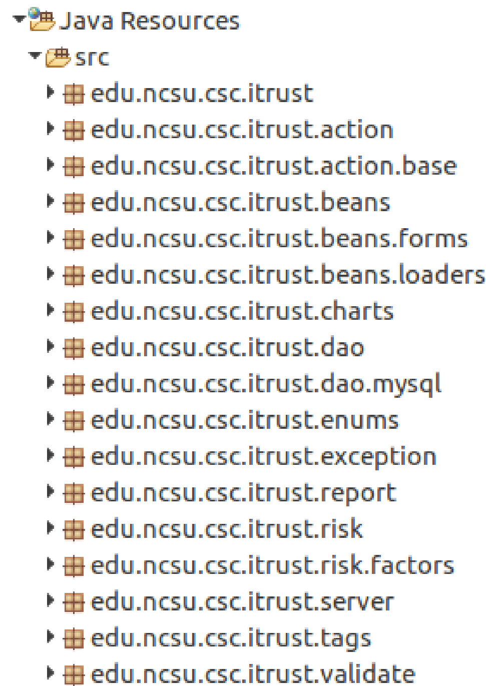


FIGURE 10.1: Package tree of iTrust Java classes

is provided. After a user is logged in, he is redirected into a subtree (`auth`). In this part, a number of JSPs are stored to provide basic functionality available for every authenticated user regardless of his role. For every role, on a coarse-grained scale, further subtrees exist (`hcp`, `patient`, and `hcp-uap`, for example). Wherever fine-grained permissions need to be checked (for instance, if a given LHCP is the designated LHCP of a given patient), this is done using Java code.

The data of the system is completely stored in a SQL database, in this case a MySQL database.

Figure 10.1 shows an overview over the package tree of iTrust regarding the Java source code, our case study will focus on. The package names have been chosen according to the features the respective classes are used for.

We briefly highlight the packages relevant for the case study.

- `action.*`: These classes provide content for JSP for every data-relevant actions. For instance, adding or editing prescriptions of a patient.

- `beans.*`: To ease handling of data, all complex types that may be persisted in similar tables or the same table, Java beans are used. For instance, `PatientBean`, `HospitalBean`, and `PrescriptionBean` exist.

- `dao.*`: while the package `edu.ncsu.csc.itrust.dao` just contains some skeleton classes and interfaces, the package `edu.ncsu.csc.itrust.dao.mysql` contains 60 classes representing Database Access Objects (DAOs). These interface with the Java beans (where applicable) and prepare as well as perform every query of the MySQL to query and alter data of the system. For example, `PatientDAO`, `PrescriptionsDAO`, and `DiagnosesDAO` are provided.

- `validate.*`: Barely 40 validator classes, such as `PatientValidator`, provide functionality to check if the data processes makes sense. For instance, syntax of dates, and phone numbers are checked in `PatientValidator`.

10.1.2 Metrics of iTrust

To quantify the size of iTrust, we show a number of typical metrics, calculated from the iTrust source code. We used the metrics tool *SourceMeter* in version 8.2.0 [FLS+14]. We use the wording and abbreviations as given by SourceMeter. Total logical lines of code is defined as the number of non-empty and non-comment code lines.

Metric	Value
Total Lines of Code (TLOC)	116 789
Total Logical Lines of Code (TLLOC)	77 502
Total Number of Classes (TNCL)	938
Total Number of Methods (TNM)	6166

TABLE 10.1: Metrics calculated for the iTrust system

Acquiring the System Model

Since the iTrust project does not provide any models, the design models were obtained in a reverse engineering process. A UML model was reverse engineered from the source code by using the TGG tool eMoflon [eDT] and a set of TGG rules that have been created for a work of Leblebici et al. [LAS17]. The resulting model turned into class diagrams. The reverse engineered model of iTrust will be used for the case study. We will show excerpts of the model to realize the case study.

10.2 Introduction to Security-Related Context evolutions

For the application examples, we consider three types of context evolution which we will shortly introduce here. These are: changes to privacy laws, encryption algorithm exploit, and trust in external libraries.

Changes to Privacy Laws

A potential source for context evolution is a law change regarding the German Federal Data Protection Act (in German: Bundesdatenschutzgesetz, BDSG [Bun05]). Privacy regulations change regularly to reflect changes in the behavior and technical possibilities to work with data as well as new juristic interpretations. The German privacy protection act has been altered several times to fulfill the European directive [EU 95]. The first versions failed to comply with the EU directive 95/46/EC and Germany was forced to adapt it. There have been several changes to the privacy-related legislation in the history of the BDSG since 1990. An illustrative but impacting change of this law took place in 2001. In addition to ordinary private data, the 2001 version of the BDSG introduces *special categories of personal data* including data about racial or ethnic origin, political opinions, religious or philosophical convictions, union membership, health and sex life [BfD14]. The access to this kind of

data needs to be more restrictive as enforced in section 13 par. 2 BDSG (translated):

> "The collection of special types of personal data (Section 3 (9)) is permissible only in so far as [...] 7. Such collection is necessary for the purposes of preventive medicine, medical diagnosis, health care or the administration of health services and the processing of these data is carried out by medical personnel or other persons who are subject to an obligation to maintain secrecy [...]."

Another example of far reaching changes to privacy laws came into effect on 25th of May, 2018. The EU directive 95/46/EC was replaced by a new, EU-wide privacy law, the General Data Protection Regulation (GDPR) [EU 16]. This law furnishes users with a number of additional rights. For example, the user has the *right to be forgotten*, requesting from a company to delete all personal data. If deletion is not possible, the company then has to make sure that the data is not processed anymore and may only be stored for legal reasons, for example. This is mainly given by Art. 18 of the GDPR:

> "The data subject shall have the right to obtain from the controller restriction of processing where one of the following applies:
> 1. [...] the processing is unlawful and the data subject opposes the erasure of the personal data and requests the restriction of their use instead [.]
> 2. Where processing has been restricted under paragraph 1, such personal data shall, with the exception of storage, only be processed with the data subject's consent or for the establishment, exercise or defence of legal claims or for the protection of the rights of another natural or legal person or for reasons of important public interest of the Union or of a Member State. [...]"

These two law changes are examples of regulatory context evolutions, having the potential to require substantial changes in legacy systems.

Encryption Algorithm Exploit

Regretfully, during the last couple of years, a number of algorithms / variants/ implementations for data encryption have been discovered to be insecure. We refer to one popular example. The cipher suite RC4 has been popular over a long period of time and has been used in Transport Layer Security (TLS) to provide security for HTTP sessions. A number of attacks to break the encryption have been discovered.

After the publication of an attack that could be carried out in merely 75 hours [VP15], the use of RC4 has been prohibited in a Request for Comments (RFC) by the Internet Engineering Task Force [Pop15]. At that time, the estimation of TLS traffic relying on RC4 was approximately 30%. HTTP is, then and now, also used to implement APIs for the interoperability of distributed systems. Thus, systems using RC4 were vulnerable and needed to be evolved.

The deprecation of RC4 was critical because, for instance, many web servers relied on it. As is also today, Hypertext Transfer Protocol (HTTP) or HTTPS was not only used for web sites but also for REST-APIs and application servers, numerous business applications are affected by this vulnerability.

Trust in external Libraries

Information systems in general and application servers in particular are prone to exploits originating from data that is processed but checked insufficiently. The more an application depends on external libraries, the more likely it is that a vulnerable library is used. For example, the iTrust project has Maven dependencies on over 90 external libraries. Especially regarding object deserialization, Java programs are prone to remote code execution attacks. Such vulnerabilities also already appeared in the wild, resulting in a remote code execution exploit. For example, the JSON library Jackson suffered a vulnerability like this, documented in the CVE-2017-7525 [MIT17a]. This means that every product relying on this particular library is still vulnerable to remote code execution attacks.

As the JavaEE application servers themselves are Java programs, this risk also applies to them and in turn to every hosted application. This in fact happened: For instance, the application server JBoss was vulnerable to deserialization attacks because it in incorporated a vulnerable version of the Apache Commons collections class. A proof-of-concept exploit was presented at the AppSecCali conference organized by OWASP and became known to the public as *ysoserial* [LF15].

10.3 Classification and Relevance of Vulnerabilities

To classify the vulnerabilities in this case study and thus assess their relevance, we will relate them to the Common Weakness Enumeration (CWE) [MIT17b] database, a set of common software security weaknesses. The knowledge represented in CWE is provided by about 50 different organizations and companies of the software engineering community.

We will discuss five cases of vulnerabilities that can be classified into six CWE entries, three of them (marked with an asterisk) being part of the 2011 CWE/SANS Top 25 most dangerous software errors[1].

These CWE entries are representative examples of weaknesses that can be addressed at the system design level. We recall the types of security-related context evolutions and relate the CWEs to them.
Regarding *changes to privacy laws*, we will cover

- CWE-284: Improper Access Control,
- CWE-311*: Missing Encryption of Sensitive Data, and
- CWE-732*: Incorrect Permission Assignment for Critical Resource.

With regard to *encryption algorithm exploit*, we will focus on

- CWE-327*: Use of a Broken or Risky Cryptographic Algorithm.

Finally, *trust in external libraries* is covered by

- CWE-20: Improper Input Validation and
- CWE-502: Deserialization of Untrusted Data.

We will cover one example related to one CWE entry per section, while the fifth example (see Section 10.9) is related to both CWE-20 and CWE-502.

[1] `http://cwe.mitre.org/top25/` (accessed Apr 9th, 2019)

10.4 S^2EC^2O Application Examples

Among others, the law changes discussed above can have a significant impact to iTrust, since it processes different sort of privacy-relevant data. In what follows in this chapter, we show application examples of S^2EC^2O to the iTrust system. Each of the following sections is built around the same pattern. The goal of the rest of this chapter is to provide application scenarios and show how S^2EC^2O can be applied to a realistic information system that features typical properties of a long-living software system. The sections are accompanied by artifacts coming from the actual source code (or model respectively).

Introduction

Every of the application examples begins with an introduction. A Security Context Catalog entry is shown and the relevant security context is given.

Initial Compliance

In this part, it is shown how a system, in this case iTrust, can be made initially secure using S^2EC^2O. This basically correlates to the S^2EC^2O initialization process introduced in Section 2.2.3 (on page 15). It incorporates both system modeling and necessary information to be put into the SCK. Assumptions the ESR makes to the system model and the SCK to work properly (in particular regarding the security property checks, SMR and run-time monitors), are part of the `description` property of the respective ESR.

Context Evolution and Vulnerability

In this part, a context evolution affecting the Security Context Catalog entry is introduced. The context evolution is verified by security breaches or changes that actually have taken place. We elaborate, why the introduced evolution now leads to a potential vulnerability and relate it to CWE entries. Where applicable, we discuss how the context evolution of the SCK can be detected and using which technique.

Security Maintenance and Co-Evolution

In this part, the pseudo code of the SMR(s) connected to the Security Context Catalog entry as introduced in the introductory section is shown. Code for the ESR method `checkConditions` is shown as well as code for the method to be called when a SMR is to be applied (`apply`). Techniques and relevant artifacts for the co-evolution are shown as well as co-evolved artifacts of iTrust.

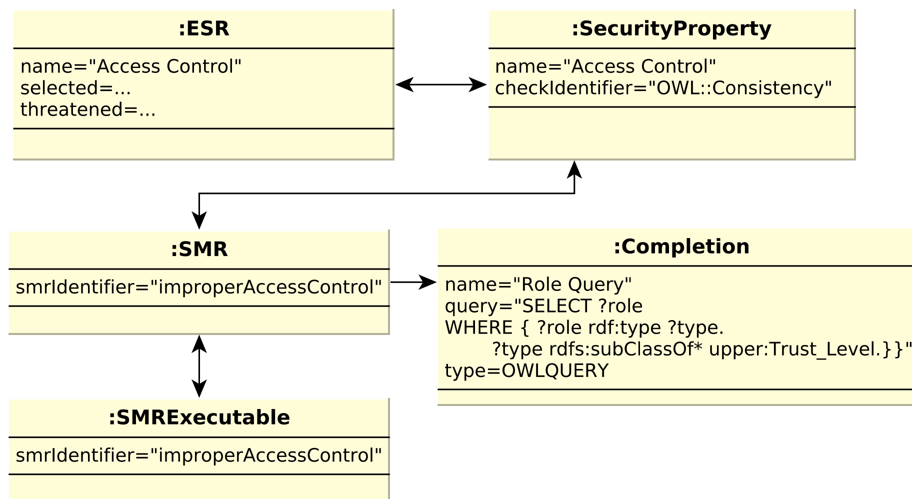


FIGURE 10.2: Security Context Catalog entry for Access Control

10.5 Example 1: Access Control

The Security Context Catalog entry we cover in this section is used to support monitoring compliance to access control restrictions. Figure 10.2 shows the model part of the Security Context Catalog entry. We want to observe access restrictions of roles regarding data that is processed by the system. It is based on modeling roles and access restrictions as part of the SCK. The UML model however contains information about which roles access what data. Both sources are brought together in the SCK and the restrictions are checked as part of the ontology. In this case, as indicated by `OWL::Consistency` in the security property is checked by monitoring the consistency of the SCK, i.e. the ontology. The assumption is that both access restrictions and roles are modeled as part of the SCK. As soon as the ontology is considered to be inconsistent, this may be an indication that these do not match anymore.

10.5.1 Initial Compliance

The initial compliance of the system with this Security Context Catalog entry is established by first modeling an appropriate base for the access control as part of the SCK.

Figure 10.3 shows an excerpt of the entities to be modeled. The role hierarchy needs to be modeled as subclasses below the `Trust_Level` class. In this example, we assume that access to data is realized via getter/setter methods of the classes holding the respective data. The `Trust_Level` class, as well as `Access_Point` for example and the used object property assertions (`accessibleBy`, `provideAccessTo`) come from the security upper ontology as introduced in Section 4.2.1 (on page 30).

Assets to be protected need also be modeled as part of the SCK. For example, we work with a hierarchy for data assets (`Data`, `Personal_Data`, `Health-related_Data`) as introduced by the German privacy law BDSG [Bun05]. In this example, we focus on `Religion` as a valuable asset. Access to this asset is possible using for example a getter method of a respective data class. Thus, the information about religion should only be accessible to trustworthy employees, in this case all kinds of health care personnel (thus the roles DLHCP, LHCP, and UAP). This means in turn that the

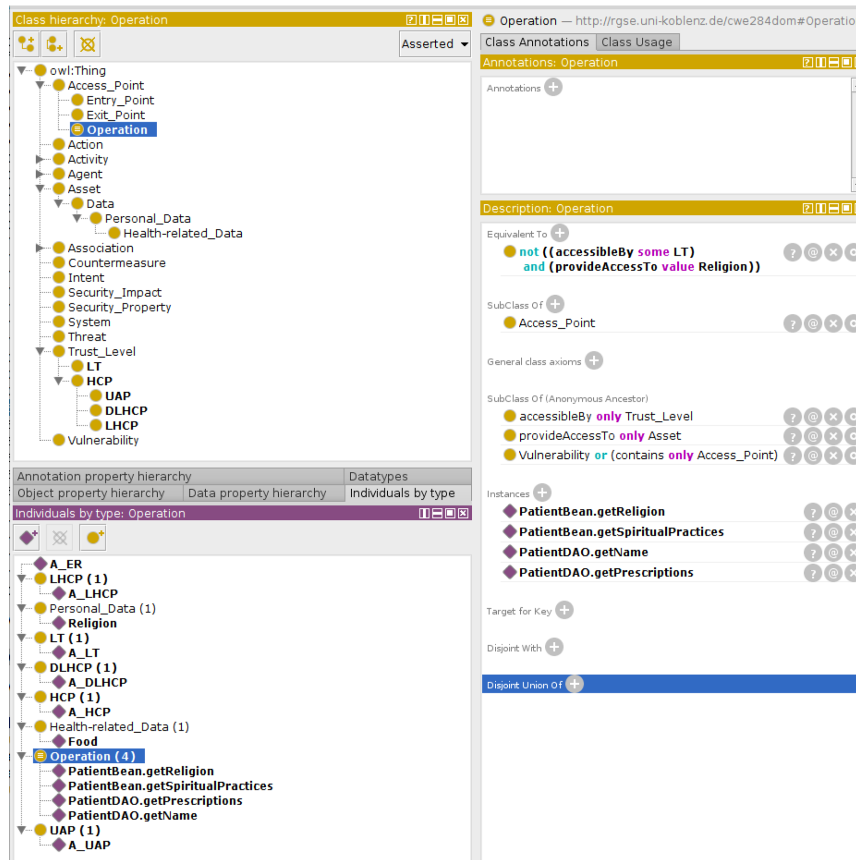


FIGURE 10.3: SCK excerpt to model access control

data must not be accessed by lab technicians (LT) for instance. This is restriction is expressed using an equivalence expression for `Operation`:

```
not ((accessibleBy some LT) and (provideAccessTo value
Religion))
```

To allow consistency checking to happen inside of the ontology representing the SCK, it is necessary for the operations (of the UML system model) to be present in it. We realized this as part of the prototype. Section 4.3.6 introduces a «SCK» stereotype that can be used to annotate elements of the UML model. Information can be provided regarding what are the ontology classes (types) and also relations to other individuals (as far as they exist) which the annotated elements should be associated with. This way, a light-weight extension to system model is at hand, providing an easy-to-use method that can help to benefit from external knowledge management tools like ontology reasoning.

In this case, the transformation provides the instances of the SCK's `Operation` class (see Figure 10.3) as well as their relations to other elements of the SCK.

The information which method provides access to what data can for instance be provided by audits, heuristics against method names and data names, or by re-using information from an access control model, for instance the UMLsec security annotation «rbac». In this case, we assume the information to come from an audit. In what follows, we relate the SCK as shown in Figure 10.3 to the UML model of iTrust.

We show excerpts of two central classes of iTrust regarding patient personal data, namely `PatientDAO` in Figure 10.4 and `PatientBean` in Figure 10.5. For the sake of clarity, we display all properties but only the operations relevant for this case study.

```
PatientDAO
- patientLoader: PatientLoader [1]
- procedureLoader: ProcedureBeanLoader [1]
- diagnosisLoader: DiagnosisBeanLoader [1]
- prescriptionLoader: PrescriptionBeanLoader [1]
- factory: DAOFactory [1]
- personnelLoader: PersonnelLoader [1]

+ deleteDesignatedNutritionist(  in patientMID: long): int
+ fuzzySearchForPatientsWithMID(  in MID: long): java.util.List<edu.ncsu.csc.itrust.beans.PatientBean>
+ addEmptyPatient(): long
+ hasHistory(  in pid: long): boolean
+ getExpiredPrescriptions(  in patientID: long): java.util.List<edu.ncsu.csc.itrust.beans.PrescriptionBean>
+ getDiagnoses(  in pid: long): java.util.List<edu.ncsu.csc.itrust.beans.DiagnosisBean>
+ getName(  in mid: long): String
+ removeAllRepresentee(  in representeeMID: long): voic
+ searchForPatientsWithName(  in last: String,   in first: String): java.util.List<edu.ncsu.csc.itrust.beans.PatientBean>
+ updateDesignatedNutritionist(  in HCPID: long,   in patientMID: long): int
+ editPatient(  in p: PatientBean,   in hcpid: long): void
+ undeclareHCP(  in hcpID: long,   in pid: long): boolean
+ addRepresentative(  in representee: long,   in representer: long): boolean
+ PatientDAO(  in factory: DAOFactory)
+ removeAllRepresented(  in representerMID: long): void
+ represents(  in representer: long,   in representee: long): boolean
+ checkIfRepresenteeIsActive(  in representee: long): boolean
+ checkPatientExists(  in pid: long): boolean
+ addDesignatedNutritionist(  in patientMID: long,   in HCPID: long): int
+ getDesignatedNutritionist(  in patientMID: long): long
+ getAllPatients(): java.util.List<edu.ncsu.csc.itrust.beans.PatientBean>
+ getPatient(  in mid: long): PatientBean
+ checkDeclaredHCP(  in hcpid: long,   in pid: long): boolean
+ getImmunizationProcedures(  in pid: long): java.util.List<edu.ncsu.csc.itrust.beans.ProcedureBean>
+ getDeclaredHCPs(  in pid: long): java.util.List<edu.ncsu.csc.itrust.beans.PersonnelBean>
+ getRepresenting(  in pid: long): java.util.List<edu.ncsu.csc.itrust.beans.PatientBean>
+ getPrescriptions(  in patientID: long): java.util.List<edu.ncsu.csc.itrust.beans.PrescriptionBean>
+ addHistory(  in hcpid: long,   in pid: long): void
+ declareHCP(  in hcpID: long,   in pid: long): boolean
+ getRepresented(  in pid: long): java.util.List<edu.ncsu.csc.itrust.beans.PatientBean>
+ getProcedures(  in pid: long): java.util.List<edu.ncsu.csc.itrust.beans.ProcedureBean>
+ getCurrentPrescriptions(  in patientID: long): java.util.List<edu.ncsu.csc.itrust.beans.PrescriptionBean>
+ checkIfPatientIsActive(  in pid: long): boolean
+ getRole(  in mid: long,   in role: String): String
+ getPatientHistory(  in mid: long): java.util.List<edu.ncsu.csc.itrust.beans.PatientHistoryBean>
+ fuzzySearchForPatientsWithName(  in last: String,   in first: String): java.util.List<edu.ncsu.csc.itrust.beans.PatientBean>
+ getRenewalNeedsPatients(  in hcpMID: long): java.util.List<edu.ncsu.csc.itrust.beans.PatientBean>
+ getDependents(  in pid: long): java.util.List<edu.ncsu.csc.itrust.beans.PatientBean>
+ removeRepresentative(  in representee: long,   in representer: long): boolean
```

FIGURE 10.4: Excerpt of iTrust class `PatientDAO`

```
PatientBean
- icPhone: String [1]
- icState: String [1]
- emergencyPhone: String [1]
- dateOfDeactivationStr: String [1]
- causeOfDeath: String [1]
- alternateName: String [1]
- spiritualPractices: String [1]
- securityQuestion: String [1]
- topicalNotes: String [1]
- icZip: String [1]
- bloodType: BloodType [1
- motherMID: String [1]
- lastName: String [1]
- confirmPassword: String [1]
- icAddress2: String [1]
- securityAnswer: String [1]
- icName: String [1]
- gender: Gender [1]
- city: String [1]
- creditCardType: String [1]
- emergencyName: String [1]
- creditCardNumber: String [1]
- icID: String [1]
- serialVersionUID: String [1]
- language: String [1]
- directionsToHome: String [1]
- firstName: String [1]
- streetAddress1: String [1]
- password: String [1]
- MID: long [1]
- ethnicity: Ethnicity [1]
- dateOfBirthStr: String [1]
- fatherMID: String [1]
- streetAddress2: String [1]
- religion: String [1]
- dateOfDeathStr: String [1]
- zip: String [1]
- phone: String [1]
- icCity: String [1]
- state: String [1]
- icAddress1: String [1]
- email: String [1]

+ getLastName(): String
+ getFullName(): String
+ getDateOfBirth(): Date
+ getAge(): int
«SCK» {types=[Operation] , relations=[provideAccessTo Religion, accessibleBy A_LHCP, accessibleBy A_DLHCP, accessibleBy A_UAP] }  + getReligion(): Strin
«SCK» {types=[Operation] , relations=[provideAccessTo Religion, accessibleBy A_LHCP, accessibleBy A_DLHCP, accessibleBy A_UAP] }  + getSpiritualPractices(): Strin
```

FIGURE 10.5: Excerpt of iTrust class `PatientBean`

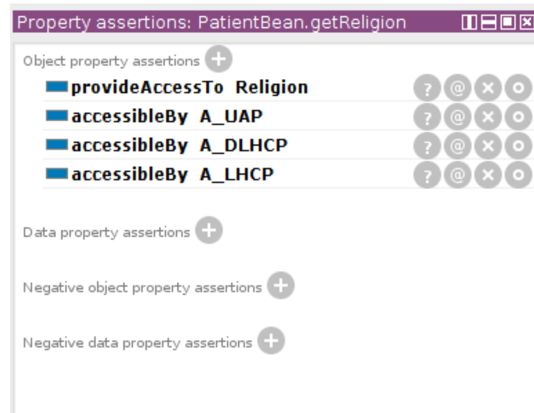


FIGURE 10.6: SCK excerpt: relations of `getReligion` to other SCK elements

The PatientDAO is heavily connected to the rest of the system. For example, just the method `getPatient` is called by over 40 other methods. It connects to the database and handles `PatientBean` objects to exchange information.

Figure 10.5 shows `PatientBean`, note the «SCK» annotations at the two methods providing access to religion-oriented information about the patient.

Figure 10.6 shows the relations that `getReligion` has to other elements in the ontology. It represents which data is exposed by the method (`provideAccessTo`) and which roles have access to this method (`accessibleBy`). As we discussed above, the ontology elements as shown in Figure 10.6 are generated by the transformation from the model as shown in Figure 10.5 (note the «SCK» stereotypes).

An advantage of this separation (role model as part of the SCK, access modeling and coupling to the methods in the system model) is that this way a separation of concerns is achieved: the system model is only used to annotate the access possibilities as is, while the restriction policies can easily be managed within the SCK.

Note the following: object property assertions can only be targeted against individuals, not classes. But regarding roles for access control, a hierarchy is more easily modeled using equivalence expressions. However, one might also want to model concrete users of the system which would take place using individuals also. To solve this issue, as is also shown by Figure 10.3, representative individuals for the roles with the prefix `A_` are used.

10.5.2 Context Evolution and Vulnerability

In this example, we consider a change that has to be made to the access control model triggered by a change in legislation that has actually taken place. As we described in Section 10.2, the German privacy law changed in 2001 and extended the notion of personal data by introducing *special categories of personal data*, including data about racial or ethnic origin. In this example, we treat religion and spiritual practices as that kind of data.

The assumption is that Unlicensed Authorized Personnel (UAP) is not allowed to access this information anymore. To adapt the access control to the changed law context, the security expert alters the access restriction as follows:

```
not (((accessibleBy some LT) or
(accessibleBy some UAP)) and (provideAccessTo value
Religion))
```

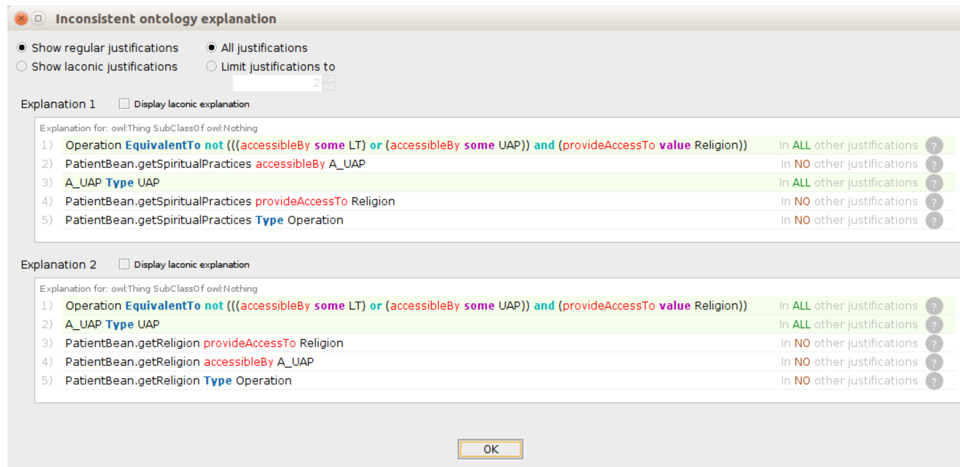


FIGURE 10.7: Reasoner explanation for SCK inconsistency

After this step, the context evolution is integrated into the SCK. The question that follows is, whether the system actually is affected by this change and a co-evolution needs to take place.

In this case, this can be answered using the reasoner (see `SecurityProperty` element in Figure 10.2). The change made leads the SCK ontology to being inconsistent. A reasoner also can provide explanations for the inconsistency. Figure 10.7 shows the explanations for the inconsistency, critical parts in red.

The reasoner is able to derive that the operations `getReligion` and `getSpiritualPractices` of the class `PatientBean` provide access to data to a role where this is not allowed anymore.

Thus, the system is now violating the respective ESR. This can be seen as an occurrence of CWE-284: Improper Access Control[2]. The short description of this CWE reads: *The software does not restrict or incorrectly restricts access to a resource from an unauthorized actor.*

10.5.3 Security Maintenance and Co-Evolution

In this section we consider how the system is co-evolved to remove the violation.

```
1  public void checkConditions(DeltaList deltaList) {
2   for (OntologyInconsistencyDelta d : deltaList.getDeltas()) {
3    criticalOperations=d.getExplanations()
4    .collect("Type Operation");
5    criticalRoles=d.getExplanations()
6    .collect("accessibleBy");
7    queryAlternativeRoles();
8   for (operation : criticalOperations) {
9    Proposal.addAlternative(operation,REMOVE_CRITICAL_ROLE);
10   Proposal.addAllAlternatives(operation,queryAlternativeRoles());
11 }
12
13 public void apply(Proposal p) {
14   alterModel(p.getChoice());
15  }
16 }
```

LISTING 10.1: Pseudo code for SMR co-evolution regarding CWE 284

[2] https://cwe.mitre.org/data/definitions/284.html (accessed Apr 9th, 2019)

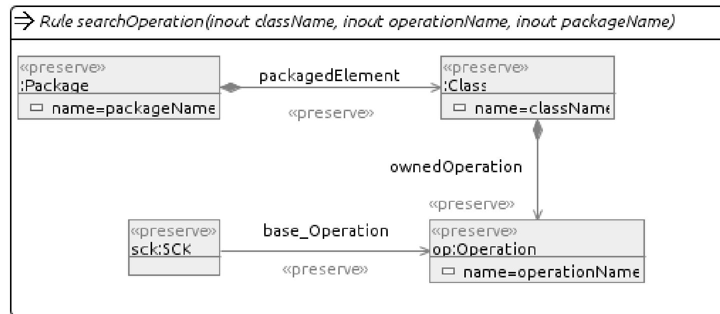


FIGURE 10.8: Henshin rule to search ⟪SCK⟫ annotated operations

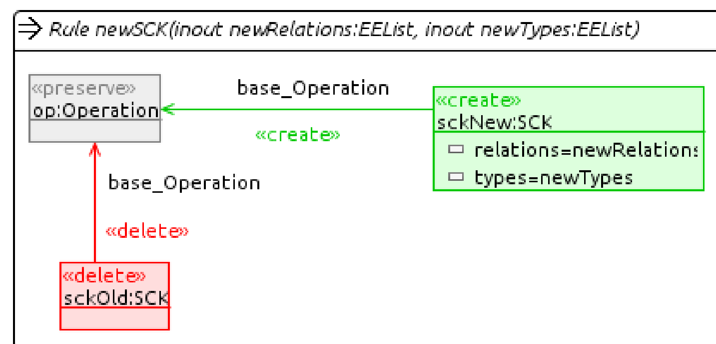


FIGURE 10.9: Henshin rule to alter ⟪SCK⟫ annotations

Listing 10.1 shows the pseudo code of the SMR coming into action. In the method `checkConditions`, the model and delta information is processed and alternatives to co-evolve the system are determined and proposed. The SMR shown here is regarding only delta information relating to inconsistent ontologies.

This is done by processing the explanation information provided by the reasoner.

Thus, in lines 3-6, the operations and roles in question are determined by processing the explanation data. Thus, all explanation data as provided in the delta information is analyzed for data that is relevant. The SMR analyzes the relations of operations and access information as provided by the reasoner. It determines alternative roles. Basically, this is realized by issuing a SPARQL query towards the SCK. The query is defined as completion element and it reads as follows:

```
SELECT ?role
WHERE { ?role rdf:type ?type.
?type rdfs:subClassOf* upper:Trust_Level. }
```

It queries the knowledge for roles that, in this example, are assumed to be modeled as subclasses of `Trust_Level`.

After that, in lines 8-11 of Listing 10.1, proposals, i.e. co-evolution alternatives are composed. The first possibility is just to remove the accessing role in question. Furthermore, all roles modeled in the SCK are gathered and presented to the security expert as alternatives to choose from, tied to the respective critical operation.

After the security expert has made the choices, the chosen alternatives are applied (see lines 13-16). The method `alterModel()` co-evolves the UML system model by searching the operation in question and altering the ⟪SCK⟫ annotation. This is realized using Henshin rules. Figure 10.8 shows the rule used to search a operation in the model identified by its name, the owning class and its package.

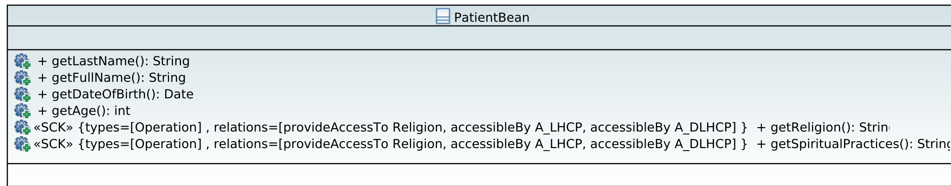


FIGURE 10.10: `The class PatientBean` in its evolved state

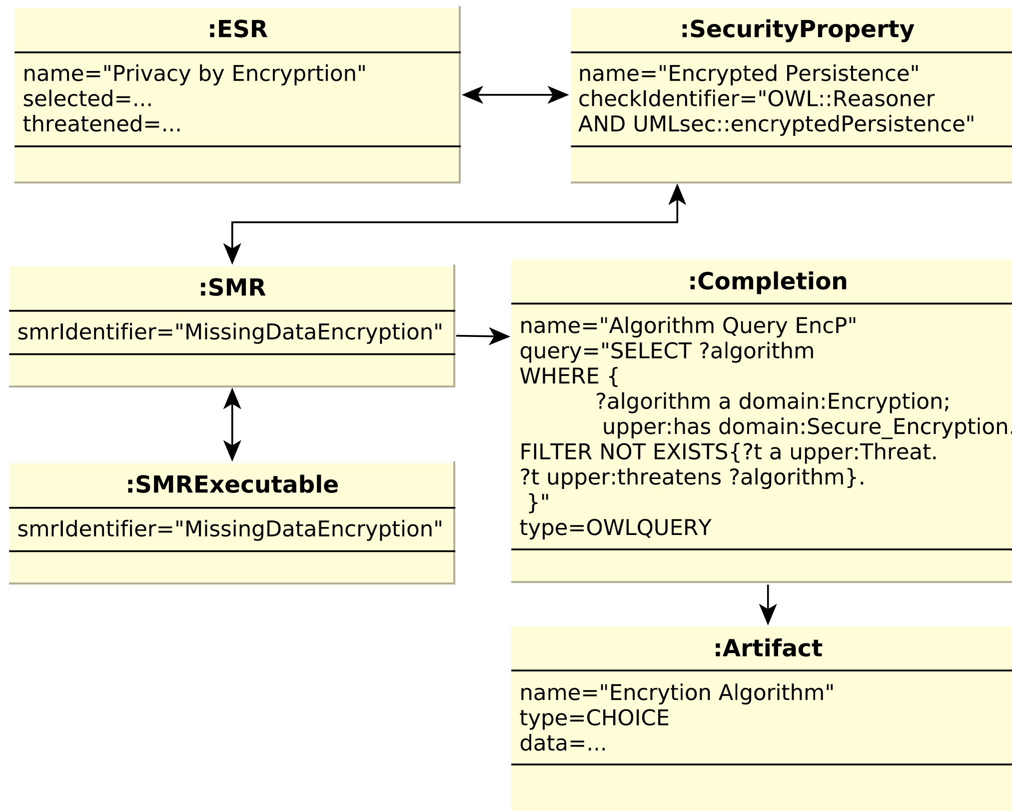


FIGURE 10.11: Security Context Catalog entry for Privacy by Encryption

Figure 10.9 shows the rule to alter a ⟪SCK⟫ annotation. Using Henshin's partial match feature as introduced in Section 3.6, it is sufficient to just refer to the `Operation` node, as the specific target EObject has been determined using the `searchOperation` rule. Using partial match and the reference to the operation, the match of this rule can be predetermined.

Figure 10.10 shows the evolved `PatientBean` class. In this case, the user selected to only remove the critical role. The access right of the UAP role is now removed. After a re-run of the transformation from the UML model to the SCK, the SCK is consistent again. The altered ⟪SCK⟫ annotations can for example be used for further analyses or a fine-grained audit to ensure the altered behavior is satisfied by the code.

10.6 Example 2: Privacy by Encryption

The goal of this Security Context Catalog entry is to ensure encryption of sensitive data. The respective ESR monitors the compliance. It makes use of both the SCK and respective modeling of the system model. The UMLsec stereotype «encryptedPersistence» is used to tag classes of the model that represent data entities and need to be persisted for example in an encrypted manner [BSG+18]. The stereotype features the tagged values `algorithm` and `keylength` to also provide the ability to specify an encryption algorithm and its key length to be used for encryption. Figure 10.11 shows the model of the Security Context Catalog entry. The combination of UMLsec modeling and leveraging the SCK can be seen by the combination of both check identifiers in the connected `SecurityProperty`. The basic idea of this ESR is that classes providing access to data assets are represented in the ontology. Furthermore, in the ontology, data assets are classified according to categories defined by privacy laws, such as we introduced in Section 10.2 (on page 132).

In this example, the OWL reasoner is used to infer which classes of the system model need «encryptedPersistence». It can then be checked by the SMR whether the classes annotated with «encryptedPersistence» match the requirements as inferred by the reasoner from the SCK.

The application relevance is as follows: A typical partitioning of information systems is to have a presentation component, a business logic component and a database as storage back-end iTrust, as introduced in Section 10.1, can also be described that way. iTrust typically uses a MySQL server as database. Especially regarding medical information systems, there are regulations to have backups available. We assume the following trigger for a context evolution: backups of databases can for example be persisted as a big set of Structured Query Language (SQL) statements (*SQL dump*). While the connection to a SQL server can be encrypted, the data itself can be stored in an unencrypted manner. For example, this can happen when database dumps/backups are stored on a server, for example using automatisms like the popular AutoMySQLBackup scripts[3]. Another possibility is, that, for example using Debian Linux, the default configuration gives administrators of the Linux system root privileges in MySQL:

> "At least since Debian 9 "stretch," operating system credentials are used by MySQL Server to authenticate users. That is, after installing mysql-server and mysql-client you can access the server with root privileges [...]."[4]

This issue can be solved by encrypting data before inserting it into the database. «encryptedPersistence» supports to annotate respective data assets in the system model.

10.6.1 Initial Compliance

The initial compliance is established by first modeling an appropriate base for the privacy as part of the SCK.

Figure 10.12 shows an excerpt of the entities to be modeled. In this example, various categories of personal/private data are modeled, namely personal data, health-related data, special personal data and data itself. These categories can be for example inferred from the German privacy law BDSG [Bun05]. As individuals, various

[3] https://sourceforge.net/projects/automysqlbackup/ (accessed Apr 10th, 2019)
[4] from https://wiki.debian.org/MySql (accessed Apr 10th, 2019)

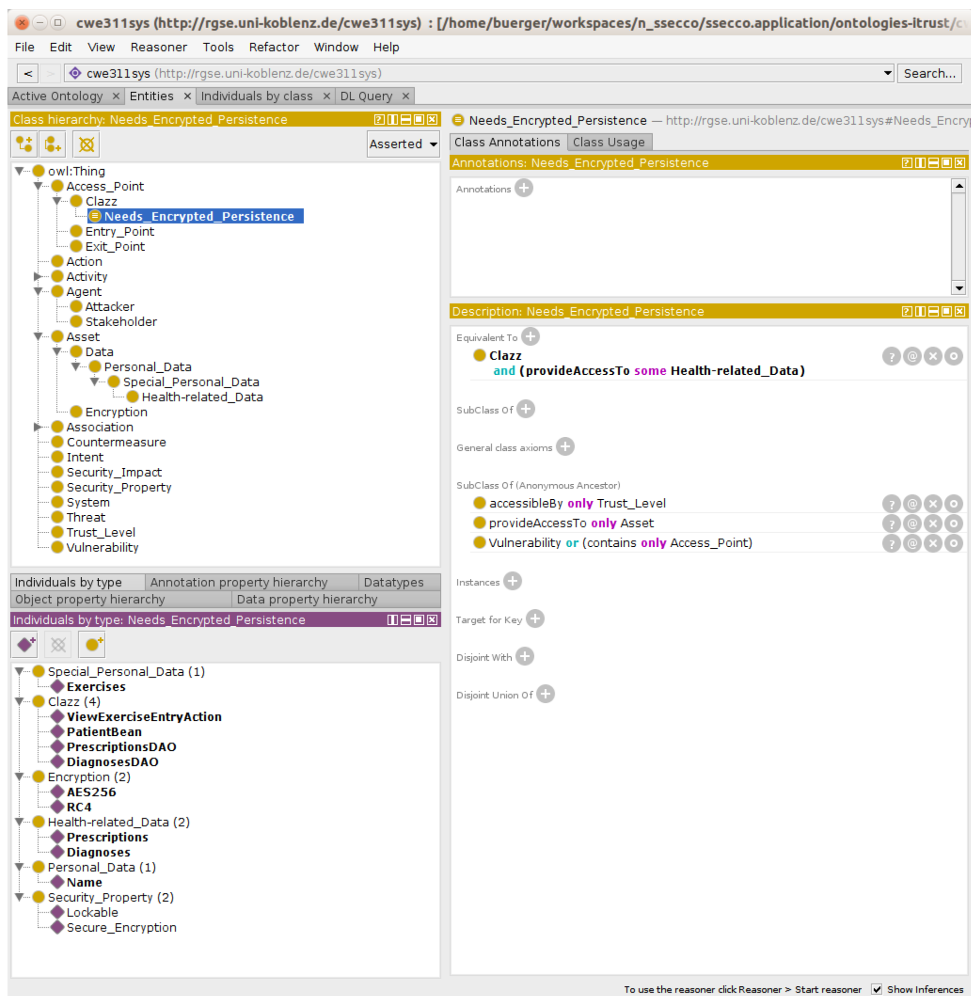


FIGURE 10.12: SCK excerpt to model privacy requirements

```
«SCK»
ViewExerciseEntryAction
«SCK»
types=[Clazz]
relations=[provideAccessTo Exercises]

- loggedInMID: long [1]
- personnelDAO: PersonnelDAO [1]
- patientDAO: PatientDAO [1]
- exerciseEntryDAO: ExerciseEntryDAO [1]

+ ViewExerciseEntryAction( in loggedInMID: long, in factory: DAOFactory)
+ getDiaryTotals( in patientMID: long): java.util.List<edu.ncsu.csc.itrust.beans.ExerciseEntryBean>
+ getBoundedDiaryTotals( in lowerDate: String, in upperDate: String, in patientMID: long): java.util.List<edu.ncsu.csc.itrust.beans.ExerciseEntryBean>
+ getDiary( in patientMID: long): java.util.List<edu.ncsu.csc.itrust.beans.ExerciseEntryBean>
+ getBoundedDiary( in patientMID: long, in upperDate: String, in lowerDate: String): java.util.List<edu.ncsu.csc.itrust.beans.ExerciseEntryBean>
```

FIGURE 10.13: iTrust class `ViewExerciseEntryAction` to provide patient exercise data

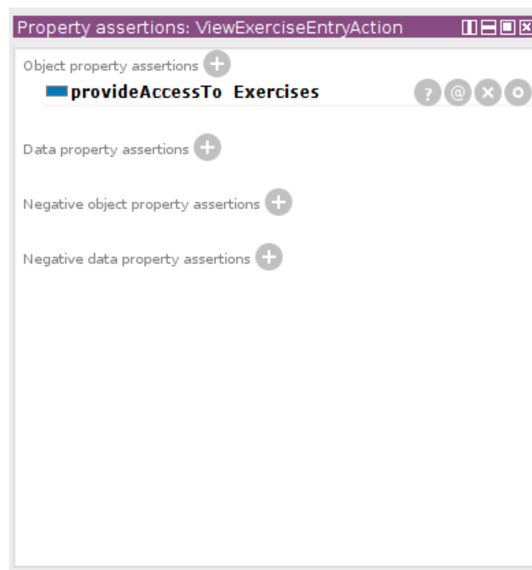


FIGURE 10.14: SCK excerpt: result of transformation from the system model to the SCK

concrete types of data assets are shown, for example `Diagnoses`, `Name`, and `Religion`. These assets need to be specified by the security expert, for example by extracting them from the system specification.

Regarding connection to the system model, as in the preceding section, the ⟪SCK⟫ annotation is used to tag classes with their ontology type (`Clazz` in this case to avoid naming confusion with classes as members of the ontology an Java classes, represented in the ontology) and also data assets they provide access to.

The SCK is populated with encryption algorithms. We discussed in Section 4.3 (on page 35), how this knowledge can be acquired and incorporated into the knowledge base.

Using the transformation we also used for the example in Section 10.5, the classes from the model as well as their associations are put into the SCK.

Figure 10.13 shows the `ViewExerciseEntryAction` class. This class is used in iTrust and directly called by a respective web page to display information about a patient's (fitness) exercises. Using a ⟪SCK⟫ annotation as shown in the figure, the information that this is a class to be represented in the SCK and the data assets it provides access to, can be transformed into the SCK and used for analyses. Figure 10.14 shows this in detail. The class `ViewExerciseEntryAction` is an individual

in the ontology and the relation `provideAccessTo Exercises` has also been applied. We can tell from the SCK that the class `ViewExerciseEntryAction` provides access to the data asset `Exercises`. While data of a class is generally provided by public getter- and setter-methods, the class itself has access to all of its methods and fields.

Figure 10.12 also shows the requirement for encrypted persistence, modeled as equivalence expression:

```
Clazz and (provideAccessTo some Health-related_Data)
```

We use the approach of inferring additional knowledge by the reasoner as we introduced in Section 5.3.4 (on page 59). Thus, the class `Needs_Encrypted_Persistence` lets the reasoner infer all classes to which the equivalence expression applies. The security expert needs to model the restriction according to the, for example, company policies for asset encryption. As introduced in Section 5.3.4, a reasoner can be used to infer knowledge. In this case, the reasoner can infer which classes need to have encrypted persistence.

In addition to the preceding section, we also show a separation of concerns in this example: In the system model, only tagging of classes regarding their access to data is modeled, while the categorization as well as modeling of security restrictions is modeled in the SCK.

10.6.2 Context Evolution and Vulnerability

We assume that data breaches have taken place which leads to a change in the company's security policies. The company policy now requires not only health-related data to be encrypted but, on a more coarse-grained level, also special personal data. Thus, the equivalence expression in the SCK is changed to the following:

```
Clazz and (provideAccessTo some Special_Personal_Data)
```

Using the reasoner, now additional classes are inferred as instances of the `Needs_Encrypted_Persistence` class. The vulnerability now being present can be found in the CWE database as CWE-311: Missing Encryption of Sensitive Data[5]. The description reads: *The lack of proper data encryption passes up the guarantees of confidentiality, integrity, and accountability that properly implemented encryption conveys.*

Please note that no changes to the system model or system model-specific elements had to be realized. Only by changing the (separate) information about the categorization, the reasoner is able to infer the additional classes needing encrypted persistence. In principle, modeling with equivalence expressions can be significantly more complex, as long as they can be still (logically) inferred by a reasoner.

Figure 10.15 shows the reasoning results after altering the subclass expression for `Needs_Encrypted_Persistence`. The class `ViewExerciseEntryAction` now also needs encrypted persistence.

10.6.3 Security Maintenance and Co-Evolution

Listing 10.2 shows the pseudo code for the relevant parts of the SMR coming into effect. In the method `checkConditions`, the delta information is processed. In line 3, the classes are filtered from the inferred facts that actually are relevant for this SMR, namely all (thus inferred) instances of the `Needs_Encrypted_Persistence` class. To put this in relation to the system model, all instances of UML classes annotated with

[5] `https://cwe.mitre.org/data/definitions/311.html` (accessed Apr 9th, 2019)

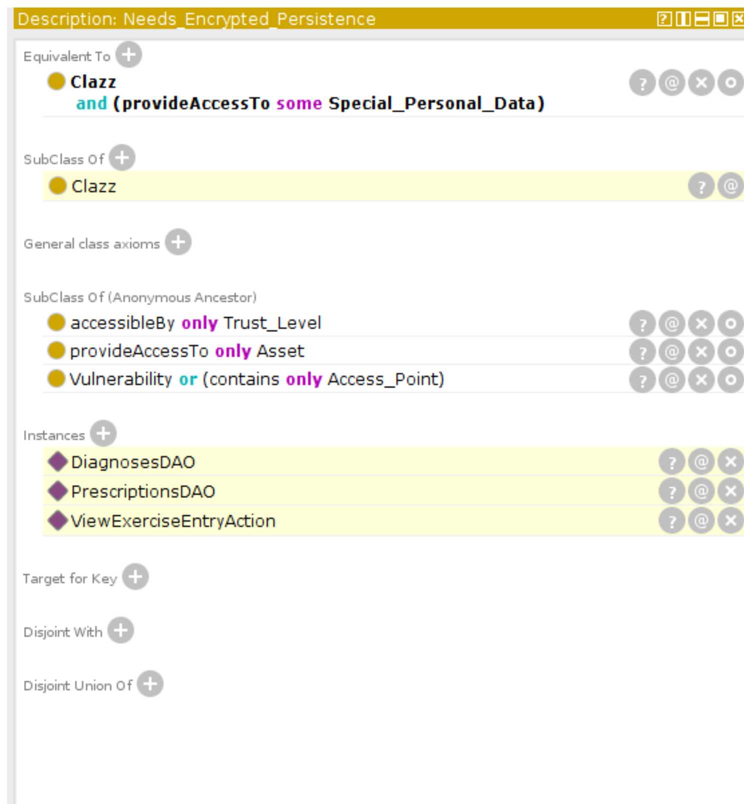


FIGURE 10.15: Inferred knowledge (yellow background) after evolution of SCK

«encryptedPersistence» are queried. This is realized using a model query as shown in Figure 10.16.

```
1  public void checkConditions(DeltaList deltaList) {
2   for (OntologyInferenceDelta d : deltaList.getDeltas()) {
3    ontologyClasses=d.getInferredFacts().filter("Type
        Needs_Encrypted_Persistence");
4    queryModelClasses();
5    if (!doAnnotationsMatch()){
6    Proposal.addAlternative("Add missing annotations");
7    }
8  }
9
10 public void apply(Proposal p) {
11   alterModel(p.getChoice());
12  }
13 }
```

LISTING 10.2: Pseudo code for SMR co-evolution regarding CWE 311

In line 5, it is checked whether for every class in the SCK a respective annotation in the system model exists. This is done by comparing both sets of classes. If the annotations do not match, an alternative to co-evolve the system is proposed where «encryptedPersistence» annotations are added to all missing classes.

Here comes the completion element of the SMR into action (see Figure 10.11 on page 142). Using a SPARQL query, the SCK is queried for applicable encryption algorithms to propose for persistence encryption.

This completion is of type `CHOICE`, so, according to the S^2EC^2O process and the associated tool support, the user is asked to make a choice when going through the delta wizard. The choice is put into the `data` attribute of the `Choice` element.

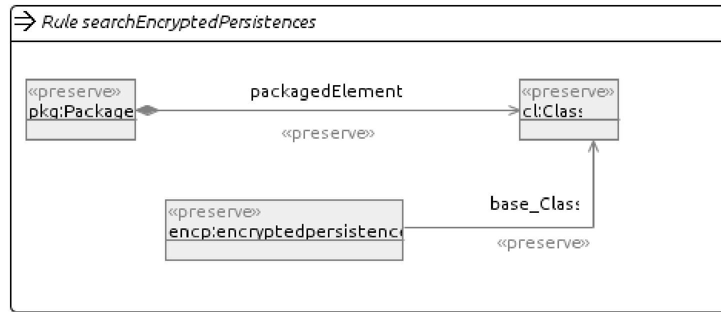


FIGURE 10.16: Henshin rule to search classes annotated with 《encryptedPersistence》

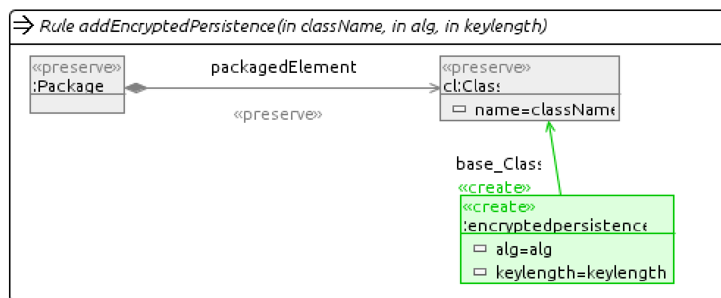


FIGURE 10.17: Henshin rule for adding 《encryptedPersistence》 to a specific class

When the SMR finally is applied (lines 10-12), the choice data is used to specify the parameters of the graph transformation rule used to carry out the model co-evolution, as shown in Figure 10.17.

Regarding the example, Figure 10.18 shows the result after the SMR has been applied: the annotation 《encryptedPersistence》 has been added.

After a re-run of the SCK transformation, the information regarding the model elements of the system model is in sync with the SCK again and fully compliant with the updated restriction in the SCK.

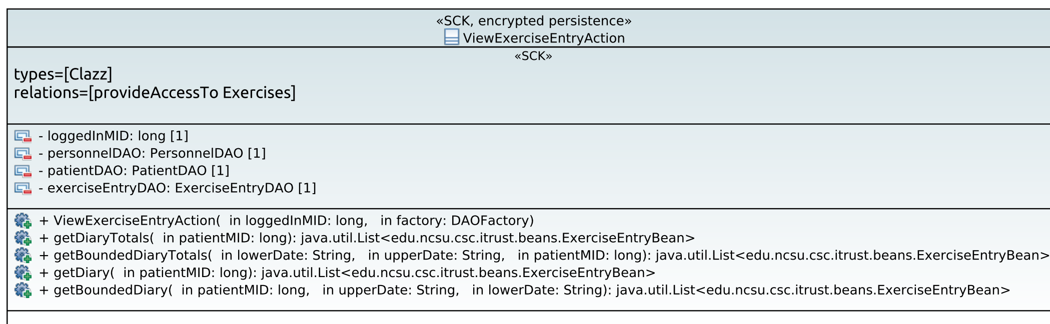


FIGURE 10.18: Co-evolved class `ViewExerciseEntryAction` with added annotation

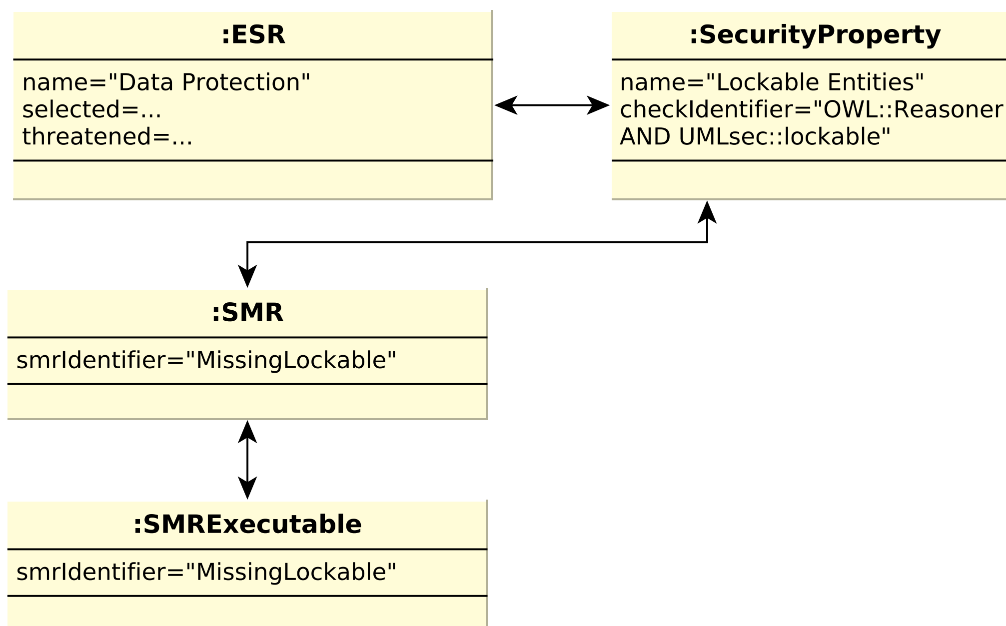


FIGURE 10.19: Security Context Catalog entry for Data Protection

10.7 Example 3: Data Protection by Locking

This Security Context Catalog entry is used to support monitoring to ensure the privacy of data. The idea behind this ESR is to support modeling of data entities which need to be lockable in an information system. For example, when the new version of the EU data protection law, GDPR, came into effect on the 25th of May, 2018, it explicitly introduced the customer's right to let a company stop processing and lock his data [EU 16].

In many cases, the data cannot be deleted by the company immediately. For reasons of liability, or required by tax laws, the customer data has to be stored, even after the customer terminated all contracts with a company. But, in turn, the company then has to make sure that the data is only available for exactly these above mentioned purposes.

The application example we discuss in this section can be seen as an extension to that in Section 10.6, as it reuses the modeling of data entities. Modeling the ability of being lockable is realized by providing a UMLsec stereotype «lockable». Using it, we can tag classes that provide access to certain data entities with the requirement of being lockable.

In this example, addition of a lockable requirement as part of the SCK is detected using a semantic difference determined by SiLift. Additionally, a reasoner is used to determine the type entailment of related ontology classes.

10.7.1 Initial Compliance

The initial compliance of iTrust with the ESR is established by first modeling which data entities need to be lockable as part of the SCK. This is realized by providing a security property `Lockable` in the SCK the ESR relates to. Furthermore, the security expert needs to associate all data entities that shall support to be locked with this security property by using the object property `has`.

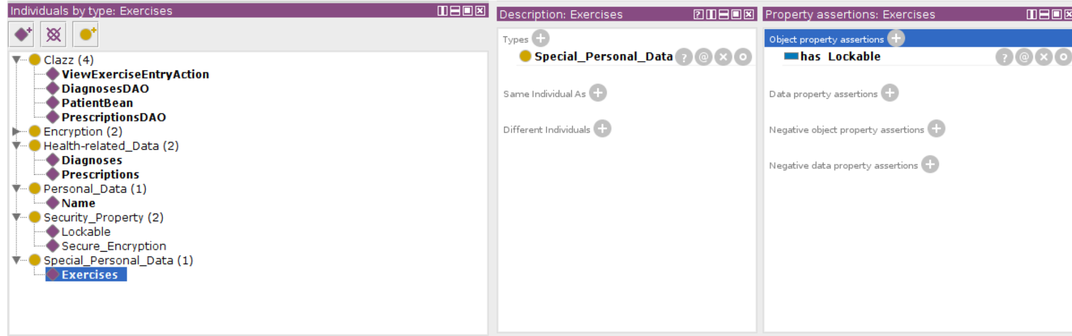


FIGURE 10.20: SCK excerpt showing a lockable data entity

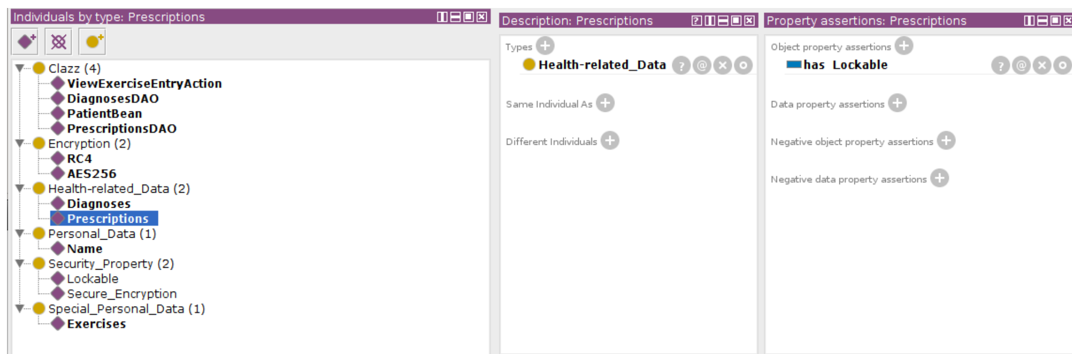


FIGURE 10.21: SCK excerpt showing a new lockable data entity according to the law change

Figure 10.20 shows an excerpt of the SCK. In this case, the data entity `Exercises` (highlighted in blue) shall be lockable. This is modeled by the object property assertion `has Lockable`.

As discussed above, the structure built in the SCK from the previous example is reused. This means, classes of the system model are tagged using ⟪SCK⟫. For all relevant classes, using the `provideAccessTo` object property, the link between classes of the system model and the SCK is established.

Using the transformation as introduced in Section 10.6, the ⟪SCK⟫ annotations can be parsed and transformed into SCK entities.

10.7.2 Context Evolution and Vulnerability

The context evolution in this example is a law change. The introduction of the new EU privacy law had far-reaching effects also on legacy systems, because customers were furnished with considerable additional rights.

For example, GDPR generally grants the person affected the right to let the data be deleted. Under given circumstances, a company cannot delete all data instantly because it is forced by laws to keep it for a certain amount of time. Nevertheless, data access needs then to be strongly restricted inside the company and deletion needs to be issued after a freeze period. In this example, we assume the law change to demand that data regarding the patient's prescriptions is affected by the legislation evolution. This means that prescription data also needs the capability to be locked. Thus, it needs to be checked whether the model still fulfills the *Lockable* requirement and eventually co-evolve it.

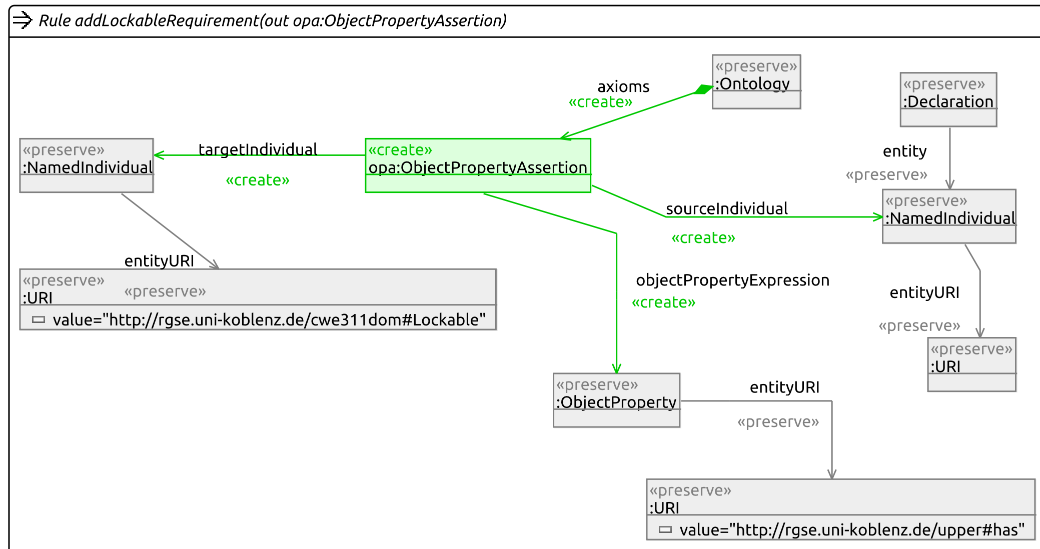


FIGURE 10.22: SiLift rule to detect addition of `Lockable` requirement in the SCK

Figure 10.21 shows the altered SCK, now with the data entity `Prescriptions` having the respective object property assertion.

10.7.3 Security Maintenance and Co-Evolution

Listing 10.3 shows the pseudo code for the relevant parts of the SMR. In the method `checkConditions`, the delta information is processed. This SMR is triggered by semantic diffs coming from analysis of different versions of the Security Context Catalog.

```
1  public void checkConditions(DeltaList deltaList) {
2   for (SemanticDiffDelta d : deltaList.getDeltas()) {
3    for (OperationInvocation opInv : d.getOperationInvocations()){
4     ObjectPropertyAssertion opa=opInv.getParameter("opa");
5     URI uri=opa.getSourceIndividual().getURI();
6     Set<String> classes=OWLReasoning.getAllClasses(uri);
7     if (classes.contains("Data")){
8      ontologyClasses=queryOntologyClasses();
9      modelClasses=queryModelClasses();
10     boolean doAnnotationsMatch=(ontologyClasses-modelClasses)==EMPTY_SET?
           true:false;
11     if (!doAnnotationsMatch){
12      Proposal.addAlternative("Add missing lockable annotations");
13     }
14    }
15   }
16  }
17 }
18
19 public void apply(Proposal p) {
20   alterModel(p.getChoice());
21  }
22 }
```

LISTING 10.3: Pseudo code for SMR co-evolution regarding CWE 732

Figure 10.22 shows the SiLift rule used for detection of the SCK's evolution. The noticeable fact is the addition of an object property assertion (shown in the upper

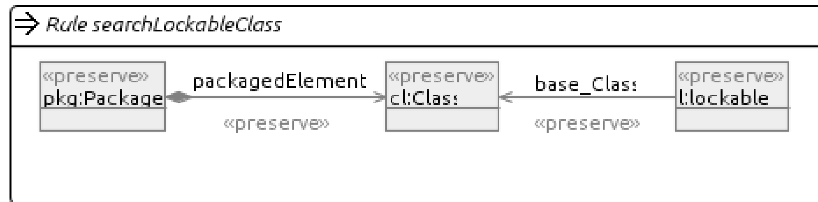


FIGURE 10.23: Henshin model query to search for «lockable» annotated classes

middle of the figure). Thus, it is specified that the object property assertion connects an individual not further specified (shown in the upper right) as source and the already existing individual `Lockable` as target. Both are connected using the object property `has`, specified in the OWL meta model and shown in the figure as `objectPropertyExpression` association.

The SMR further investigates the delta. The actual Uniform Resource Identifier (URI) is queried (lines 4-5) from the diff to determine the name of the individual for which the `Lockable` property was added.

Note the following detail: As Figure 10.22 shows, the actual type (i.e. OWL class) of the individual is not specified in the Henshin rule for SiLift. The reason is as follows: regarding this SMR, it is relevant that the individual in question is of the type *Data*. But in the SCK, the data entities are only assigned subtypes (i.e. subclasses) of this type, for example `Health-related_Data` (see also Figure 10.12 on page 144 for the hierarchy of data asset types). However, the individual in the SCK only has the specific type asserted (for example `Health-related_Data`).

From the ontology class hierarchy, one can infer that the entity also has the type `Data` because there is a path of subclass-relations to `Health-related_Data`. To get this information at run time, the SMR uses a reasoner (line 6 of Listing 10.3).

After that, similar to the SMR we introduced in Section 10.6, two sets are calculated. First, a set of classes (of the system model) needing «lockable» according to the SCK (see line 8) is calculated. To accomplish this, data entities are investigated and their associations to `clazz` individuals of the ontology is traced. In the ontology, this is realized using `provideAccessTo` object property. For every (system model) class providing access to a specific data entity which in turn is associated to the `Lockable` security property, «lockable» is required in the system model. To establish the link from system model classes represented in the SCK and the data entities, a SPARQL query as follows is used:

```
SELECT ?clz
WHERE { ?clz a domain:Clazz.
?clz upper:provideAccessTo sys:Religion.}
```

After that, the SMR has a set of all classes in need of «lockable» according to the SCK. After that, a set of classes of the system model currently annotated with «lockable» is gathered (line 9). This is realized using a model query. Figure 10.23 shows the Henshin rule used for that. It simply matches against all classes having «lockable».

In line 10 of Listing 10.3 (on page 151) it is checked whether all obligations regarding «lockable» as required by SCK are currently fulfilled by the model. This is accomplished by subtracting all annotated classes from the UML model from the set of classes as determined by the SCK. If there is no match, i.e. there are classes

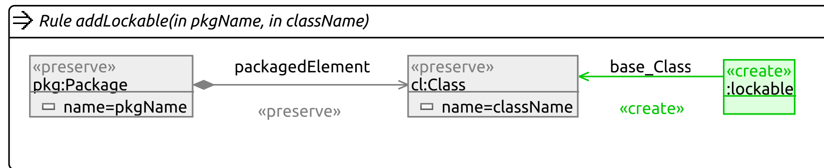


FIGURE 10.24: Henshin rule to add ⟪lockable⟫ to a given class

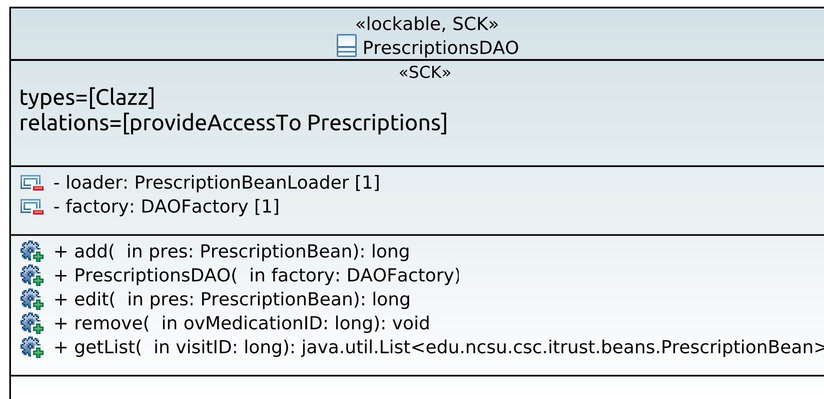


FIGURE 10.25: Co-evolved class `PrescriptionsDAO` with added ⟪lockable⟫

of the system model left missing annotations, a proposal is created (line 12) to add ⟪lockable⟫ to the remaining classes.

When the SMR finally is to be applied (lines 19-21), the set of classes to be annotated with ⟪lockable⟫ is iterated. In every iteration, a Henshin rule is triggered to realize the annotation. Figure 10.24 shows the rule the SMR uses for that. The class and package name are used as input parameters to identify the respective class.

Applying the SMR in this example leads to first recognizing `PrescriptionsDAO` as affected system model class that provides access to the data entity *Prescriptions*. Thus, the co-evolution leads to proposing addition of ⟪lockable⟫ to this class as shown in Figure 10.25.

After a re-run of the SCK transformation, model and respective SCK parts are in sync again and the co-evolution is completed.

10.8 Example 4: Communication using Insecure Encryption

This Security Context Catalog entry supports the usage of secure encryption algorithms in a system. In this application example, we especially consider encryption of communication among nodes in a distributed system. The SCK is used to model the knowledge which encryption algorithms exist and also which ones are potentially endangered and should not be used anymore respectively. We discussed in Section 4.3 (on page 35), how this knowledge can be acquired and incorporated into the knowledge base.

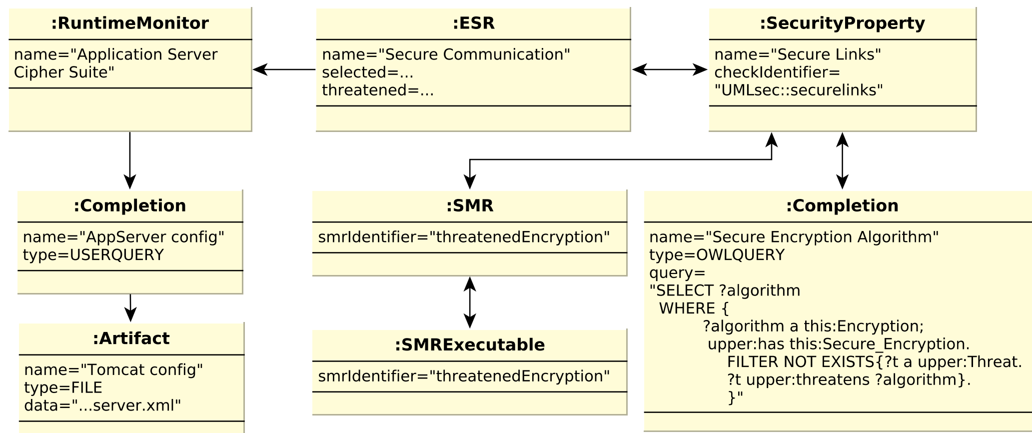


FIGURE 10.26: Security Context Catalog entry for secure communication

Figure 10.26 shows the model of the Security Context Catalog entry. It is based on the UMLsec security requirement *secure links* [JJ05]. The security requirement targets a deployment diagram where all nodes of the distributed system are modeled and dependencies among them show data transfer between the deployed artifacts. Basically, all communication paths need to be annotated with the type of their connection, for instance «LAN», «Internet» (meaning an unencrypted Internet connection), or «encrypted» respectively. In this example, we use an extended version of the stereotype, namely «encrypted enc». It provides additional tagged values (`alg` and `keylength`) to also model which specific encryption algorithm shall be used as well as the key length to be set.

10.8.1 Initial Compliance

The initial compliance is established by modeling a deployment diagram according to «secure links». As the deployment cannot be reverse engineered from the bare source code, this needs to be done manually. Regarding secure communication modeling and «secure links», CARiSMA support can be used to model a deployment diagram according to UMLsec [APRJ17]. Selecting appropriate encryption algorithms is supported by the S^2EC^2O tool's wizard by using the completion element shown in Figure 10.26. The completion element contains an OWL query that is used to find encryption algorithms that are currently not threatened (shown in the lower right of Figure 10.26).

Figure 10.27 shows an excerpt of the SCK. Encryption algorithms are shown in the figure's left hand side, all of them associated to the security property `Secure_Encryption`. This ontology class is used to identify (secure) encryption algorithms. At the time of writing, the S^2EC^2O tool supports key lengths when they are encoded using `::` as delimiter in the SCK individual name. Supporting additional information accompanying ontology individuals is in principal possible using the concept of OWL annotations. In case no delimiter is used, the key length is not relevant.

Figure 10.27 shows two exemplary encryption algorithms: RC4 and AES with 256 bit key length as available encryption algorithms.

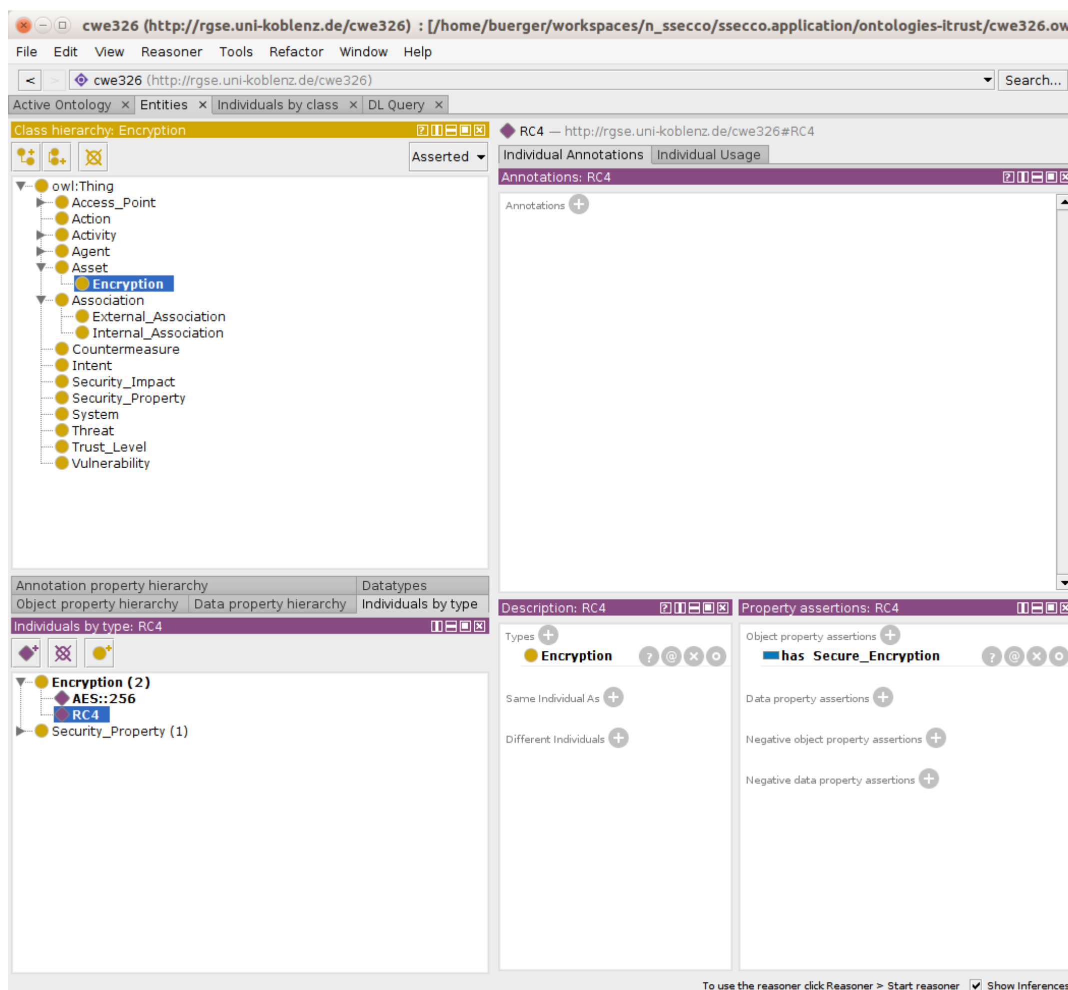


FIGURE 10.27: SCK excerpt describing available encryption algorithms

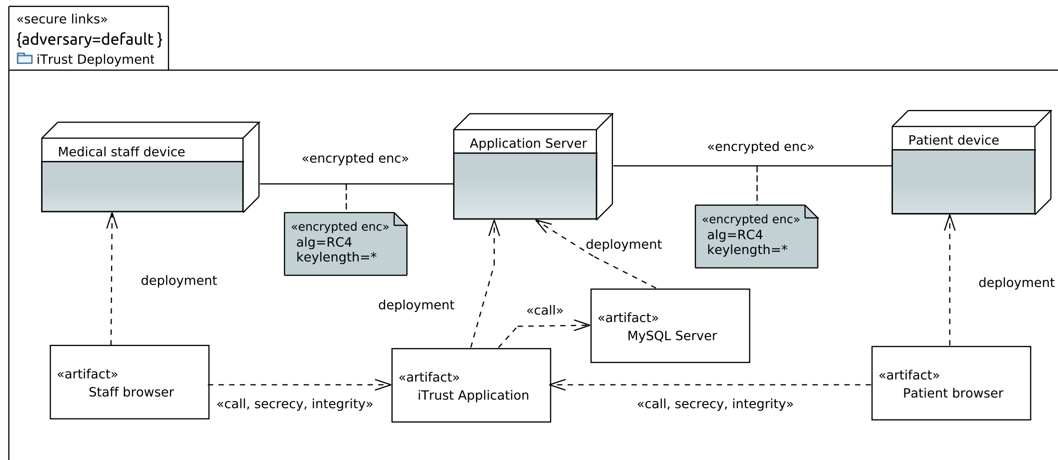


FIGURE 10.28: Deployment diagram for iTrust with ⟪secure links⟫

Figure 10.28 shows a deployment diagram of iTrust. It is already annotated with ⟪secure links⟫ and also has concrete encryption algorithms annotated. Thus, the diagram is the result of the security expert going through the S^2EC^2O tool's initialization wizard and annotating the model accordingly. The deployment is rather typical: there is an application server, executing the iTrust application as well as the database. Apart from that, there are two kinds of devices to act as clients. On the left hand side, a client device of the medical staff running a browser, and on the right hand side a patient's device, also running a browser. The database runs on the same node as the iTrust application, thus does not require communication path encryption. The communication between the server and the clients shall ensure the integrity and the secrecy of the data transmitted over the communication paths.

According to the SCK, RC4 is selected as the encryption algorithm to secure the communication paths between the application server and the client devices' browsers.

We will also sketch how a run-time monitoring for this ESR can be realized. As iTrust is built upon application server techniques, the used cipher suites for HTTPS connections are configured in the application server configuration file. To incorporate this, a run-time monitor needs to be configured with the location of the application server's configuration file.

10.8.2 Context Evolution and Vulnerability

The context evolution in this example is a vulnerability discovery regarding an encryption algorithm. We consider the RC4 encryption algorithm, which was declared insecure after the discovery of severe attacks in 2015 [Pop15]. Thus, systems using RC4 are principally vulnerable and need to be evolved.

Figure 10.29 shows the SCK after context evolution. A threat targeting the RC4 algorithm has been added. Regarding this example, the vulnerability now being present can be found in the CWE database as CWE-327: Use of a Broken or Risky Cryptographic Algorithm[6]. The description reads: *The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected. Well-known techniques may exist to break the algorithm.*

[6]https://cwe.mitre.org/data/definitions/327.html (accessed Apr 11th, 2019)

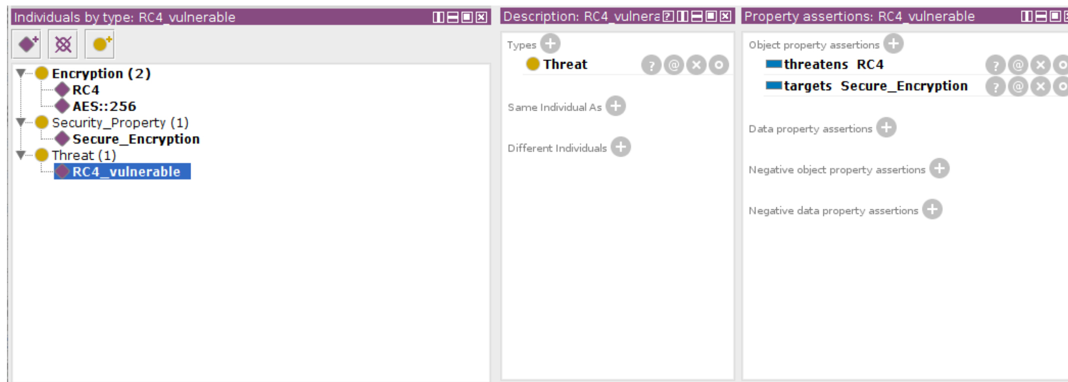


FIGURE 10.29: SCK evolution incorporating threatened RC4 algorithm

10.8.3 Security Maintenance and Co-Evolution

Listing 10.4 shows the pseudo code of the SMR used for this ESR. This SMR is triggered by a semantic diff. Figure 10.30 (on page 158) shows the SiLift rule used to model the diff: A new threat (in the lower middle) is added. Moreover, an object property `threatens` (in the lower left) is added to an already existing encryption individual (in the upper left).

```
1  public void checkConditions(DeltaList deltaList) {
2   currentAlgorithms=queryModelforUsedAlgorithms();
3   alternativeAlgorithms=queryAlternativeAlgorithms();
4   for (SemanticDiffDelta d : deltaList.getDeltas()) {
5    for (OperationInvocation opInv : d.getOperationInvocations()){
6     threatened.add(opInv.getParameter("alg"));
7     if (currentAlgorithms.contains(threatened.getValue())){
8      foreach (b:alternativeAlgorithms AND a:currentAlrogithm)
9       Proposal.addAlternative("Replace"+a+" by "+b);
10    }
11   }
12  }
13 }
14
15 public void apply(Proposal p) {
16  foreach (communicationPath cp) {
17   if (threatened.contains(cp.getAlgorithm)){
18    alterModel(p.getChoice());
19   }
20  }
21 }
```

LISTING 10.4: Pseudo code for SMR co-evolution regarding CWE 327

Regarding the SMR itself, first, all currently used encryption algorithms are queried from the model. Figure 10.31 (on page 159) shows the Henshin rule used to carry out the respective model query (line 2 of Listing 10.4). After that, all alternative algorithms are gathered (line 3). This means: The SCK is queried for encryption algorithms that are not threatened currently. To accomplish this, the same SPARQL query of the SMRs completion is used as it has been used for the S^2EC^2O initialization wizard (see Figure 10.26 on page 154).

The threatened encryption algorithms are gathered from the diff (line 6). For each relevant delta (i.e. match of the `addEncryptionThreat` rule), the parameter `alg` from the rule is requested and added to a collection. This is a parameter of the Henshin complex edit rule (Figure 10.26) and the node `alg` refers to an individual in the SCK representing a now threatened encryption algorithm.

At this stage, it is clear that the delta is in principle relevant and the threatened encryption algorithm in question is also determined. As next step, the SMR checks whether the threatened algorithm is currently also used in the model (line 7). If this is the case, for every possible combination of exchanging this specific algorithm with a non-threatened one, a respective proposal is added (line 9).

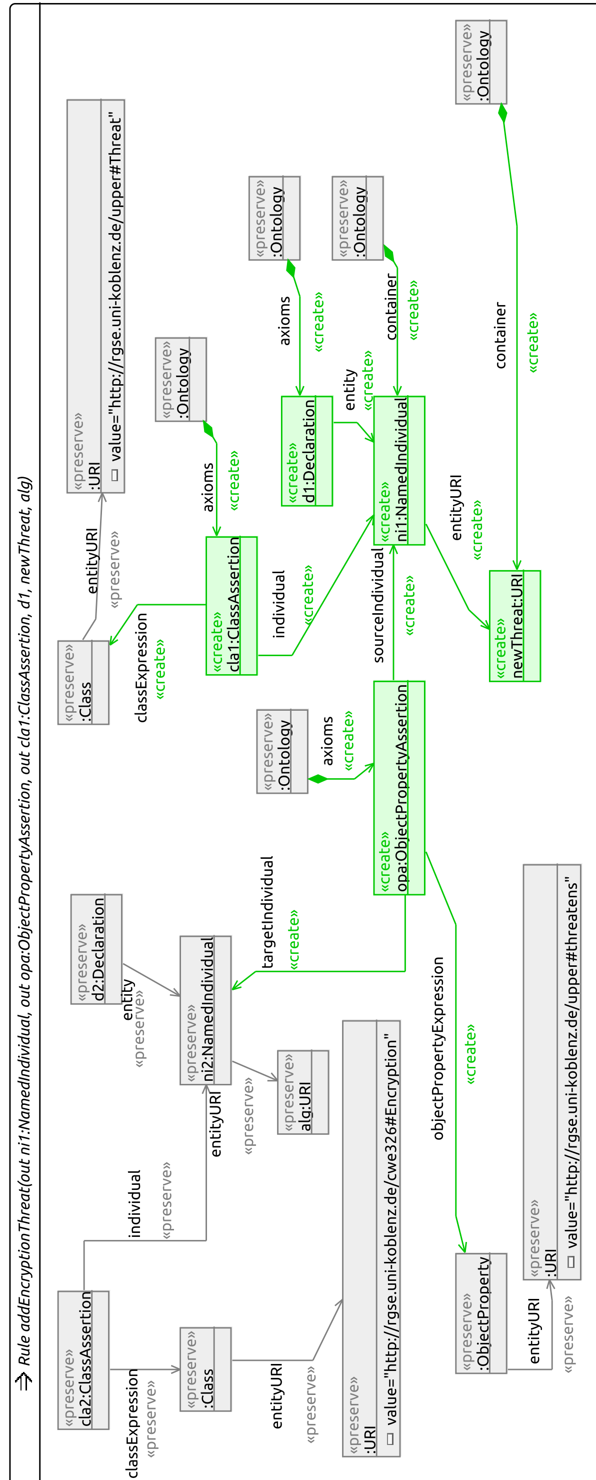


FIGURE 10.30: SiLift complex edit rule to detect addition of a new threat targeting an encryption algorithm

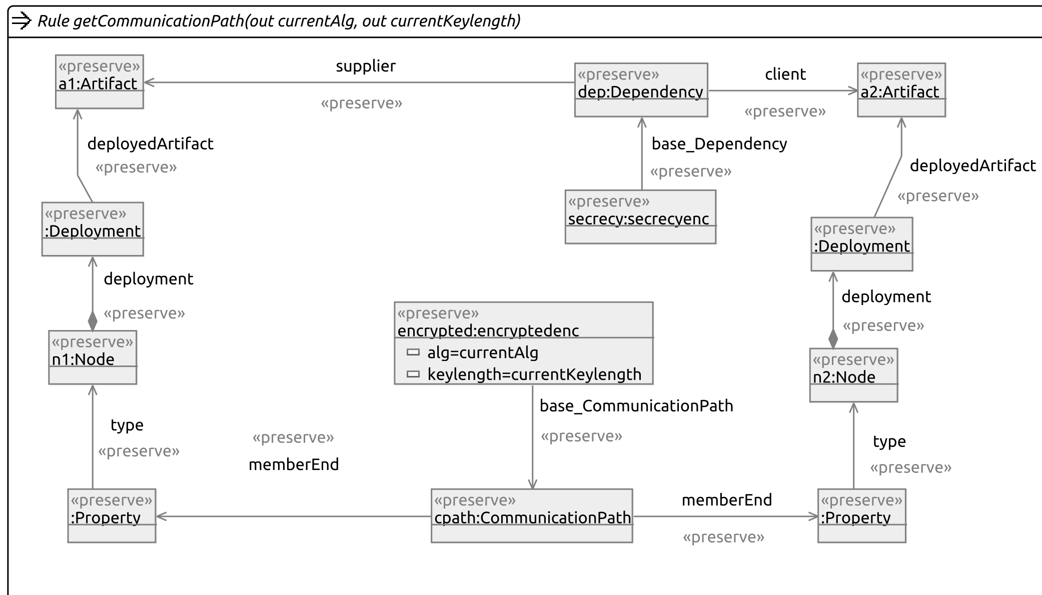


FIGURE 10.31: Henshin rule to query all ⟪encrypted enc⟫ annotated communication paths

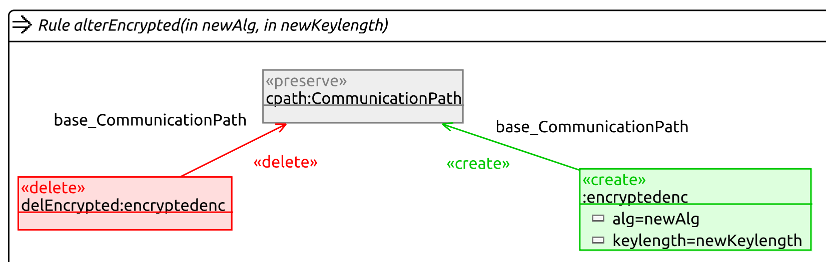


FIGURE 10.32: Henshin rule to alter ⟪encrypted enc⟫ annotations

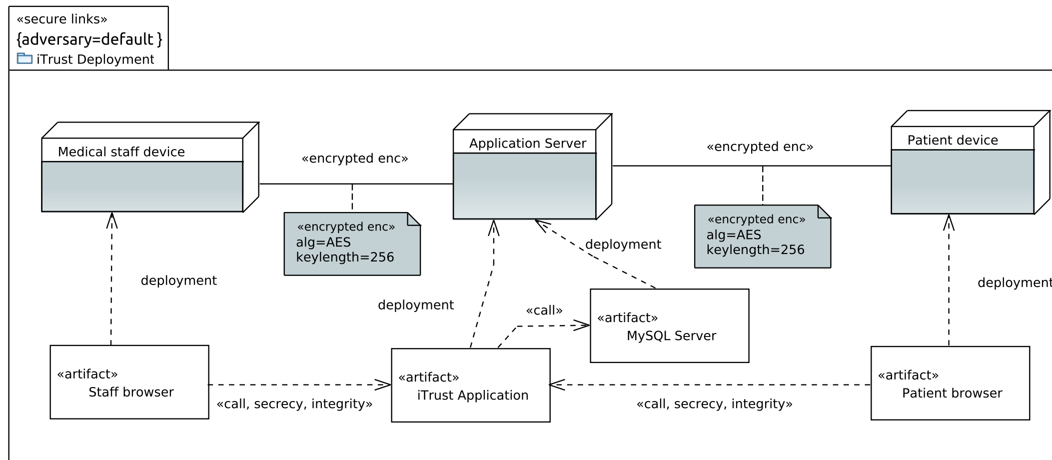


FIGURE 10.33: Evolved deployment diagram of iTrust with ⟪secure links⟫

After the user has made his decision regarding which encryption algorithm should replace which threatened one, the SMR finally is applied (lines 15-20). It iterates over all communication paths of the model (while the set of communication paths in the model has been cached by `queryModelforUsedAlgorithms()`). Every threatened encryption algorithm is replaced according to the choice the user has made. Figure 10.32 shows the Henshin rule used for that. By using cached values from earlier queries, the communication path does not need to be specified with full details or by its name, but we can rather make use of partial match here and directly set the node `cpath` to needed `CommunicationPath` as collected by `queryModelforUsedAlgorithms()` before. The attributes for the new ⟪encrypted enc⟫ are provided as string input parameters for the rule.

Figure 10.33 shows the deployment diagram after evolving it. The algorithm chosen is AES with 256 bit key length. The system design is now co-evolved and adapted to the evolved context knowledge.

In this section, we also consider run-time monitoring of ESRs and, in what follows, sketch a run-time aware SMR. A run-time aware SMR has the advantage of an immediate reaction to a newly discovered vulnerability is possible. In this case, the encryption algorithms as part of the deployment diagram are deployed to the application server by configuring the ciphers it accepts when a client requests a connection.

Listing 10.5 shows pseudo code for a possible additional SMR to interact with a run-time monitor to control the cipher suites used at run time. It works analogously to Listing 10.4 (on page 157) but does not alter the system model. Instead, it requests a S²EC²O run time to change the application server's configuration (line 17). Another difference to the design-time SMR that his SMR collects cipher suites actually in use (line 2).

Currently used ciphers of can be checked either by parsing the application server's configuration file (for Tomcat application server this is the server.xml and the `connector` directive with the `ciphers` tag) or, on a more common ground, using a SSL test suite. For example, the OWASP foundation provides O-Saft [OWA], the OWASP SSL advanced forensic tool.

Changing the supported cipher suites is possible by changing the configuration file accordingly and restarting the application server.

```
1  public void checkConditions(DeltaList deltaList) {
2   currentAlgorithms=queryCipherSuites();
3   alternativeAlgorithms=queryAlternativeAlgorithms();
4   for (SemanticDiffDelta d : deltaList.getDeltas()) {
5    for (OperationInvocation opInv : d.getOperationInvocations()){
6     threatened.add(opInv.getParameter("alg"));
7     if (currentAlgorithms.contains(threatened.getValue())){
8      foreach (b:alternativeAlgorithms AND a:currentAlrogithm)
9       if (getRuntimeMonitor().queryCipherSuites().contains(a)){
10       Proposal.addAlternative("Replace app. server ciphersuite: "+a+" by
            "+b);
11    }
12   }
13  }
14 }
15
16 public void apply(Proposal p) {
17  getRuntimeMonitor().alterCipherSuites(p.getChoice());
18 }
```

LISTING 10.5: Pseudo code for run-time SMR regarding CWE-327

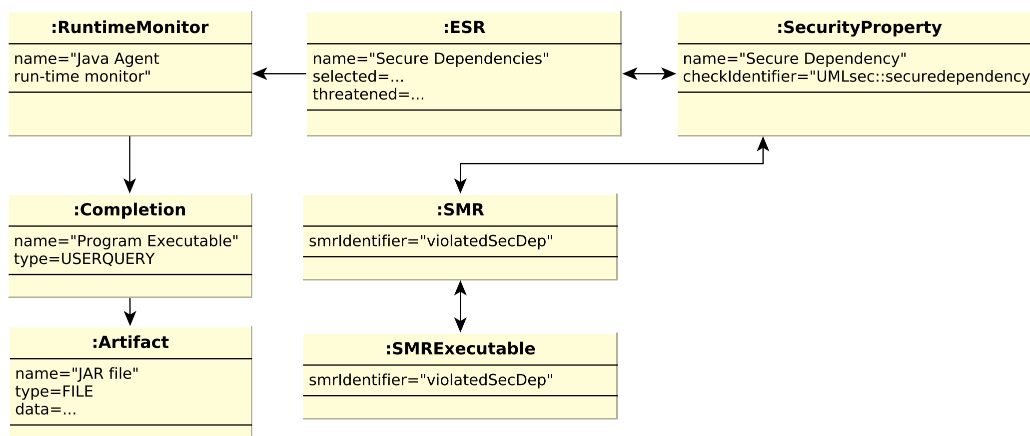


FIGURE 10.34: Security Context Catalog entry for secure dependencies

10.9 Example 5: Secure Dependencies

This Security Context Catalog entry shows run-time monitoring regarding usage of security-critical external libraries as we introduced in Chapter 7. Figure 10.34 shows the model of the Security Context Catalog entry. It is based on the UMLsec security property *secure dependency* [JJ05] as introduced in Section 7.2.1.

As run-time monitor, the run-time component we introduced in Chapter 7 is used.

10.9.1 Initial Compliance

The initial compliance is established by accompanying the class diagram of the system according to «secure dependency». In this example, regarding iTrust, we focus on the prescriptions of a patient.

In iTrust, the only class able of managing prescriptions in the database is `PrescriptionsDAO`. The only class to use it is `EditPrescriptionsAction`. Thus, these two classes are annotated with «critical» and the tagged values `secrecy` and

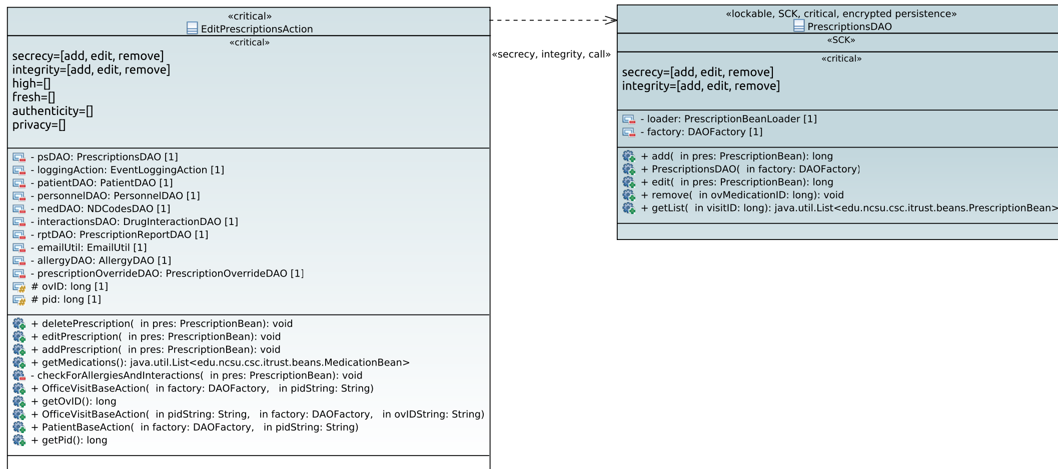


FIGURE 10.35: Classes `EditPrescriptionsAction` and `PrescriptionsDAO` annotated according to ⟪secure dependency⟫

`integrity` according to ⟪secure dependency⟫. Figure 10.35 shows the respective classes including the annotations. Additionally, we assume safe mode support for iTrust as we introduced in Chapter 8.

For example, one kind of safe mode can be to restrict access to clients from inside the hospital network. The effect is to reduce the attack surface while continuing providing the service.

```
1 @SafeMode{
2   redirection="blockExternalClients(thisMethod)",
3   conditions={"Secure Dependencies"}
4 }
```

LISTING 10.6: iTrust SafeMode annotation to block external clients

Listing 10.6 shows a possible safe mode-annotation for redirecting requests. A method `blockExternalClients` can then check the IP address and, if the source IP address is valid, call the original method. In the other case, an error page generation can be issued or an exception can be thrown.

Regarding source code annotations, we focus in this example not on annotated countermeasures but rather on the run-time agent as proposed in Chapter 7 to produce a monitor finding delta at run time, as soon as a violation has been detected.

10.9.2 Context Evolution and Vulnerability

The context evolution in this case is that the application is compromised. For example, the violation can occur because an external library that has been trusted so far happens to be not trustworthy during run time. This can also be the case when an audited (and in this case annotated) library is (maybe in malicious intention) exchanged by a malicious version. Especially two CWE entries come into question here. Firstly CWE-502: *Deserialization of Untrusted Data*[7]. The description reads *The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.*.

Secondly, CWE-20 is also appropriate: *Improper Input Validation*[8]. The description reads *When software does not validate input properly, an attacker is able to craft the input in*

[7] https://cwe.mitre.org/data/definitions/502.html
[8] https://cwe.mitre.org/data/definitions/20.html

a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

10.9.3 Security Maintenance and Co-Evolution

As we discussed in Chapter 7, a number of reactions that are carried out automatically using the S^2EC^2O run-time agent can be defined statically as part of the model and/or the source code. Thus, in this case the co-evolution action is up to the developer. The security expert is informed about the actions that have taken place by the reports the S^2EC^2O run time creates. This information is obtained and displayed as part of S^2EC^2O's delta process introduced in Section 2.2.3 (on page 17).

In this example, we assume that the run-time monitor reports its vulnerability finding using a report with trace data.

```
1  public void checkConditions(DeltaList deltaList) {
2   for (MonitorFindingDelta d : deltaList.getDeltas()) {
3    if (d.getRuntimeMonitor.equals(getRuntimeMonitor())){
4     apply(null);
5    }
6   }
7  }
8
9  public void apply(Proposal p) {
10  getRuntimeMonitor().issueSafeMode("Secure Dependencies");
11 }
```

LISTING 10.7: Pseudo code for SMR co-evolution regarding «secure dependency» run-time violation

Listing 10.7 shows pseudo code for a SMR that triggers a safe mode using its name as safe mode condition immediately. The only condition being checked is the originating run-time monitor (line 3). No proposal is made as the SMR applies itself immediately.

10.10 Performance Observations

The question of S^2EC^2O's applicability is accompanied by the run time of the S^2EC^2O tool. Thus, when conducting the case study, we measured the run-time of the automated parts of the S^2EC^2O tool. The performance measurements were done using the following system:

- Lenovo ThinkPad T450s, Core i5 5200U CPU at 2.20 GHz, 8 GB RAM
- Ubuntu Linux 16.04.6 LTS
- Oracle Java 1.8.0 Update 201
- Eclipse Neon.3 Release (4.6.3)

Table 10.2 shows execution times we measured while conducting the case study. As cases, all of the investigated examples are displayed and referenced by the respective CWE. The only example we omit is the one of Section 10.9 because the focus is not automated monitoring. Regarding S^2EC^2O run time, we measured an average overhead of 7.7x. By repeating the executions, we determined a fluctuation in the run times. Thus, we display the arithmetic mean here. Moreover, since all times are at a level way below threshold of noticeable or disturbing level and for the sake of clarity, the values are rounded.

	CWE-284 (Sec. 10.5)	CWE-311 (Sec. 10.6)	CWE-732 (Sec. 10.7)	CWE-327 (Sec. 10.8)
SCK Transformation	0.8 s	0.4 s	-	-
Explain ontology inconsistency	<5 s*	-	-	-
Reasoning	-	<1 s*	-	-
Sem. Diff of SCK	-	-	14 s	17 s
Load iTrust system model	13 s	19 s	14 s	8.8 s**
SMR Completion	-	-	4.4 s	-
SMR `checkConditions()`	5 s	0.05 s	1 s	3.6 s
SMR `apply()`	0.2 s	0.04 s	0.02 s	0.09 s

TABLE 10.2: Run-time of the S^2EC^2O tool for various parts of S^2EC^2O

Regarding the values marked with an asterisk (*): as we discussed in Chapter 9, ontology reasoning could not be employed due to technical reasons. Instead, we show the times Protégé needs to calculate the respective values in its GUI.

Regarding the value marked with a double asterisk (**): the system deployment could not be obtained from the source code, thus it was not part of the iTrust model. Thus the deployment was modeled in a separate model file.

The structure of the table is as follows: Times are shown where applicable, as, for example, not every SMR makes use of reasoning. Some Security Context Catalog entries require or make use of the transformation of UML elements annotated with «SCK» into the SCK itself. Below the double line, the time used to assess and react to deltas is given, split up into the parts where S^2EC^2O executes steps without user interaction. They focus all on the delta handling phase. The first three lines show the steps necessary to determine the delta. The next line shows the time needed to load the iTrust model, i.e. deserialize it back into EObjects, relating to the EMF UML meta model. In the last three lines, the time consumed by the participating SMRs is shown.

We conclude that, regarding this case study, the S^2EC^2O tool provides an appropriate measure of performance. Using the prototypical implementation that has not been explicitly performance-optimized and a common system, execution times are below any noticeable delay or around a couple of seconds.

Nevertheless, the most time consumption lies in calculating semantical diffs and loading the UML model. We note that both processes need to create a vast amount of (E)Objects at run time and involved plug-ins need to be started and initialized. Several meta models need to be loaded, additionally to the models itself.

While most of these work is up to internal EMF code, we experienced a caching effect: As soon as certain components relevant for these EMF-based tasks are loaded, the times for subsequent process executions drop drastically.

Thus, the overall execution times of the S^2EC^2O tool can be lowered by optimizing against this and fostering that these expensive tasks only are to be executed once.

Chapter 11

Contribution to Research

In the preceding chapters, we introduced the S^2EC^2O approach. We also showed a prototypical implementation as well as application examples as part of a case study.

In Chapter 2, we identified five research questions. In this chapter, we come back to the questions and discuss the contribution this thesis makes to these research questions. We also elaborate on insights gained from implementation of the S^2EC^2O tool as well as the case study.

As a cross-cutting contribution, we introduced two processes, defined using BPMN, that S^2EC^2O is built upon. S^2EC^2O consists of two phases which we reflect with two processes: an initialization phase and a delta-handling phase. For each of both phases, we introduced a process that is used to orchestrate all substeps.

11.1 Review of Research Question 1

RQ1: How can changes in security-relevant context knowledge be used to assess the impact on the system?

To contribute to this research question, we introduced S^2EC^2O's concept of Security Context Knowledge (SCK). We presented ontologies as viable concept to build and maintain a knowledge base. The context knowledge accompanying software systems was outlined and we argued that it makes sense to split security requirements into a common part and a technical part, because the latter is subject to more and more frequent changes.

We introduced a security upper ontology that builds the core taxonomy for the SCK. Based on this ontology, we showed that modularization of the knowledge base is supported by the OWL import mechanism. By using imports, ontologies can be layered to tackle different abstraction levels and aspects of a system in different ontologies that can be maintained separately and exchanged between projects. To query the knowledge for details to reify then template-oriented security requirements, SPARQL queries were introduced as interface. While carrying out the case study, SPARQL proved to be a flexible and easy-to-learn language. Queries can also be build dynamically, for example by generating query prefixes dependent on the currently loaded ontologies. We introduced various publicly available sources for security knowledge.

We presented several existing approaches to incorporate available security (domain) knowledge into the ontology semi-automatically. We presented one approach that solely works on natural language and thus is not tailored to a specific kind of knowledge. Two approaches we showed are tailored to a specific database and take advantage of its meta data and structure.

As S^2EC^2O is focused on analyzing differences in the system context, we introduced two different approaches to analyze the SCK. One approach analyzes the

SCK as is and another one takes different versions of the SCK ontology into account. Using SiLift, deltas between two versions of the ontology can detect primarily structural changes on a semantic level. With reasoners, we showed a way of making implicit knowledge explicit and detect inconsistencies by generating insightful explanations for the user. However, while implementing the S^2EC^2O tool, the work with OWL ontologies showed additional challenges.

The OWL meta model is cumbersome to use. To model an ordinary ontology individual, a big number of objects is necessary, for instance:

1. A class entity,
2. a URI giving a class its name,
3. an individual entity,
4. a URI giving the individual its name, and
5. a class assertion, asserting that the given individual is of a certain class.

In contrast to that, in other meta models like UML, a single node with properties and an association to a parent element (a class associating to its package, for example) is sufficient. Apart from that, nearly all elements of the OWL (EMF) meta model do not have any properties, as virtually every property that an entity in an ontology has, is asserted by associations to further (property-less) elements.

Identification is ultimately possible using URIs. At this point, another challenge arises. A URI of an ontology element mingles the actual name of an individual with the ontology it is coming from. For example, the URI

```
http://rgse.uni-koblenz.de/upper#threatens
```

is the name of an object property. But given the prefix before the hash symbol, the originating ontology is specified using the exact same string. This makes design of, for example, Henshin rules complicated because there is no easy way to express that the hosting ontology of a given individual is not relevant or can be one element of a given set of ontologies. This also applies to the way ontologies are accessed by the OWL API for example. The above mentioned type diversity does also exist considering the classes that we need to interface with OWL ontologies. Basically every type in OWL ontologies is represented by, in this case, a separate Java class, comparable to the situation with the EMF meta model. This makes writing code to analyze ontologies complex and highly inflexible.

Another finding occurred during the implementation of S^2EC^2O is that the approach would be in need of closed world assumption concepts such as local closed-world reasoning. But despite the fact that this is a problem well understood and solutions are described on a theoretical base, at the time of writing, no working implementation was available.

Especially, while carrying out the application examples, it showed that building security-related expressions has often a demand for restricting degrees of freedom for reasoning, for example. In this thesis, we thus only could show examples that work without using concepts like the closed world assumption (CWA).

Apart from that, reasoning was able to infer logical inconsistencies as well as making implicit knowledge explicit with no notable delay. This can be of great value in projects with a complex requirements structure.

11.2 Review of Research Questions 2 and 3

While conducting the research for this thesis, it turned out that RQ2 and RQ3 have to be treated jointly. Thus, we also review both RQs together.

RQ2: How can rules be formalized that are able to preserve the system's security given knowledge evolution?

S^2EC^2O focuses on reacting to evolution of a system's context. Hence, it is necessary to ensure that a system is compliant with its security requirements prior to evolution. This is supported by the concept of Essential Security Requirements (ESRs). We presented them as part of a catalog, being technology-independent security requirements. ESRs can be selected in a lightweight way and bridge between ordinary software engineering and a system that can be accompanied by S^2EC^2O. We introduced a meta model for the Security Context Catalog. It includes ESRs and links selected ESRs, to security properties that can be checked using existing tooling, for example. S^2EC^2O in this case makes use of but is not limited to UMLsec checks, the UMLsec tool platform CARiSMA and ontology reasoning. The Security Context Catalog further is related to Security Maintenance Rules (SMRs). These rules realize the Event-Condition-Action principle. They are used to determine appropriate co-evolution steps when a context evolution has taken place. We introduced a meta model for SMRs. Using the concepts of Essential Security Requirements and Security Maintenance Rules, we can leverage context evolution and determine co-evolution actions. With the concept of *Completion*, usable both by ESRs and SMRs, user interaction is possible and thus providing system-specific information to complete the rather generic knowledge in the SCK.

RQ3: How can these rules be used to carry out a semi-automatic co-evolution given a context evolution?

To coordinate and formalize the context evolution information, we introduced the *Delta* meta model. It contributes to RQ3, as it provides a possibility of structuring all different kinds of delta sources. With the concept of *proposals*, SMRs can propose co-evolutions they offer to co-evolve the system. S^2EC^2O as well as the user can then decide which proposal(s) to choose. Regarding S^2EC^2O, we introduced a *Delta handling* algorithm which shows how the whole co-evolution process according to the S^2EC^2O delta process can be carried out.

While carrying out the case study, the technologies supporting SMRs, namely Henshin, SiLift, and the possibility to issue SPARQL queries, lowered the amount of source code to be written. For example, in Section 10.8, the SPARQL query as the SMR's completion element is used both for building the initial compliance of the system as well as carrying out the co-evolution.

Regarding the ESR implementation, a challenge with regard to EMF arose. The Security Context Catalog is designed as a Ecore model, it can thus be natively serialized as XML file. However, Ecore does not allow to have different implementations of a operation of the same class. This contradicts the S^2EC^2O tool's requirement, as every SMR implementation needs to have its own implementation of the methods `inspectSystem()`, `checkConditions()`, and `apply()`. However, a purely Java-based implementation of SMR would also not solve the problem because then the SMRs would not have been part of the Ecore model anymore. We solved the problem by splitting SMR in one part being serialized and a second part, `SMRExecutable`, incorporating the method implementations for a given SMR. Both are coupled using a

identifier string. During run time, available `SMRExecutable` instances are gathered and instantiated using Eclipse's extension point mechanism.

While we carried out the case study, it showed that there is no technical need for having the `inspectSystem()` method. The application examples showed that both steps can be performed at the same time or even have to because they are mingled.

11.3 Review of Research Question 4

RQ4: How can information coming from the system execution be used, to assess the quality of the security requirements compliance during run time?

The S^{2}EC^{2}O run-time approach we presented in Chapter 7 builds upon reverse engineering of models from code and security annotations in model as well as in code. UMLsec Java annotations support security annotations at source code level and thus bridge the gap between the system model and executable code. Countermeasures which can take place as soon as security violations are detected are supported by call trace logging and also by providing modified return values to protect real application data.

Furthermore, the security expert can gain insight regarding the system's security at design level by feeding back gathered run-time data into the model, thus supporting round-trip engineering. Adapting the model with newly discovered calls and classes dynamically loaded during run time can support understanding how security attacks are accomplished. Apart from that, this can also help finding differences between the system model and source code. A sequence diagram bound to the UML model elements can be generated to graphically show a sequence of actions during a detected violation. Thus, system evolution detection is also tackled.

The approach presented in Chapter 7 realizes support for checking secure call dependencies by extending the realization *secure dependency* for the UMLsec extension which could only be checked statically (and thus partially) before. In tests, the Java agent run-time monitor showed an average slowdown of the factor 7. While this provides a degree of performance that is sufficient for proof-of-concept, performance needs to be improved for potential practical usage.

11.4 Review of Research Question 5

RQ5: How can information gathered at run time be used to adapt the system when the context evolves, avoiding shutdown or additional design cycle?

The run-time adaptation approach we introduced in Chapter 8 contributes to this research question by extending the run-time monitoring approach we introduced in Chapter 7. The extension concerns run-time adaptation. We proposed an interface between the run-time agent and the Security Context Catalog (S^{2}EC^{2}O service).

It is used to react to context evolution because S^{2}EC^{2}O tool can trigger adapted behavior at run time whenever ESRs are considered threatened when processing SCK deltas. We introduced a `@SafeMode` annotation, so that the security expert is enabled to specify redirection methods which should be called alternatively as soon as ESRs are tagged as *threatened*.

Furthermore, we specified a set of JSON objects to be used by the run-time agent and the S^{2}EC^{2}O tool for synchronization and control purposes. This way, a system shutdown is avoided, saving time until design, implementation and test of a permanent solution is issued.

Chapter 12

Conclusion

Software systems become increasingly complex. While more and more parts of our daily lives are touched by information systems in particular, also a growing number of originally hardware-based systems are replaced by software-defined solutions. All in all, critical systems are complex and/or costly, so that they are not to be replaced often and thus have to be maintained over a long period of time. Nevertheless, these systems are not used in an isolated manner but communicate with each other. As these systems control crucial parts of our daily lives and/or process sensitive data, providing secure software is inevitable. However, security vulnerabilities that might be disclosed in the future cannot be foreseen when the system is initially designed. This thesis makes a contribution to accompany development and maintenance of long-living software systems. The presented approach supports security experts in assessing impact to security requirements when context evolutions affecting the security take place and adequately react to them.

Chapter 1 motivates the overall work and identifies challenges to be solved.

Chapter 2 presents the research road map of this thesis. Proceeding from the identified challenges, we formulate a number of research questions and give an overview of the approach presented in this thesis, *S²EC²O*. We show the components of S²EC²O and how they relate to each other, model-driven software development, and its environment. We further introduce the S²EC²O process, consisting of an initialization phase to make a software system *S²EC²O aware*, followed by a delta handling phase that reacts to context evolutions by assessing and eventually applying appropriate co-evolutions.

Chapter 3 presents foundations on which S²EC²O is built upon and also elaborates on the SecVolution research project, which is related to S²EC²O.

Chapter 4 elaborates on the question of how knowledge relevant for a system's security context can be incorporated in a semi-formal manner. It introduces the notion of Security Context Knowledge (SCK) and the security upper ontology providing the foundation of SCK. We present potential sources of security-relevant knowledge as well as techniques to incorporate them in SCK. We show a technique to connect UML system model elements to the SCK ontology as well.

Chapter 5 focuses on the question of how changes to a system's context can be used to support the design of a secure system model. Semantic differencing can be used to detect complex changes between two versions of the SCK. Ontology reasoning provides possibilities to gain insight from a given ontology, namely context changes leading to an inconsistent ontology and inferring additional knowledge based on inference.

Chapter 6 shows how a system can be co-evolved, as soon as a context evolution is detected. Rules to co-evolve the system's security are introduced (Security Maintenance Rules, SMRs) as well as a notion of technology-independent security requirements (Essential Security Requirements, ESRs). We introduce the Security Context

Catalog to support coordination of ESRs and SMRs. A delta information meta model organizes context evolution information and proposals for co-evolution operations. We introduce a delta-handling algorithm that implements S^2EC^2O's delta handling process.

Chapter 7 shows how security requirements defined in the system model at design time can be carried further to the source code level. Based on an already existing secure modeling extension, Java source code annotations are defined and run-time related features are added. We present a run-time monitoring approach that supports dynamic method call re-assignment in case a violation is detected. Round-trip engineering is also supported by extending the system model with information gathered at run time.

Chapter 8 elaborates on how a system's security can be adapted during run time. We propose an approach of the run-time agent communicating with the core of S^2EC^2O by exchanging JSON messages (S^2EC^2O service). Furthermore, we introduce a concept for a safe mode to dynamically alter the system's behavior in case a security vulnerability emerges at design time. This reduces the attack surface and thus lowers the risk for further exploitation while avoiding a system shutdown, which may have even worse impact.

Chapter 9 presents the S^2EC^2O tool, a prototypical implementation of S^2EC^2O. At first, we introduce S^2EC^2O tool's architecture. After that, we show its components in detail and relate them to the S^2EC^2O process steps they realize.

Chapter 10 constitutes a case study. For a number of realistic security vulnerabilities and context evolutions, coming from law changes and also technical vulnerabilities, S^2EC^2O is applied to the medical information system *iTrust*. Five cases show on the basis of real artifacts exemplary, how the system can be made secure initially, how the context evolution is detected and how the co-evolution is carried out.

Chapter 11 reviews the research questions we specified in Chapter 2. We discuss, how this thesis contributes to these questions.

Regarding this chapter, in Section 12.1, we put emphasis on which contributions this thesis makes with regard to the research objective and the challenges defined in Chapter 1. Section 12.2 shows, to what extent the results of this thesis can be generalized and what are actual limitations we identified. Section 12.3 focuses on open challenges on a coarse-grained scale and gives directions for possible future research and concludes this chapter and therefore this thesis.

12.1 Contributions

In Chapter 1, we motivated the research for this thesis, formulated a research objective, and identified a number of challenges.

The research objective is as follows:

> We want do develop an approach that supports maintaining the security of long-living software. Security-relevant knowledge from various sources, in various forms, and at various abstraction levels shall be captured and managed. The knowledge shall be used by the approach to assess, if the security of a given system is compromised, given a change of the security context knowledge. If this is the case, the approach shall support revising the system, so that it fulfills its security requirements again.

In the remainder, we put emphasis on the contributions S^2EC^2O makes, relate them to the challenges we identified and thus show how S^2EC^2O fulfills the research objective.

C1: Leverage evolution of context knowledge

Regarding *C1: Leverage evolution of context knowledge*, we showed how the security upper ontology first introduced by [Gär16] can be used to store common security knowledge. By introducing the concept of the SCK in a layered manner, we foster modularization of the knowledge base and thus exchange between projects and central management.

We introduced UML annotations to bridge the gap between the system-unspecific ontology and the concrete system design. We realize leveraging context knowledge evolution by employing a number of methods, namely ontology reasoners and semantic differences. These realize making implicit knowledge explicit and also find subtle changes in a difference between two versions of an ontology.

C2: Infer concrete security requirements

Regarding *C2: Infer concrete security requirements*, we introduced the separation of ordinary (security) requirements into Essential Security Requirements and information provided by the SCK. This enables security experts and stakeholders to define security requirements independent of a given domain and especially without temporal dependencies. Thus, security requirements, which are originally system-specific, do not need to be adapted to an evolving environment anymore. The only part that needs to be changed is considering time- and domain-dependent information incorporated by the SCK. This part can be managed independently from given projects, thus gaining synergy effects.

C3: Assess impact of context evolution to a given system

S^2EC^2O not only supports the security expert by identifying evolutions of security-relevant knowledge, but also by investigating the system with regard to the impact a context evolution may have. This relates to the challenge *C3: Assess impact of context evolution to a given system*. Using the delta model we introduced, context evolution information can be leveraged and treated as required, according to their type and origin.

By employing techniques like model queries, S^2EC^2O is able to inspect the system under consideration, to determine if a given evolution is relevant. Model queries are also used to calculate co-evolution alternatives, if no additional input from the user is needed at this point.

We introduced Security Maintenance Rules to formalize the connection from context evolution and system co-evolution in terms of rules. We further introduced the Security Context Catalog. In the Security Context Catalog, all elements regarding the system co-evolution are connected as needed. Thus, mappings between Essential Security Requirements, security properties and Security Maintenance Rules are defined here. Thus, the Security Context Catalog contains the guidelines, S^2EC^2O uses to investigate the system and eventually co-evolve it.

C4: Co-evolve system design to preserve its security

With regard to *C4: Co-evolve system design to preserve its security*, we introduced a delta-handing algorithm. It orchestrates triggering of the SMRs and involves the security expert when necessary. The Security Context Catalog provides completion elements to indicate these cases.

C5: Assess system's security during run time

With regard to *C5: Assess system's security during run time*, S^2EC^2O features round-trip engineering. First of all, the gap between the system model and executable code is bridged using standard Java annotations. S^2EC^2O supports deploying countermeasures to security violations directly into the executable code. Information about the system design discovered at run time can be fed back into the design.

C6: Adapt the system to preserve security during run time

Regarding *C6: Adapt the system to preserve security during run time*, we extended S^2EC^2O to not only have countermeasures statically as part of the code, but also support adaptation at run time. We introduced safe-mode adaptation points that realize a fine-grained adaptation of the system behavior relating to the currently known threats as defined by the SCK and stored inside the Security Context Catalog.

Unique Contributions

In large part, we built tool support for S^2EC^2O. We showed the applicability of S^2EC^2O and also the S^2EC^2O tool by conducting a case study using the medical information system iTrust.

S^2EC^2O is a unique approach because of two reasons. First, it covers a wide range of development artifacts. By incorporating common security knowledge, S^2EC^2O treats relevant aspects even before development of a software starts. While focusing on model-based software engineering and using a system model as main design artifact, S^2EC^2O acknowledges the fact that manually built code exists and is able to also incorporate it. Finally, round-trip engineering is supported by having insights gained at run time put into the design level.

The second aspect of S^2EC^2O to make mention of is that it is entirely designed as a lightweight extension to a mature software engineering process and does not make strong assumptions about the artifacts involved or the architecture of the system that is supplemented by S^2EC^2O. It is able of incorporating natural language sources to build the SCK instead of requiring a formal requirements representation. A given UML system design can be used with S^2EC^2O just by extending it by the respective annotations. Finally, S^2EC^2O works with ordinary Java source code, as its annotations make use of the standard, lightweight Java annotation mechanism.

S^2EC^2O is defined by two processes. The initialization process is used to equip a given system design with the necessary annotations and meta data to make it *S^2EC^2O aware*, regardless if it is a legacy system or greenfield development. The delta-handling process then defines how to react to the continuous stream of delta information.

Finally, we conclude that S^2EC^2O fulfills the research objective by contributing to all research questions and challenges respectively. Security experts can use the approach to have the maintenance of long-living systems accompanied. While managing the SCK, S^2EC^2O determines changes to the knowledge, determines evolutions

relevant to the system under consideration and guides the user in appropriately co-evolving the system design. Additionally, the run-time phase is supported by monitoring security requirements at run time and equipping code with countermeasures in case a violation is detected. While being a semi-automatic approach, S^2EC^2O frees valuable time from security experts. The SCK and the Security Context Catalog were built in a way that managing this data once for a multitude of projects is conceivable, gaining additional synergy effects.

12.2 Assumptions and Limitations

In this section, we discuss the generalizability of S^2EC^2O and limitations we identified during research.

12.2.1 Generalizability Considerations

To the best of the author's knowledge, the applicability of S^2EC^2O for certain types of systems is only limited by, for instance, the expressiveness of the accompanying approaches it makes use of. In other words, S^2EC^2O is principally applicable to any system that can be described by a UML model, and for security properties that can be checked either during system design time or at run time, as we discussed in Sections 6.5 (on page 73) and 7.2.1 (on page 80). Especially, the S^2EC^2O tool is designed in an open way. If a case of limiting expressiveness comes up, it is worth considering to extend S^2EC^2O by incorporating a more expressive approach for the limiting aspect. In particular, the Security Context Catalog can be extended by further entries to cover additional aspects. For example, S^2EC^2O could be extended to support additional kinds of knowledge bases, or further modeling languages.

12.2.2 Scalability Considerations

Regarding the scalability of S^2EC^2O and the S^2EC^2O tool respectively, our case study showed that, using an average desktop system, running the S^2EC^2O tool to investigate iTrust does not show an arguable delay. As iTrust is an example of a functional information system, S^2EC^2O can apparently be used for a software system in realistic size. The performance of S^2EC^2O mainly relies on the performance of the underlying technologies like SiLift and Henshin for model queries, model differencing, and model co-evolution. For ontology analysis, reasoners are also used. For all of these aspects, unmodified tooling is used, so in case of performance issues, variants with better performance could be used.

In the current implementation of the S^2EC^2O tool, the usage of EMF models is not optimized. In case the size of Security Context Catalog raises performance issues, the implementation could be adapted to persist EMF models in databases.

However, an actual performance issue exists with the run-time monitoring approach. The current implementation's overhead is too big to use it in everyday situations. A more efficient run-time monitoring implementation is necessary.

12.2.3 Limitations of the Case-Study Results

A threat to the validity of our case-study results is that the case-study only considered one specific system. Thus, the prototypical tool support is also evaluated using test cases and this specific system (iTrust).

To reveal the eventual need for adjustments of S^2EC^2O, considering additional systems, also from other domains and/or in conjunction with industrial partners, seems reasonable.

Moreover, the ESRs and the Security Context Catalog we showed in in Chapter 10, examine a number of specific use cases and do not make a claim to completeness of contents. However, the presented Security Context Catalog can be extended to support further security properties, domains, etc., similar to the idea of the UMLsec approach [JJ05].

By using an appropriate system in terms of size, longevity, applicability, and security-relevance, we showed that S^2EC^2O is applicable, and, by providing the S^2EC^2O tool, we laid the groundwork for extending the Security Context Catalog and realize subsequent studies. In the succeeding section, we focus on the current set of ESRs supported by the Security Context Catalog and how to extend it.

12.2.4 Support for Additional Security Properties

By now, the Security Context Catalog covers a limited set of security properties. As the catalog is designed openly, support for further security properties can be added by adding the respective catalog entries.

Regarding source-code level annotations and run-time monitoring, S^2EC^2O currently supports one security property, namely the UMLsec requirement «secure dependency». In this case, the check for this stereotype's violation needed to be implemented tailored to this specific security property. From a syntactical point of view, adding support for further security properties is straightforward. Basically, appropriate Java annotations and a UML profile with matching stereotypes need to be defined. However, the question is where in the source code information for the respective property can be found to check its validity at run time. Regarding the original set of security properties defined by UMLsec [JJ05], «secure links» needs information about the system deployment. Typically, there is no clear way of how this information manifests in a system, even in source code. In case the deployment is not part of the source code, the run-time agent as we introduced is not sufficient and needs to be extended at a conceptual level.

Referring to the assumptions for using S^2EC^2O, we gave in Section 2.2.1 (on page 11), a user who wants to provide support for additional ESRs, needs to provide additional expertise. In case of ontology reasoning should be used to detect context evolutions in the SCK, the user needs to provide knowledge about how to design an ontology to foster reasoning methodologies.

Regarding design of semantic differences, model queries, and graph-transformation based co-evolutions, the user needs to use the graph-transformation tool Henshin.

To query the SCK, the implemented and preferred way currently is to conduct SPARQL queries. Thus, the user needs to be able to specify these queries.

An additional Security Context Catalog entry is added by editing the Security Context Catalog Ecore model. The editor plug-in generated by Eclipse does not require additional domain knowledge, as it, for example, can be used with a tree-based editor Graphical User Interface (GUI).

Finally, the user needs to provide a `SMRExecutable` class by implementing the `checkConditions` and `apply` methods. The SCK can be used by its generated API from the EMF. As the tool architecture in Section 9.1 (on page 109) shows, we encapsulated complex services like reasoning, graph transformations, etc., so that no detailed knowledge about these APIs is required.

12.2.5 Support for Further Programming Languages

By now, S^2EC^2O builds upon object-oriented programming and Java explicitly. S^2EC^2O could also support other object-oriented programming languages. Nevertheless, S^2EC^2O, for example, assumes that data is encapsulated in classes and, for external callers, only accessible via getter methods. While access readability restrictions can be in principal be circumvented using Java reflection, violations like this can be detected during run time (as we showed). For programming languages that allow to circumvent data encapsulation like this, for example C++, additional static or run-time checks may be necessary. Basically, this guideline also holds regarding usage of S^2EC^2O targeting a non object-oriented programming language.

12.2.6 Support for Further Software Engineering Methods

S^2EC^2O currently is a lightweight extension to an already existing software engineering approach, namely model-based security engineering. This software engineering method relies on the following artifacts: a set of requirements as result of a requirements elicitation process and a system model, adhering to the requirements. Source code is then generated more or less using the model as starting point. S^2EC^2O currently refers to related approaches that realize examination of natural-language requirements regarding security relevant terms. By sacrificing this option and demanding manual selection of relevant ESRs, a software engineering method not involving requirements engineering could be supported.

Regarding the UML(sec) model, S^2EC^2O could be extended to support other modeling languages, assuming they are not less expressive than UML. The features used by S^2EC^2O that are specific to UML are annotations and UML profiles. However, S^2EC^2O needs the user to annotate the system model with data relevant for security requirements. As long as an other modeling language also supports this, extending S^2EC^2O into this direction seems feasible. The question arises which other software engineering approaches that can be loosely compared to or used with model-based engineering, also feature mature methodologies to incorporate security engineering. For example, in agile software engineering, no stable and widespread processes for security testing exist by now [CFO+17].

12.3 Future Work

While conducting the research for this thesis, we identified additional challenges and research questions regarding the problem of system co-evolution under context evolution: These can be used as starting points for follow-up work. We highlight a number of open questions.

12.3.1 Regard Evolution of ESRs and SMRs

S^2EC^2O is built to react to context evolution by co-evolving the system under consideration. The methodology we presented for this is based on Security Maintenance Rules and Essential Security Requirements. While ESRs help designing a secure system at the first place, SMRs contain strategies to recover certain aspect of the system's security.

It sounds reasonable that the connections between security requirements and security properties may evolve. Moreover, co-evolution strategies may change. While

evolution of ESRs and SMRs may provide support for additional security requirements, run-time monitors, etc., it is also possible that knowledge in these structures is to be refactored or components even are to be removed.

However, S^{2}EC^{2}O currently does not deal with this. One challenge here is that removing elements may cause parts of the system to become insecure because a co-evolution that was known to recover an ESR is discovered to be insecure. This means that currently implemented co-evolutions or deployed monitors must be revised.

A possible way to tackle this would be to handle these evolutions as an additional form of context evolution events.

12.3.2 Share security and co-evolution knowledge of S^{2}EC^{2}O publicly among projects

One goal of S^{2}EC^{2}O is to lower the effort security experts have when they maintain long-living software systems. As one contribution, the Security Context Knowledge offers a possibility to manage knowledge about security best practices, regulations, vulnerabilities, and mitigations. ESRs and SMRs further help to semi-automate the process of context-driven co-evolution.

The needed effort for maintenance can be lowered even more when there is support for sharing the knowledge and data between different projects or domains. As we described in Section 4.2.2 (on page 33), the way ontologies, as the base of SCK, are used in S^{2}EC^{2}O already support this idea. With having the used knowledge base built by a layered ontology stack realizes modularization.

To tie a system to S^{2}EC^{2}O, it is necessary to provide system-specific details. At least, the information which ESRs shall be fulfilled, needs to be stored. Moreover, further details necessary for ESRs and SMRs to work need to be provided as required by the `Completion` elements. Using this kind of realization, establishing additional trace links between the system and S^{2}EC^{2}O's data is avoided.

A vision is to have ready-to-use versions of SCK parts, ESRs and SMRs, tailored for different domains and application scenarios. These could be maintained by the community, comparable to the CVE and CWE databases.

Currently, SMRs and ESRs do not provide a mechanism for modularization. They could be distributed by just resetting all system-specific aspects to their defaults. However, support for evolving SMRs and ESRs would still be necessary then.

12.3.3 Consider Implementation-Specific Security Vulnerabilities

In its current form, S^{2}EC^{2}O accompanies the model-based design approach. That means, there is roughly the order from requirements to models, to code, and finally code that is executed.

When a problem is detected at the source code level, the development process is run through again. Even if some artifacts do not need to be changed, every level at least needs to be checked.

Given by its nature, the focus of S^{2}EC^{2}O is to support security by design, thus promoting and demanding that security requirements are already treated at the design phase. This is the suitable approach for many security properties. Nevertheless, naturally, a design model has a more coarse-grained abstraction level than the actual source code. However, many security vulnerabilities come up at implementation level, for example deserialization vulnerabilities and improper validation of inputs.

S^{2}EC^{2}O currently only supports linking security checks at the model level. There is only one bridge from security requirements to the system model. Thus, S^{2}EC^{2}O

could be extended to support security annotations, properties, and checks that are specific to the implementation level directly. This additionally would require to align the security design support coming from the model level and that defined at the implementation level.

12.3.4 Investigate Influence of Security Co-Evolutions on Functional Requirements

S^2EC^2O, as presented in this thesis, determines a number of co-evolution operations and lets the security expert choose appropriate alternatives. A full security check of the system can be used to ensure the full compliance to the security requirements. However, it is currently not aligned with possible functional requirements. A series of security co-evolution operations may have impact on the overall system's requirements.

This could be tackled by involving continuous integration techniques and ordinary testing in general to estimate effects on the system as a whole upfront.

The work in hand lays the foundations for the research directions introduced above.

Appendix A

Preliminary Publications

This thesis builds upon preliminary work, mostly in conjunction with project members of the SecVolution project. This section lists them, briefly states what their content is, and states what the contribution of this thesis' author is.

- Jens Bürger, Jan Jürjens, and Sven Wenzel: Restoring security of evolving software models using graph transformation.
  In: *STTT*, 17(3): 267–289, 2015. [BJW15]

  This publication presents work on using security anti-patterns to check if a given UMLsec model is compliant with its security requirements. It basically stems from the author's diploma thesis. Beyond that, a performance measurement of the prototypical implementation has been laid out, showing that it is faster than manual model inspection.

  Basic work on how the approach can be applied to evolution scenarios using model differencing approaches is presented. The contribution of model manipulation using graph transformation techniques is used as the technological base in S^2EC^2O when it comes to co-evolution of system models.

- Thomas Ruhroth, Stefan Gärtner, Jens Bürger, Jan Jürjens, and Kurt Schneider: Versioning and evolution requirements for model-based system development.
  In: *CVSM 2014*,
  volume 34/2 of *Softwaretechnik-Trends*, pages 20–24, 2014. [RGB+14a]

  This publication ascertains that, to support evolution in model-based development, difference calculation on a semantic rather than syntactic scale is necessary.

  With having semantic differencing of knowledge evolution in mind, triggering actions to co-evolve models was considered the next step. Here, the *Security Maintenance Model (SMM)* steps in. The author of this thesis presented first details on it like a rule mechanism. This work evolved into *Security Maintenance Rules (SMRs)* in subsequent publications. The concept of SMRs is also used in S^2EC^2O. Moreover, this paper is the first publication featuring an overview of the SecVolution approach, to which the author also contributed.

- Stefan Gärtner, Thomas Ruhroth, Jens Bürger, Kurt Schneider, and Jan Jürjens: Maintaining requirements for long-living software systems by incorporating security knowledge.
  In: *RE 2014*. IEEE, 2014. [GRB+14]

  This publication's main idea is to analyze natural-language project artifacts (for example, use case descriptions) regarding their security relevance. A

heuristics-based approach incorporating Natural language Processing is presented to extract security knowledge from use case descriptions. Security incidents are provided in terms of misuse cases. Security knowledge gathered this way is captured as part of an ontology, whereby its core structure resulted from a literature research of publications concerning security ontologies and taxonomies. The work has also been presented by the first author at the German software engineering conference *SE* [GRB+15], to present updated results and gain additional feedback.

The author of this thesis contributed to the case study by modeling the ontology Security Context Knowledge. Moreover, an analysis of the iTrust requirements regarding their security relevance and interconnections was carried out. The work conducted on the iTrust case study made use of experience gained from a prequel study on the Palladio Component Model (PCM) based realization of CoCoME [HKW+07] (one of the case studies inside the DFG priority programme) which is not part of the publication. For S^2EC^2O, iTrust is also used for evaluation. Moreover, the model of the upper ontology used to describe security notions is reused.

- Jens Bürger, Jan Jürjens, Thomas Ruhroth, Stefan Gärtner, and Kurt Schneider: Model-based security engineering with UML: Managed co-evolution of security knowledge and software models.
  In: *FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *LNCS*, pages 34–53, 2014. [BJR+14]

  This article gives an overview of security analysis in model-based software engineering. Furthermore, the SecVolution approach is presented in more detail, accompanied by a case study featuring a web shop extension of CoCoME. Security considerations are modeled using UMLsec annotations for secure information flow (SIF) [RJ12].

  This thesis' author main contribution to this article is in the co-evolution of UMLsec models. Details of the Security Maintenance Model are introduced, including a formalism for Security Maintenance Rules. The rules are exemplified using the CoCoME case study and a standard UMLsec check concerning data integrity. A co-evolution to the security issue is also shown. Furthermore, the author contributed to the design of the web shop extension and created its model as well as their SIF annotations. The Security Maintenance Model and rules as presented are incorporated by S^2EC^2O.

- Thomas Ruhroth, Stefan Gärtner, Jens Bürger, Jan Jürjens, and Kurt Schneider: Towards adaptation and evolution of domain-specific knowledge for maintaining secure systems.
  In: *PROFES 2014*, volume 8892 of *LNCS*, pages 239–253. Springer, 2014. [RGB+14b]

  This publication presents an approach to the question, how layered ontologies can be re-integrated after evolution took place. SecVolution's concept of supporting knowledge that can be expressed in different layers of refinement (from global to domain to system) is published.

  The author of this thesis contributed to the paper by analyzing privacy laws and their evolution. In the end, the result is a setting for a changing system context (evolution) which makes a system (co-)evolution necessary. The privacy setting is part of the evaluation of S^2EC^2O.

- Stefan Gärtner, Jens Bürger, Kurt Schneider, and Jan Jürjens: Zielgerichtete Anpassung von Software nach der Evolution von kontextspezifischem Wissen. In: *1st Collaborative Workshop on Evolution and Maintenance of Long-Living Systems (EMLS)*, 2014. [GBSJ14]

  This publication is the result of a problem statement sent in to a collaborative workshop. An open discussion which has been attended by research as well as industry participants resulted in feedback on the current view on how context knowledge is documented.

- Jens Bürger, Stefan Gärtner, Thomas Ruhroth, Johannes Zweihoff, Jan Jürjens, and Kurt Schneider: Restoring security of long-living systems by co-evolution. In: *COMPSAC 2015*. IEEE Computer Soc., 2015. [BGR+15]

  This publication focuses on how a system model can be co-evolved after a context evolution has taken place. This work firstly shows a co-evolution approach, accompanied by a case study regarding iTrust and privacy law evolution. The approach is evaluated using a case study, showing how iTrust is co-evolved to solve security vulnerabilities introduced by context evolution.

  The author conducted the main part of this research. This includes the co-evolution approach itself, definition of Essential Security Requirements (ESRs) and application as well as implementation of the case study. The implementation is merged in the implementation of S^2EC^2O, *Security Maintenance Rules* is a concept which is also used in the approach.

- Jens Bürger, Daniel Strüber, Stefan Gärtner, Thomas Ruhroth, Jan Jürjens, and Kurt Schneider: A framework for semi-automated co-evolution of security knowledge and system models.
  In: *Journal of Systems and Software*, Elsevier, 2018 [BSG+18].

  This publication presents a deeper insight into the SecVolution design-time approach. It emphasizes on system co-evolution under knowledge evolution by presenting relevant results in conjunction. The publication features an extended version of the upper ontology, a revised definition and examples of Essential Security Requirements (ESR) and a more detailed case study showing modeled knowledge using the upper ontology as well as its evolution. Co-evolution of the system model under consideration is also shown.

  The author of this thesis contributed in substantial manner to this publication by providing a refined Essential Security Requirements notion. Moreover, the case study has been carried out mainly by the author, presenting examples of Essential Security Requirements and making use of the extended upper ontology.

- Cyntia Montserrat Vargas Martinez, Jens Bürger, Fabien Viertel, Birgit Vogel-Heuser, and Jan Jürjens: System evolution through semi-automatic elicitation of security requirements: A Position Paper.
  In: *3rd IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control* (CESCIT), 2018 [VBV+18].

  This publication is a position paper, it comprises ideas of how to enhance the SecVolution design-time approach with run-time relevant information to deal with requirements of the automation domain. Vulnerability assessment and penetration testing are presented as two mechanisms for actively assessing a

given industrial control system. These ought to be matched against vulnerability databases. Outcome of this assessment is envisioned to contribute to the Security Maintenance Model (SMM).

This publication is the first result of an industrial collaboration with an external PhD student. The approach presented is the result of joint discussions. The author of this thesis contributed his expertise gained during his work and the SecVolution project, especially concerning management of context knowledge and possible reactions (co-evolutions) to context evolutions.

- Jan Jürjens, Kurt Schneider, Jens Bürger, Fabien Patrick Viertel, Daniel Strüber, Michael Goedicke, Ralf Reussner, Robert Heinrich, Emre Taşpolatoğlu, Marco Konersmann, Alexander Fay, Winfried Lamersdorf, Jan Ladiges, Christopher Haubeck: Maintaining Security in Software Evolution.
  In: *Managed Software Evolution* (*to appear; Springer Open*).
  Editors: Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin.

  This publication is written by all projects which have participated in the first or second funding period of the DFG priority programme 1593, of which SecVolution has been part of. The part mainly relevant to the results of the SecVolution project have been contributed by Jan Jürjens, Kurt Schneider, Jens Bürger, Fabien Patrick Viertel, and Daniel Strüber. The other authors contributed content coming from other projects of the SPP.

  The author of this thesis contributed to the chapter ranging the design-time co-evolution part of SecVolution into the SPP context. Moreover, a first concept is published on how the SecVolution approach can handle challenges coming from run time. Emphasis is put on an approach of how the system can adapt its behavior at run time to react on sudden odd context behavior occurring at run time.

- Sven Peldszus, Jens Bürger, Jan Jürjens: Reactive Security Monitoring of Java Applications with Round-Trip Engineering.
  Under review [PBJ19].

  This publication introduces an approach to combine security monitoring and round-trip engineering for Java applications. The approach, called *UMLsecRT*, makes use of the UMLsec to annotate a UML model with security annotations, in this case *secure dependency*. As model, a pre-existing one can be used or a model generated from source. Synchronization of annotations between source code and model is supported. UMLsec annotations are proceeded into the source code as Java annotations. A security monitor is executed alongside the actual application, having countermeasures for security violations deployed. Insights gained at run time are fed back into the model. The approach is evaluated using a case study extracted from real attacks and the DaCapo benchmark suite.

  The work introduced in Chapter 7 is based on joint work with Sven Peldszus. At the time of writing, the work is currently under review. The chapter introduces a method for round-trip engineering of security properties. To limit the scope of this thesis, we show the application for one exemplary UMLsec stereotype, namely *secure dependency*. The author of this thesis contributed to the paper by building the approach jointly in numerous workshops. The author of this thesis contributed to the approach by bringing in his expertise in

secure model annotations and UML meta-modeling. Due to its domain spanning requirements ranging from abstract modeling to instrumented code, the implementation was mainly done using pair programming between the author and Sven Peldszus.

Bibliography

[Abb12] Sunitha Abburu. A survey on ontology reasoners and comparison. *International Journal of Computer Applications*, 57(17), 2012.

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Model Driven Engineering Languages and Systems (MoDELS)*, pages 121–135, 2010.

[ACPS16] Albin Ahmeti, Diego Calvanese, Axel Polleres, and Vadim Savenkov. Handling inconsistencies due to class disjointness in SPARQL updates. In Harald Sack, Eva Blomqvist, Mathieu d'Aquin, Chiara Ghidini, Simone Paolo Ponzetto, and Christoph Lange, editors, *The Semantic Web. Latest Advances and New Domains*, pages 387–404. Springer International Publishing, 2016.

[ADET12] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. Achieving dynamic adaptation via management and interpretation of runtime models. *Journal of Systems and Software*, 85(12):2720–2737, 2012.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[AMR07] Omar H. Alhazmi, Yashwant K. Malaiya, and Indrajit Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.

[APRJ17] Amir Shayan Ahmadian, Sven Peldszus, Qusai Ramadan, and Jan Jürjens. Model-based privacy and security analysis with CARiSMA. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 989–993, 2017.

[ARS15] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23, May 2015.

[BA14] Mourad Bouneffa and Adeel Ahmad. The change impact analysis in BPM based software applications: A graph rewriting and ontology based approach. In *Enterprise Information Systems*, pages 280–295. Springer, 2014.

[BBFS05] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. *Web and Semantic Web Query Languages: A Survey*, pages 35–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[BET12] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, May 2012.

[BfD14] BfDI. BDSG Änderungen, 2014. http://www.bfdi.bund.de/bfdi_wiki/index.php/BDSG_%C3%84nderungen (accessed: Apr 14th, 2019).

[BGR+15] Jens Bürger, Stefan Gärtner, Thomas Ruhroth, Johannes Zweihoff, Jan Jürjens, and Kurt Schneider. Restoring security of long-living systems by co-evolution. In *39th Annual IEEE Computer Software and Applications Conf. (COMPSAC 2015)*. IEEE Computer Soc., 2015. 6 pp.

[BHL+07] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative runtime verification with tracematches. In *Proceedings of the 7th International Workshop on Runtime Verification (RV)*, pages 22–37, 2007.

[BJR+14] Jens Bürger, Jan Jürjens, Thomas Ruhroth, Stefan Gärtner, and Kurt Schneider. Model-based security engineering with UML: Managed co-evolution of security knowledge and software models. In *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, page 282, 2014.

[BJW15] Jens Bürger, Jan Jürjens, and Sven Wenzel. Restoring security of evolving software models using graph transformation. *STTT*, 17(3):267–289, 2015.

[BKB+07] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571–583, 2007.

[BM04] Patrick Brézillon and Ghita Kouadri Mostéfaoui. Context-based security policies: A new modeling approach. In *2nd IEEE Conf. on Pervasive Computing and Communications Workshops (PerCom 2004 Workshops)*, pages 154–158, 2004.

[BOAI14] Doaa Saleh Abobakr Baras, Siti Hajar Othman, Mohammad Nazir Ahmad, and Norafida Ithnin. Towards managing information security knowledge through metamodelling approach. In *International Symposium on Biometrics and Security Technologies (ISBAST)*, pages 310–315. IEEE, 2014.

[BOCB+17] Lotfi Ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, and Achim D. Brucker. Time for addressing software security issues: Prediction models and impacting factors. *Data Science and Engineering*, 2(2):107–124, 2017.

[Bor97] Willem Nico Borst. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, University of Twente, Netherlands, Sep 1997.

[Bow13] Sue Bowman. Impact of electronic health record systems on information integrity: Quality and safety implications. *Perspectives in Health Information Management*, 10(Fall), October 2013.

[BPP02] Patrick Brézillon, Laurent Pasquier, and Jean-Charles Pomerol. Reasoning with contextual graphs. *European Journal of Operational Research*, 136(2):290–298, 2002.

[Bri] British Parliament. Data Protection Act 1998.

[BSG+18] Jens Bürger, Daniel Strüber, Stefan Gärtner, Thomas Ruhroth, Jan Jürjens, and Kurt Schneider. A framework for semi-automated co-evolution of security knowledge and system models. *Journal of Systems and Software*, 139:142–160, 2018.

[BSS+11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 241–250, 2011.

[Bun] Bundesamt für Sicherheit in der Informationstechnik (BSI). IT-Grundschutz. https://www.bsi.bund.de/DE/Themen/ITGrundschutz/itgrundschutz_node.html (accessed: Apr 14th, 2019).

[Bun05] Bundesministerium des Inneren. Bundesdatenschutzgesetz. Bundesgesetzblatt, September 2005.

[Bun18] Bundesamt für Sicherheit in der Informationstechnik (BSI). BSI TR-02102-1 Kryptographische Verfahren: Empfehlungen und Schlüssellängen, February 2018. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf (accessed: Apr 14th, 2019).

[CEK+08] Luca Compagna, Paul El Khoury, Alžběta Krausová, Fabio Massacci, and Nicola Zannone. How to integrate legal requirements into a requirements engineering methodology for the development of security and privacy patterns. *Artificial Intelligence and Law*, 17(1):1–30, November 2008.

[CFO+17] Daniela Soares Cruzes, Michael Felderer, Tosin Daniel Oyetoyan, Matthias Gander, and Irdin Pekaric. How is security testing done in agile teams? A cross-case analysis of four software teams. In Hubert Baumeister, Horst Lichter, and Matthias Riebisch, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 201–216, Cham, 2017. Springer International Publishing.

[Chi] Shigeru Chiba. Javassist. http://www.javassist.org (accessed: Apr 14th, 2019).

[Chi00] Shigeru Chiba. Load-time structural reflection in Java. In *ECOOP 2000 - Object-Oriented Programming, 14th European Conference*, pages 313–336, 2000.

[CM04] Brian Chess and Gary McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[CMM+10] Gabriele Costa, Fabio Martinelli, Paolo Mori, Christian Schaefer, and Thomas Walter. Runtime monitoring for next generation Java ME platform. *Computers & Security*, 29(1):74–87, 2010.

[Crn10] Gordana Dodig Crnkovic. *Constructive Research and Info-computational Knowledge Generation*, pages 359–380. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Day94] Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[Deu18] Deutsche Presseagentur (DPA). Bahn schränkt Zahlungsarten für Ticketkauf ein, Dec 20th, 2018. https://www.faz.net/agenturmeldungen/dpa/bahn-schraenkt-zahlungsarten-fuer-ticketkauf-ein-15951983.html (accessed: Apr 14th, 2019).

[DFG] DFG. DFG SPP 1593 website. https://www.dfg-spp1593.de (accessed: Apr 14th, 2019).

[DHMM10] Éric Dubois, Patrick Heymans, Nicolas Mayer, and Raimundas Matulevičius. A systematic approach to define the domain of information system security risk management. In *Intentional Perspectives on Information Systems Engineering*, pages 289–306. Springer, 2010.

[DKA+14] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[DRW+18] Dongdong Du, Xingzhang Ren, Yupeng Wu, Jien Chen, Wei Ye, Jinan Sun, Xiangyu Xi, Qing Gao, and Shikun Zhang. Refining traceability links between vulnerability and software component in a vulnerability knowledge graph. In Tommi Mikkonen, Ralf Klamma, and Juan Hernández, editors, *Web Engineering*, pages 33–49, Cham, 2018. Springer International Publishing.

[DSRdV+07] Geiza Maria Hamazaki Da Silva, Alexandre Rademaker, Davi Romero de Vasconcelos, Fernando Náufel do Amaral, Carlos Bazílio, Vaston G. Costa, and Edward Hermann Haeusler. Dealing with the formal analysis of information security policies through ontologies: A case study. In *Proceedings of the Third Australasian Workshop on Advances in Ontologies-Volume 85*, pages 55–60. Australian Computer Society, Inc., 2007.

[EBM11] Neil A. Ernst, Alexander Borgida, and John Mylopoulos. Requirements evolution drives software evolution. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 16–20. ACM, 2011.

[Ecl13] Eclipse contributors. Workbench User Guide – Secure Storage – How secure storage works. Technical report, The Eclipse Foundation, 2013. https://help.eclipse.org/photon/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Freference%2Fref-securestorage-works.htm (accessed Apr 14th, 2019).

[eDT] eMoflon Developer Team. eMoflon – A tool for building tools. http://www.emoflon.org/ (accessed: Apr 14th, 2019).

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[EPS10] Antti Evesti and Susanna Pantsar-Syväniemi. Towards micro architecture for security adaptation. In *4th European Conference on Software Architecture (ECSA 2010)*, pages 181–188. ACM, 2010. Companion Volume.

[ERW08] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph technology in reverse engineering: The TGraph approach. In *10th Workshop Software Reengineering, 5-7 May 2008, Bad Honnef, Germany*, pages 67–81, 2008.

[EU] EU. ViSion - the new privacy solution for citizen and public administration. https://www.visioneuproject.eu (accessed: Jun 13th, 2019).

[EU 95] EU Parliament. Directive 95/46/EC of the European Parliament and of the council of 24 October 1995. *Official Journal of the European Union*, L 281:0031–0050, 1995.

[EU 16] EU Parliament. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, L119:1–88, May 2016.

[Eva04] Eric Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[EYZ09] Golnaz Elahi, Eric Yu, and Nicola Zannone. A vulnerability-centric requirements engineering framework: analyzing security attacks, countermeasures, and requirements based on vulnerabilities. *Requirements Engineering*, 15(1):41–62, 2009.

[Far12] Stefan Farfeleder. *Requirements Specification and Analysis for Embedded Systems*. PhD thesis, TU Wien, Austria, 2012.

[FGO+11] Xavier Franch, Paul Grünbacher, Marc Oriol, Benedikt Burgstaller, Deepak Dhungana, Lidia Lopez, Jordi Marco, and João Pimentel. Goal-driven adaptation of service-based systems from runtime monitoring data. In *Computer Software and Applications Conf. Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 458–463, July 2011.

[FLS+14] Rudolf Ferenc, László Langó, István Siket, Tibor Gyimóthy, and Tibor Bakota. Source meter sonar qube plug-in. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 77–82, Sep. 2014.

[FNO+93] Gerhard Fischer, Kumiyo Nakakoji, Jonathan Ostwald, Gerry Stahl, and Tamara Sumner. Embedding critics in design environments. *The Knowledge Engineering Review*, 8(4):285–307, 1993.

[Gal] Galigator Github account. Openllet. https://github.com/Galigator/openllet (accessed: Apr 14th, 2019).

[Gär16] Stefan Gärtner. *Heuristische und wissensbasierte Sicherheitsprüfung von Softwareentwicklungsartefakten basierend auf natürlichsprachlichen Informationen*. PhD thesis, University of Hanover, Hannover, Germany, 2016.

[GBSJ14] Stefan Gärtner, Jens Bürger, Kurt Schneider, and Jan Jürjens. Zielgerichtete Anpassung von Software nach der Evolution von kontextspezifischem Wissen. In *1st Collaborative Workshop on Evolution and Maintenance of Long-Living Systems (EMLS)*, 2014.

[GMP06] Stephan Grimm, Boris Motik, and Chris Preist. Matching semantic service descriptions with local closed-world reasoning. In *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings*, pages 575–589, 2006.

[GRB+14] Stefan Gärtner, Thomas Ruhroth, Jens Bürger, Kurt Schneider, and Jan Jürjens. Maintaining requirements for long-living software systems by incorporating security knowledge. In *22nd IEEE Int. Requirements Eng. Conference*. IEEE, 2014.

[GRB+15] Stefan Gärtner, Thomas Ruhroth, Jens Bürger, Kurt Schneider, and Jan Jürjens. Towards maintaining long-living information systems by incorporating evolving security knowledge. In *Software Engineering (SE 2015)*, Lecture Notes in Informatics. GI, 2015.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[Gru95] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*, 43(5-6):907–928, 1995.

[GSM] GSM Association. N2020.01 – VoLTE service description and implementation guidelines (version 1.0). http://www.gsma.com/network2020/wp-content/uploads/2015/03/N2020.01-v1.0.pdf (accessed: Apr 14th, 2019).

[GW07] Michael Gegick and Laurie Williams. On the design of more secure software-intensive systems by use of attack patterns. *Information and Software Technology*, 49(4):381–397, April 2007.

[HG03] H.M. Harmain and Robert Gaizauskas. CM-Builder: A natural language-based case tool for object-oriented analysis. *Automated Software Engineering*, 10:157–181, 2003.

[HIK+10] Siv Hilde Houmb, Shareeful Islam, Eric Knauss, Jan Jürjens, and Kurt Schneider. Eliciting security requirements and tracing them to design: An integration of Common Criteria, heuristics, and UMLsec. *Requirements Engineering Journal*, 15(1):63–93, March 2010.

[HKSS08] Piotr Habela, Krzysztof Kaczmarski, Krzysztof Stencel, and Kazimierz Subieta. OCL as the query language for UML model execution. In *International Conference on Computational Science (ICCS)*, volume 5103, pages 311–320. Springer Berlin Heidelberg, 2008.

[HKW+07] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. CoCoME. In *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 16–53, 2007.

[HLJA10] Sebastian Höhn, Lutz Lowis, Jan Jürjens, and Rafael Accorsi. Identification of vulnerabilities in web services using model-based security. In *Web Services Security Development and Architecture: Theoretical and Practical Issues*, chapter 1, pages 1–32. Information Science Reference, Hershey, 2010.

[HLMN08] Charles B. Haley, Robin C. Laney, Jonathan D. Moffett, and Bashar Nuseibeh. Security requirements engineering: A framework for representation and analysis. *IEEE Trans. Software Eng.*, 34(1):133–153, 2008.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004.

[HS06] Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering (SWESE)*, 2006.

[HSD07] Almut Herzog, Nahid Shahmehri, and Claudiu Duma. An ontology of information security. *International Journal of Information Security and Privacy (IJISP)*, 1(4):1–23, 2007.

[HTMM08] Guillaume Hiet, Valerie Viet Triem Tong, Ludovic Me, and Benjamin Morin. Policy-based Intrusion Detection in Web Applications by Monitoring Java Information Flows. In *Proceedings of the 3rd International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pages 53–60, 2008.

[HWP+14] Thorsten Humberg, Christian Wessel, Daniel Poggenpohl, Sven Wenzel, Thomas Ruhroth, and Jan Jürjens. Using ontologies to analyze compliance requirements of cloud-based processes. In *Cloud Computing and Services Science (selected best papers)*, volume 453 of *Communications in Computer and Information Science*, pages 1–16. Springer, 2014.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web (WWW)*, pages 40–52, 2004.

[i3l] i3labs. Architexta website. http://www.architexa.com/ (accessed: Apr 14th, 2019).

[IDC07] Iulia Ion, Boris Dragovic, and Bruno Crispo. Extending the Java virtual machine to enforce fine-grained security policies in mobile devices. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, pages 233–242, 2007.

[Int07] International Standardization Organization (ISO). ISO 15408:2007 Common Criteria for information technology security evaluation, version 3.1, revision 2, CCMB-2007-09-001, CCMB-2007-09-002 and CCMB-2007-09-003, September 2007.

[Int17] International information system security certification consortium ISC^2. 2017 global information security workforce study, 2017.

[ISO16] ISO/IEC. ISO/IEC 9075:2016 Information technology - Database languages - SQL, 2016.

[JJ05] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[JKM+15] Ravi Jhawar, Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Rolando Trujillo-Rasua. Attack trees with sequential conjunction. In *IFIP International Information Security Conference (SEC)*, pages 339–353. Springer, 2015.

[JLFJ13] Arnav Joshi, Ravendar Lal, Tim Finin, and Anupam Joshi. Extracting cybersecurity related linked data from text. In *Semantic Computing (ICSC), 2013 IEEE Seventh International Conference on*, pages 252–259. IEEE, 2013.

[JS14] Jan Jürjens and Kurt Schneider. The SecReq approach: From security requirements to secure design while managing software evolution. In *Software Engineering (SE2014)*, volume Lecture Notes in Informatics. GI, 2014.

[KC07] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical report, EBSE-2007-01, 2007.

[KKOS12] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. Understanding model evolution through semantically lifting model differences with SiLift. In *Proc. of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 638 –641, 2012.

[KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 163–172, 2011.

[KKT13] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-preserving edit scripts in model versioning. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 191–201, 2013.

[KLM09] Eric Knauss, Daniel Lübke, and Sebastian Meyer. Feedback-driven requirements engineering: The heuristic requirements assistant. In *Int.*

Conf. on Software Eng. (ICSE'09), Formal Research Demonstrations Track, pages 587 – 590, Vancouver, Canada, 2009.

[KM14] Andreas Kuehn and Milton Mueller. Shifts in the cybersecurity paradigm: Zero-day exploits, discourse, and emerging institutions. In *New Security Paradigms Workshop (NSPW)*, pages 63–68. ACM, 2014.

[KSOK13] Haruhiko Kaiya, Junya Sakai, Shinpei Ogata, and Kenji Kaijiri. Eliciting security requirements for an information system using asset flows and processor deployment. *International Journal of Secure Software Engineering*, 4(3):42–63, 2013.

[LAS14] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Developing eMoflon with eMoflon. In *Proceedings of the 7th International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 138–145, 2014.

[LAS17] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Inter-model consistency checking using triple graph grammars and linear optimization techniques. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 191–207, 2017.

[LF15] Gabriel Lawrence and Chris Frohoff. Marshalling pickles: how deserializing objects can ruin your day, 2015. https://www.slideshare.net/frohoff1/appseccali-2015-marshalling-pickles (accessed: Apr 14th, 2019).

[LFR13] Steffen Lehnert, Qurat-ul-ann Farooq, and Matthias Riebisch. Rule-Based Impact Analysis for Heterogeneous Software Artifacts. *2013 17th European Conference on Software Maintenance and Reengineering*, pages 209–218, March 2013.

[LHR17] Sungho Lee, Sungjae Hwang, and Sukyoung Ryu. All about Activity Injection: Threats, Semantics, and Detection. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 252–262, 2017.

[LNHE14] Steffen Lohmann, Stefan Negru, Florian Haag, and Thomas Ertl. VOWL 2: User-oriented visualization of ontologies. In Krzysztof Janowicz, Stefan Schlobach, Patrick Lambrix, and Eero Hyvönen, editors, *Knowledge Engineering and Knowledge Management*, pages 266–281, Cham, 2014. Springer International Publishing.

[LP13] Ivan Launders and Simon Polovina. A semantic approach to security policy reasoning. In *Strategic Intelligence Management*, pages 150–166. Elsevier, 2013.

[LR03] Meir M. Lehman and Juan F. Ramil. Software evolution – Background, theory, practice. *Information Processing Letters*, 2003.

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, pages 139–160, 2005.

[LZSL13] Bixin Li, Qiandong Zhang, Xiaobing Sun, and Hareton Leung. Using water wave propagation phenomenon to study software change impact analysis. *Advances in Engineering Software*, 58:45–53, 2013.

[Mar03] Peter Marwedel. *Embedded system design*. Kluwer, 2003.

[MARS12] Sebastian Meyer, Anna Averbakh, Torsten Ronneberger, and Kurt Schneider. Experiences from establishing knowledge management in a joint research project. In Oscar Dieste, Andreas Jedlitschka, and Natalia Juristo, editors, *Product-Focused Software Process Improvement*, volume 7343 of *Lecture Notes in Computer Science*, pages 233–247, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[MEHDH13] Frank D. Macías-Escrivá, Rodolfo Haber, Raul Del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications, 2013.

[Men02] Tom Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, May 2002.

[MGM03] Haralambos Mouratidis, Paolo Giorgini, and Gordon Manson. An ontology for modelling security: The Tropos approach. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES)*, pages 1387–1394. Springer, 2003.

[MH05] Hugh McManus and Daniel Hastings. A framework for understanding uncertainty and its mitigation and exploitation in complex systems. In *INCOSE International Symposium*, volume 15, pages 484–503. Wiley Online Library, 2005.

[MITa] MITRE. CAPEC – common attack pattern enumeration and classification. https://capec.mitre.org/ (accessed: Apr 14th, 2019).

[MITb] MITRE. CVE – common vulnerabilities and exposures. https://cve.mitre.org (accessed: Apr 14th, 2019).

[MIT17a] MITRE. CVE-2017-7525, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7525 (accessed: Apr 14th, 2019).

[MIT17b] MITRE. CWE – Common Weakness Enumeration, 2017. https://cwe.mitre.org/ (accessed: Apr 14th, 2019).

[MLH+15] Nicolas Matentzoglu, Jared Leo, Valentino Hudhra, Uli Sattler, and Bijan Parsia. A survey of current, stand-alone OWL reasoners. In *ORE*, pages 68–79, 2015.

[MMF+10] Brice Morin, Tejeddine Mouelhi, Franck Fleurey, Yves Le Traon, Olivier Barais, and Jean-Marc Jézéquel. Security-driven model-based dynamic adaptation. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 205–214, 2010.

[MMG] Chris Mungall, Eleni Mikroyannidi, and Rafael Gonçalves. OWL API. https://github.com/owlcs/owlapi (accessed: Apr 14th, 2019).

[MNG+10] André Miede, Nedislav Nedyalkov, Christian Gottron, André König, Nicolas Repp, and Ralf Steinmetz. A generic metamodel for IT security attack modeling for distributed systems. In *International Conference on Availability, Reliability, and Security (ACES)*, pages 430–437. IEEE, 2010.

[MNGL98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(2):158–191, 1998.

[MSW] Andrew Meneely, Ben Smith, and Laurie Williams. iTrust website. http://agile.csc.ncsu.edu/iTrust (accessed Apr 14th, 2019).

[MSW12] Andrew Meneely, Ben Smith, and Laurie Williams. iTrust electronic health care system case study. In *Software and Systems Traceability*, pages 425–438. Springer, 2012.

[Nat] National Institute of Standards and Technology (NIST). National vulnerability database. https://nvd.nist.gov/ (accessed: Apr 14th, 2019).

[NCLM06] Natalya F. Noy, Abhita Chugh, William Liu, and Mark A. Musen. A framework for ontology evolution in collaborative environments. In *Proc. of the 5th ISWC*, pages 544–558, 2006.

[Neo] Neo4j, Inc. Neo4j graph platform. https://neo4j.com/ (accessed: Apr 14th, 2019).

[NVLG14] Tuong Huan Nguyen, Bao Quoc Vo, Markus Lumpe, and John Grundy. KBRE: a framework for knowledge-based requirements engineering. *Software Quality Journal*, 22(1):87–119, 2014.

[NYZ+15] Armstrong Nhlabatsi, Yijun Yu, Andra Zisman, Thein Tun, Niamul Khan, Arosha Bandara, and Bashar Nuseibeh. Managing security control assumptions using causal traceability. In *8th Int. Symp. on Software and Systems Traceability (SST)*, 2015.

[Obj16] Object Management Group. Meta Object Facility specification version 2.5.1, 2016. https://www.omg.org/spec/MOF/2.5.1/ (accessed: May 27th, 2019).

[Obj17] Object Management Group (OMG). UML 2.5.1 superstructure specification. Technical Report OMG Document Number: formal/2011-08-06, Object Management Group (OMG), 2017.

[OCS+13] Inah Omoronyia, Luca Cavallaro, Mazeiar Salehie, Liliana Pasquale, and Bashar Nuseibeh. Engineering adaptive privacy: On the role of privacy awareness requirements. *Proc. - Int. Conf. on Software Engineering*, pages 632–641, 2013.

[OD09] Martin J. O'Connor and Amar K. Das. SQWRL: A query language for OWL. In *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23-24, 2009*, 2009.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[OKY11] Takao Okubo, Haruhiko Kaiya, and Nobukazu Yoshioka. Effective security impact analysis with patterns for software enhancement. *2011 Sixth International Conference on Availability, Reliability and Security*, pages 527–534, August 2011.

[OKY12] Takao Okubo, Haruhiko Kaiya, and Nobukazu Yoshioka. Analyzing impacts on software enhancement caused by security design alternatives with patterns. *International Journal of Secure Software Engineering*, 3(1):37–61, 2012.

[OPS+12] Inah Omoronyia, Liliana Pasquale, Mazeiar Salehie, Luca Cavallaro, Gavin J. Doherty, and Bashar Nuseibeh. Caprice: a tool for engineering adaptive privacy. *Proc. of the 27th IEEE/ACM Int. Conf. on Automated Software Eng. - ASE 2012*, page 354, 2012.

[Ora] Oracle. Java Agent API. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html (accessed: Apr 14th, 2019).

[OWA] OWASP. O-Saft. https://www.owasp.org/index.php/O-Saft (accessed: Apr 14th, 2019).

[OWL09] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, October 2009. Available at http://www.w3.org/TR/owl2-overview/.

[PBJ19] Sven Peldszus, Jens Bürger, and Jan Jürjens. Reactive Security Monitoring of Java Applications with Round-Trip Engineering. Under Review, 2019.

[PKLS15] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. Incremental co-evolution of Java programs based on bidirectional graph transformation. In *Proceedings of the 12th Principles and Practices of Programming on The Java Platform (PPPJ)*, pages 138–151, 2015.

[PMG+16] Bijan Parsia, Nicolas Matentzoglu, Rafael S. Gonçalves, Birte Glimm, and Andreas Steigmiller. The OWL reasoner evaluation (ORE) 2015 resources. In *International Semantic Web Conference*, pages 159–167. Springer, 2016.

[PMS+12] Liliana Pasquale, Claudio Menghi, Mazeiar Salehie, Luca Cavallaro, Inah Omoronyia, and Bashar Nuseibeh. SecuriTAS: a tool for engineering adaptive security. In *20th ACM SIGSOFT Symp. on the Foundations of Software Eng. (FSE-20)*, page 19, 2012.

[Pop15] Andrei Popov. RFC 7465: Prohibiting RC4 cipher suite, Feb 2015. https://tools.ietf.org/html/rfc7465 (accessed: Apr 14th, 2019).

[PS04] Bijan Parsia and Evren Sirin. Pellet: An OWL-DL reasoner. In *Third international semantic web conference-poster*, volume 18, pages 1–2, 2004.

[RGB+14a] Thomas Ruhroth, Stefan Gärtner, Jens Bürger, Jan Jürjens, and Kurt Schneider. Versioning and evolution requirements for model-based system development. In *International Workshop on Comparison and Versioning of Software Models (CVSM 2014)*, volume 34/2 of *Softwaretechnik-Trends*, pages 20–24, 2014.

[RGB+14b] Thomas Ruhroth, Stefan Gärtner, Jens Bürger, Jan Jürjens, and Kurt Schneider. Towards adaptation and evolution of domain-specific knowledge for maintaining secure systems. In *15th Int. Conf. of Product Focused Software Development and Process Improvement (PROFES'14)*, volume 8892 of *LNCS*, pages 239–253. Springer, 2014.

[RJ12] Thomas Ruhroth and Jan Jürjens. Supporting security assurance in the context of evolution: Modular modeling and analysis with UMLsec. In *IEEE: 14th Int. Symp. on High-Assurance Systems Eng. (HASE 2012)*. IEEE CS, October 2012.

[RPZ10] Yuan Ren, Jeff Z. Pan, and Yuting Zhao. Closed world reasoning for OWL2 with NBox. *Tsinghua Science & Technology*, 15(6):692 – 701, 2010.

[SBG+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A usability-focused framework for EMF model transformation development. In *ICGT*, pages 196–208, 2017.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[Sch06] Kurt Schneider. Rationale as a by-product. In Allen H. Dutoit, Raymond McCall, Ivan Mistrik, and Barbara Paech, editors, *Rationale Management in Software Engineering*, pages 91–109. Springer, 2006.

[Sch11] Kurt Schneider. Focusing spontaneous feedback to support system evolution. In *Proc. of IEEE 19th Int. Requirements Eng. Conf. (RE'11)*, pages 165–174, Trento, Italy, 2011. IEEE.

[Set12] Burr Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, 2012.

[SLAM13] Vítor E. Silva Souza, Alexei Lapouchnian, Konstantinos Angelopoulos, and John Mylopoulos. Requirements-driven software evolution. *Computer Science-Research and Development*, 28(4):311–329, 2013.

[SLWZ13] Xiaobing Sun, Bixin Li, Wanzhi Wen, and Sai Zhang. Analyzing impact rules of different change types to support change impact analysis. *International Journal of Software Engineering and Knowledge Engineering*, 23(03):259–288, 2013.

[SO05] Guttorm Sindre and Andreas Lothe Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering Journal*, 10(1):34–44, 2005.

[SPL18] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. SYNODE: Understanding and automatically preventing injection attacks on Node.js. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[SPO+12] Mazeiar Salehie, Liliana Pasquale, Inah Omoronyia, Raian Ali, and Bashar Nuseibeh. Requirements-driven adaptive security: Protecting variable assets at runtime. *2012 20th IEEE Int. Requirements Eng. Conf. (RE)*, pages 111–120, September 2012.

[SS04] Frank Swiderski and Window Snyder. *Threat modeling*. Microsoft Press, 2004.

[SSFS11] Ansgar Scherp, Carsten Saathoff, Thomas Franz, and Steffen Staab. Designing core ontologies. *Applied Ontology*, 6(3):177–221, 2011.

[SSK08] Kurt Schneider, Kai Stapel, and Eric Knauss. Beyond documents: Visualizing informal communication. In *Proceedings of Third International Workshop on Requirements Engineering Visualization (REV '08)*, Barcelona, Spain, November 2008.

[Sta] Stanford Center for Biomedical Informatics Research (BMIR). Protégé website. http://protege.stanford.edu (accessed: Apr 14th, 2019).

[SZS10] Igor Siveroni, Andrea Zisman, and George Spanoudakis. A UML-based static verification framework for security. *Requirements Engineering*, 15(1):95–118, 2010.

[TG06] Bill Tsoumas and Dimitris Gritzalis. Towards an Ontology-based Security Management. In *Proc. of the 20th Int. Conference on Advanced Information Networking and Applications (AINA)*, volume 1, pages 985–992. IEEE, 2006.

[Thea] The Apache Software Foundation. Apache Jena. http://jena.apache.org/ (accessed: Apr 14th, 2019).

[Theb] The Eclipse Foundation. MoDisco website. https://www.eclipse.org/MoDisco/ (accessed: Apr 14th, 2019).

[Thec] The Eclipse Foundation. Papyrus Modeling Environment. https://www.eclipse.org/papyrus/ (accessed: Apr 14th, 2019).

[Thed] The OWASP Foundation. OWASP – open web application security project website. https://www.owasp.org (accessed: Apr 14th, 2019).

[Ton05] Paolo Tonella. Reverse engineering of object oriented code. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 724–725, 2005.

[TYB+18] Thein Than Tun, Mu Yang, Arosha K. Bandara, Yijun Yu, Armstromg Nhlabatsi, Niamul Khan, Khaled M. Khan, and Bashar Nuseibeh. Requirements and specifications for adaptive security. *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems - SEAMS '18*, pages 161–171, 2018.

[UBAH+15] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, Part 1:80 – 99, 2015.

[Ul 14] Anees Ul Mehdi. *Epistemic Reasoning in OWL 2 DL*. PhD thesis, Karlsruhe Institute of Technology, 2014.

[VBV+18] Cyntia Vargas, Jens Bürger, Fabien Viertel, Birgit Vogel-Heuser, and Jan Jürjens. System evolution through semi-automatic elicitation of security requirements: A Position Paper. *IFAC-PapersOnLine*, 51(10):64 – 69, 2018. 3rd IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control CESCIT 2018.

[Vis] Visual Paradigm. Visual Paradigm website. https://www.visual-paradigm.com/ (accessed: Apr 14th, 2019).

[VJL+14] Colin C. Venters, Caroline Jay, Lydia Lau, Michael K. Griffiths, Violeta Holmes, Rupert R. Ward, Jim Austin, Charlie Dibsdale, and Jie Xu. Software sustainability: The modern tower of babel. In *Proceedings of the Third International Workshop on Requirements Engineering for Sustainable Systems, RE4SuSy 2014, co-located with 22nd International Conference on Requirements Engineering (RE 2014), Karlskrona, Sweden, August 26, 2014.*, pages 7–12, 2014.

[VP15] Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium, USENIX Security 15*, pages 97–112, 2015.

[W3Ca] W3C OWL Working Group. MOF-Based metamodel. https://www.w3.org/2007/OWL/wiki/MOF-Based_Metamodel (accessed: Apr 14th, 2019).

[W3Cb] W3C OWL Working Group. OWL 2 web ontology language manchester syntax. https://www.w3.org/TR/owl2-manchester-syntax/ (accessed: Apr 14th, 2019).

[W3Cc] W3C SPARQL Working Group. SPARQL 1.1 Overview. https://www.w3.org/TR/sparql11-overview/ (accessed: Apr 14th, 2019).

[W3Cd] W3C SPARQL Working Group. SPARQL 1.1 Update. https://www.w3.org/TR/sparql11-update/ (accessed: Apr 14th, 2019).

[WSR10] Tobias Walter, Hannes Schwarz, and Yuan Ren. Y.: Establishing a bridge from graph-based modeling languages to ontology languages. In *In: Proceedings of 3rd Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE). Volume CEUR of 604., CEUR-WS.org*, 2010.

[WYY05] Xiaoyun Wang, Lisa Yin Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology–CRYPTO 2005*, pages 17–36. Springer, 2005.

[Xia09] Liang Xiao. An adaptive security model using agent-oriented MDA. *Information and Software Technology*, 51(5):933–955, 2009.

[YSJ12] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. Change patterns. *Software & Systems Modeling*, 13(2):625–648, August 2012.